Université de Montréal

Développement d'un serveur LSP pour Typer

par

Soilihi BEN SOILIHI BOINA

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

28 avril 2022

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Développement d'un serveur LSP pour Typer

présenté par

Soilihi BEN SOILIHI BOINA

a été évalué par un jury composé des personnes suivantes :

Michalis Famelis
(président-rapporteur)

Stefan Monnier
(directeur de recherche)

Marc Feeley
(membre du jury)

Résumé

Programmer en un langage de programmation peut être une tâche ardue. Même les plus chevronnés ne sont pas à l'abri de commettre des erreurs. Il est donc important pour les programmeurs d'avoir des aides pour écrire leur code plus efficacement et plus rapidement. Typer est un jeune langage de programmation en développement. Actuellement, le langage a beaucoup de limitations d'aides pour les programmeurs. En effet, on ne peut coder en Typer que dans un terminal, dans un fichier et compiler le fichier ou dans des environnements primitifs. On apporte une solution à ce problème en offrant, dans ce travail, un serveur LSP au langage qui va offrir des fonctionnalités comme la complétion de code, le surlignement des erreurs, etc, pour permettre aux programmeurs Typer de coder plus facilement et de pouvoir le faire dans leur éditeur/IDE préféré.

Mots clé : OCaml, serveur LSP, JSON-RPC.

Abstract

Programming in a programming language can be a daunting task. Even the most seasoned are not immune to make mistakes. It is therefore important for programmers to have helpers to write their code more efficiently and quickly. Typer is a young programming language in development. Currently, the language has a lot of helper limitations for programmers. Indeed, we can code in Typer only in a terminal, in a file and compile the file or in primitive environments. We solve this problem by offering in this work, an LSP server to the language which will offer features such as code completion, error highlighting...etc, to allow Typer programmers to code more easily and efficiently, and also, to be able to do it in their favorite editor/IDE.

Keywords : OCaml, LSP server, JSON-RPC.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et des abréviations

GHz	*GigaHertz*
HTML	*Hypertext Markup Language*
IDE	*Integrated Development Environment*
JSON-RPC	*JavaScript Object Notation Remote Procedure Call*
KB	*Kilobyte*
lexp	*Lambda Expression*
LSP	*Language Server Protocol*
MB	*Megabyte*
ML	*Meta Language*
PHP	*Pre Hypertext Preprocessor*

sexp	*Symbolic Expression*
SML	*Standard Meta Language*
VS Code	*Visual Studio Code*

Remerciements

J'ai réservé plus particulièrement mes profonds remerciements à mon directeur de recherche, monsieur Stefan Monnier, pour sa patience et son accompagnement éclairé et bienveillant tout le long de ce travail. Je remercie également mes parents et toutes les personnes qui, de près ou de loin, m'ont soutenu le long de mon parcours.

Chapitre 1

INTRODUCTION

1.1. Motivations, problème et solution

Notre travail consiste à concevoir un serveur pour le langage Typer. Le serveur a pour dessein de faciliter la programmation dans le langage.

1.1.1. Motivations et problème

Typer est un langage de programmation expérimental actuellement en phase de développement. À l'heure actuelle, on peut déjà programmer en Typer, mais le langage n'est pas encore publié. Toutefois, on le fait avec beaucoup de limitations. En effet, on peut programmer directement dans un terminal, ou écrire dans un fichier et compiler, mais on n'a aucune aide pour écrire son code. On peut écrire dans un simple fichier avec un éditeur, mais les erreurs ne sont pas surlignées, on n'a pas d'informations sur les types, ni de complétion de code, et toutes les facilités habituelles qu'on a de nos jours quand on programme dans un langage. Aussi, les programmeurs sont habitués à travailler avec leur outil de développement favori, et ce dernier n'offre aucune aide pour programmer en Typer. Or, on sait que programmer dans un langage de programmation n'est pas une tâche facile. Même lorsqu'on est expérimenté dans le langage, on n'est jamais à l'abri d'une erreur ou de l'oubli du nom d'une certaine expression qu'on aimerait écrire. Et pour les programmeurs novices qui apprennent le langage, c'est encore beaucoup moins évident. L'objet du projet est de fournir un serveur à Typer pour permettre aux programmeurs de programmer plus facilement en Typer en leur offrant beaucoup de fonctionnalités comme la complétion de texte, la détection des erreurs, les informations sur les types, etc. Le serveur aidera les programmeurs chevronnés comme les novices à écrire leur code. Idéalement, on voudrait que le serveur fonctionne avec le plus d'outils clients possibles, sans faire de distinction entre les IDEs et les éditeurs. Ainsi, chaque programmeur pourra utiliser ce même serveur avec l'outil de développement de son choix pour programmer en Typer.

1.1.2. La solution d'un serveur LSP

Les éditeurs de texte modernes et les IDEs facilitent beaucoup l'écriture du code avec les nombreuses fonctionnalités qu'ils proposent. Ils font de la coloration syntaxique en fonction de la signification des mots dans le texte, offrent la complétion du texte par exemple. Les IDEs offrent plus de fonctionnalités que les éditeurs, mais sont plus restreints à certains langages. Quant aux éditeurs, ils sont plus généralistes, mais ont moins de fonctionnalités. Les deux aident à écrire du code. Dans la programmation de nos jours, les programmeurs ont pris la coutume d'avoir beaucoup de facilités d'écriture de code, à savoir des fonctionnalités comme le surlignement des erreurs, la vue des informations sur l'élément sur lequel se positionne la souris, la complétion de texte quand on commence à écrire quelque chose, le saut à la position où est défini initialement un élément dans le texte, la coloration syntaxique, le formatage du texte et plein d'autres fonctionnalités. Pour fournir ces avantages, l'une des solutions est l'utilisation d'un serveur LSP. L'IDE comme l'éditeur a besoin des résultats du compilateur parce qu'il a besoin de ces derniers pour fournir les fonctionnalités qu'on vient de citer. Le serveur LSP les lui fournit à sa demande ou non selon le contexte, en allant les chercher depuis le compilateur. Il est toutefois important de comprendre que ce serveur n'est pas un compilateur et n'a pas le même rôle que ce dernier. Le serveur partage du code au compilateur qui va le lire et nous donner des informations dessus. Il se place comme pont entre le compilateur qui va analyser et fournir des informations comme les erreurs, le type des éléments, la position de la définition d'un élément, etc, au serveur et l'IDE ou l'éditeur qui agira comme client qui demandera ces informations au serveur. Il faut aussi noter la possibilité du client (IDE ou éditeur) à envoyer des informations sans attendre de réponse du serveur. On dit là que le client envoie des notifications au serveur. Toutefois, le serveur peut entreprendre des actions à la suite de la réception de ces informations de notification.

1.1.3. Pourquoi un serveur LSP?

Pour fournir les fonctionnalités habituelles qui nous aident à écrire du code, comme la complétion de code par exemple, les outils de développement ont besoin de faire une analyse du code qu'on écrit. L'une des solutions possibles à ce problème est l'utilisation d'un serveur qui se placerait comme intermédiaire entre le compilateur et l'outil de développement, qui jouera le rôle de client, pour lui fournir des informations à sa demande (ou en recevoir selon le contexte), en allant chercher ces informations depuis le compilateur du langage. Dans le passé, étant donné que les outils de développement sont différents, pour faire ce travail, on devait développer un adaptateur pour chaque serveur d'un langage donné par éditeur ou IDE, et cet adaptateur était spécialisé. Donc, cela demandait d'implémenter beaucoup d'adaptateurs pour supporter un grand nombre d'outils, ce qui demande beaucoup de répétitions pour fournir les mêmes fonctionnalités. Mais en 2016, Microsoft a introduit

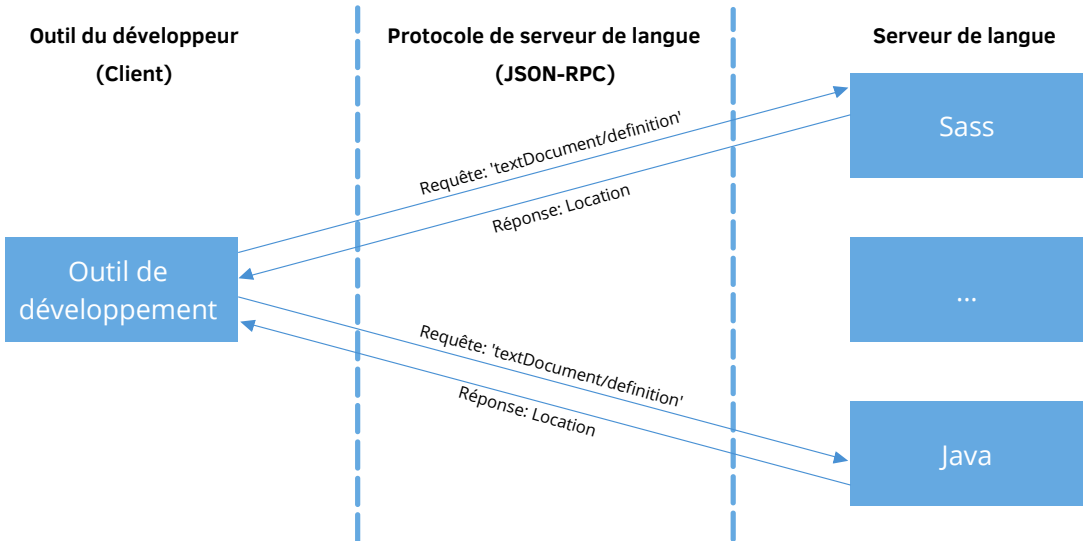

Fig. 1.1. Une vue des serveurs LSP.

le protocole LSP qui s'est très vite imposé comme un standard du fait de l'importance de l'entreprise et le nombre élevé d'utilisateurs des services Microsoft.
Les serveurs LSP peuvent être développés dans beaucoup de langages de programmation (mais on a un serveur par langage), et les clients peuvent communiquer avec plusieurs serveurs à la fois. Le client et le serveur communiquent à travers le protocole JSON-RPC[**12**] fait en JSON[**8**]. Tant que cette communication est possible entre un client et un serveur qui supportent chacun le JSON-RPC, le processus peut être mis en place, même si, par exemple, le client et le serveur ne se trouvent pas aux mêmes emplacements (le serveur peut être dans une autre machine par exemple). L'argument principal, et pas des moindres, des serveurs LSP, est qu'on implémente un seul serveur par langage qui peut être utilisé pour n'importe quel éditeur/IDE, tant que ce dernier supporte le protocole LSP. Le protocole, aussi, ne fait pas de différence entre les IDEs et les éditeurs de texte. Les uns comme les autres, tant qu'ils supportent le JSON-RPC, peuvent être les clients d'un serveur LSP. La plupart des IDEs et des éditeurs modernes populaires supportent le protocole LSP. On peut citer comme exemple **Emacs**, **Visual Studio Code**, **Vim**, **Eclipse**, **Atom**, etc.
Et coder un seul serveur une seule fois, et utiliser le même pour autant d'IDEs et éditeurs à la fois, c'est un énorme avantage parce qu'on gagne en temps et en qualité, pour les serveurs et les clients.

La Figure 1.1 illustre nos dires [1]. Comme on peut le voir, le protocole LSP ne fait pas de différence entre les outils de développement. Aussi, lorsqu'on travaille avec plusieurs langages sur le même outil de développement, il y a un serveur pour chaque langage et ils

[1]image tirée du site microsoft.github.io présentant le protocole LSP

peuvent fonctionner parallèlement. Grâce au protocole LSP, on peut utiliser facilement le même éditeur pour pleins de langages différents, ce qui était plus limité avant.

1.2. Un œil sur Typer

Le langage Typer s'inspire de beaucoup de systèmes pour en tirer le meilleur de chacun d'eux. Dans cette partie, on va voir quelques aspects du langage ainsi qu'un aperçu de la syntaxe du langage.

1.2.1. Typer en quelques mots

Typer est un langage de programmation expérimental de la famille ML [**21**] (tel qu'OCaml, SML, ou Haskell) écrit en OCaml[**28**]. Le nom Typer (la terminaison "er") est un hommage à Scheme[**1**] qui s'appelait originairement Schemer, de qui il hérite certaines idées, à savoir avoir un cœur de langage minimal et avoir la possibilité d'étendre le langage avec notamment le système de macros. D'autres langages comme Idris[**9**] (Section 2.3.1) partagent également cette philosophie.

Typer s'inspire également d'autres systèmes. Le cœur du langage est similaire à celui de Coq[**11**][**26**], mais avec un système de macro semblable à ceux de la famille Lisp[**15**][**26**]. Quant à la syntaxe, elle s'inspire de ML, ce qu'on peut voir dans le Listing 1.3 sauf pour la déclaration de macro qui ressemble plus à la syntaxe de Lisp, sauf qu'en Typer, les parenthèses ne sont là que pour corriger la précédence des opérateurs, mais n'ont pas d'autres significations. Enfin, Typer peut aussi être utilisé comme un assistant à la preuve[**14**] (un assistant à la preuve est un outil logiciel qui aide au développement de preuves formelles. On verra dans la Section 1.2.5 pourquoi Typer peut être utilisé à cette fin).

1.2.2. Petit aperçu syntaxique

Avant d'aller plus loin, voyons quelques exemples de code Typer pour nous familiariser avec la syntaxe du langage et pouvoir nous y référer par la suite. On peut associer des valeurs à des variables de la façon qui suit:

```
1 % entier
2 ent = 5 ;
3
4 % chaîne de caractère
5 str = "une chaine" ;
6
7 % flottant
8 flt = 5.4 ;
9
```

```
10 % booléen
11 bl = false ;
12
13 % liste
14 lst = cons 1 (cons 2) nil ;
```

Listing 1.1. Déclaration de variables.

Comme on peut le remarquer, le '%' est utilisé pour faire des commentaires. Quant au ';' , il est utilisé pour séparer deux déclarations. Typer a essentiellement deux éléments principaux dans son cœur: les *fonctions* et *types algébriques* (algebraic data types)[**26**]. Pour définir une fonction, on peut écrire comme suit:

```
1 multiplyTwoEl : Int -> Int -> Int ;
2 multiplyTwoEl a b = a * b ;
3
4 % Une fonction qui fait la même chose.
5 mul2El = lambda a b -> _*_ a b ;
```

Listing 1.2. Définition de fonctions.

Ces deux fonctions font la même chose. Elles prennent deux entiers et renvoient leurs multiplications. Comme on peut le voir, on peut spécifier le type des arguments de la fonction et le type de retour, comme on le voit à la ligne 1, ou tout simplement ne pas spécifier les types d'arguments comme à la ligne 5. On remarque aussi que les opérateurs peuvent s'écrire de manière infixe et préfixe, c'est-à-dire l'opérateur au milieu des opérandes ou avant. Pour définir un type algébrique, on utilise le mot-clé **type**. Pour définir le type représentant une liste par exemple, on pourrait écrire:

```
1 type List (t : Type)
2     | nil
3     | cons (hd : t) (tl : List t) ;
```

Listing 1.3. Définition du type List.

Cette déclaration définit le type **List** et ses constructeurs **nil** et **cons**. On remarque ici que parmi les types prédéfinis, il y a le type **Type** qui est le type des types. **nil** représente la liste vide et **cons** est utilisé dans le cas d'une liste non vide. **cons** prend deux arguments, un élément de la liste nommé **hd** de type **t**, la tête de la liste, et le reste de la liste nommé **tl** de type **List t**. On a pu voir un exemple de la définition d'une liste plus haut avec l'exemple du Listing1.3. Il faut noter que les noms qu'on a donnés aux arguments de **cons**, **hd** et **tl** ne sont pas obligatoires, on peut ne pas les préciser et écrire :

```
type List (t : Type)
    | nil
    | cons t (List t) ;
```

Listing 1.4. Le type List.

La fonction **map** qui transforme une liste en une autre en appliquant une fonction à chacun des éléments de la liste peut s'écrire comme suit :

```
List_map : (A : Type) ≡> (B : Type) ≡>
             (A -> B) -> List A -> List B ;
List_map f as = case as
    | nil => nil
    | cons a as => cons (f a) (List_map f as) ;
```

Listing 1.5. Fonction map.

La fonction **List_map** prend explicitement en arguments une fonction **f** qui prend en argument un **A** et renvoie un **B**, une liste **as** de **A** et renvoie une liste de **B**. Elle prend deux arguments implicites **A** de type **Type** et **B** de même type. Avec le mot **case**, on évalue les alternatives possibles de la liste **as** qui peuvent être **nil**, une liste vide, et dans ce cas, on renvoie **nil** ou **cons**, où la liste est non vide, et donc dans ce cas, on renvoie **cons** de l'application de la fonction **f** sur **a** et un appel récursif de **List_map** avec la fonction **f** et le reste de la liste. Ce qui nous renverra à la fin une liste avec chacun des éléments transformés avec la fonction **f**. On remarque aussi 3 types de flèches :

- la flèche avec 3 traits est utilisée pour les arguments implicites d'une fonction. Ils désignent les paramètres de types que l'élaboration retrouve sans que le programmeur les ait écrits. On remarque d'ailleurs dans l'appel récursif de **List_map**, on a mis que deux arguments.
- la flèche avec 2 traits est utilisée pour spécifier le côté droit des branches d'un **case**
- et la flèche avec un seul trait est utilisée pour préciser les arguments explicites d'une fonction ou pour définir une fonction anonyme avec **lambda** comme on a pu le voir dans le Listing 1.2. Les arguments explicites sont tout simplement les arguments normaux qu'on utilise avec une fonction pour l'appeler. Dans notre exemple, **f** et **as** sont des arguments explicites de la fonction **List_map**.

Comme on l'a vu plus haut, Typer est un langage à cœur minimal avec des possibilités d'extensions grâce aux macros. Pour définir une macro, on peut faire comme suit :

```
nom_macro : Macro ;
nom_macro = macro fonction ;
```

Listing 1.6. Définition d'une macro.

nom_macro est le nom de la macro qu'on veut définir, qui sera de type **Macro**. Quant à **macro**, c'est le constructeur du type **Macro**. Pour ce qui est de **fonction**, c'est la fonction qui fait l'expansion. Ci-dessous un exemple d'une définition de macro en Typer :

```
1 foo : Integer -> List Sexp -> IO Sexp ;
2 foo a _args = IO_return (Sexp_integer a) ;
3 six = macro (foo 6) ;
4 huit = 2 + (six _) ;
```

Listing 1.7. Exemple de définition d'une macro.

Quand on définit une macro en Typer, on le fait avec le constructeur *macro* qui prend en paramètre une fonction qui fait l'expansion. Cette fonction doit prendre en paramètres une liste de *sexp* (s-expressions ou expression symbolique) et renvoyer une *sexp*. On a d'abord défini la fonction **foo** qui prend un **Integer a**, une **liste de Sexp _args** et renvoie une **Sexp**. Pour notre cas, on n'utilise pas la liste prise en paramètre, c'est pour ça qu'on ajoute _ au début pour dire au compilateur qu'on n'a pas l'intention de l'utiliser. Dans notre fonction **foo**, on applique la fonction **Sexp_integer** à notre **Integer** pris en paramètre. Cette fonction prend un **Integer** et le transforme en **Sexp**. Ensuite, on utilise notre fonction **foo** ainsi définie pour définir la macro **six** en l'utilisant pour faire l'expansion. À la fin, on utilise juste la macro définie. Il faut noter que l'appel d'une macro se fait toujours avec au moins un paramètre, d'où le paramètre **'_'** qu'on donne à la macro **six** définie.

1.2.3. La philosophie de Typer

Comme le décrit l'article [**26**] qui présente Typer, il est réalisé dans le but d'avoir un langage de programmation avec un système de type aussi puissant que celui de Coq, et un système de méta-programmation comme ceux de la famille Lisp. En effet, le langage a un cœur minimaliste, avec un système de type robuste, et combine ça avec la possibilité de créer et appeler des macros, ce qui permet de manipuler et étendre le langage comme on le souhaite. La syntaxe, quant à elle, s'inspire des langages de la famille ML.

1.2.4. Fonctionnalités de Typer

Typer est un langage encore en développement. Pour l'instant, le langage a les fonctionnalités suivantes :

- Typer supporte tout d'abord la *programmation fonctionnelle*. En effet, Typer est un langage à paradigme fonctionnel.
- il prend également en charge *les types dépendants*. On peut définir des types dont la définition dépend d'une valeur dans Typer. Grâce aux types dépendants, on peut

écrire, manipuler et prouver des propositions logiques mathématiques, comme dans les assistants à la preuve tels que Coq[**11**].

- Typer permet aussi de faire du *polymorphisme paramétrique*.
- le langage prend aussi en charge les *paramètres implicites* qui désignent les paramètres de types déduits par l'élaborateur, comme on a pu en voir un exemple dans le Listing 1.5 où les types de **A** et **B** sont définis de manière implicite.
- il y a également la possibilité de définir des *macros*. Typer est un langage à cœur minimaliste et extensible avec l'utilisation des macros.

1.2.5. Système de types

Les types sont très importants en Typer, et ils ont aussi une place prépondérante dans notre serveur LSP parce qu'on les retrouve dans beaucoup de fonctionnalités qu'on veut offrir comme montrer le type d'un élément quand la souris est dessus ou encore donner le type d'une complétion dans les détails de cette dernière. Les types en Typer sont des objets de première classe, c'est-à-dire qu'ils peuvent être manipulés comme n'importe quelle valeur, stockés dans des structures de données, des variables, passés en arguments ou encore renvoyés comme valeur de retour d'une fonction. L'usage de **Type** par exemple dans le Listing 1.5 ne rend pas les arguments spéciaux. Aussi, les types en Typer correspondent à des valeurs contenues dans des variables ordinaires. Par exemple, **Int** est en fait une variable tout à fait normale qui contient initialement une valeur qui s'avère être un type représentant les entiers. La représentation officielle de cette valeur en Typer est **##Int**. Celle-ci se généralise avec les autres types comme **##Type** ou encore **##String**. Il faut aussi savoir que Typer est statiquement typé et les types sont déduits automatiquement du programme grâce à l'inférence de types de Hindley-Milner[**22**] (c'est pour ça qu'on a pu voir dans la Section 1.2.2 qu'on n'est pas obligé de spécifier le type des arguments d'une fonction). Cette méthode d'inférence de type peut retrouver le type des variables, des fonctions et des expressions pouvant être écrites sans annotation de leurs types par rapport à leurs utilisations dans le programme. On peut aussi, bien évidemment, annoter les types des fonctions avec un **':'** comme on l'a vu précédemment avec la fonction **List_map**.

1.3. Sommaire du protocole LSP

Le serveur LSP s'interpose entre le compilateur et le client (un éditeur ou un IDE). Le client lui demande des informations et le serveur les lui fournit. Le client peut également envoyer au serveur des informations sans attente de réponse. Le serveur offre des fonctionnalités comme les diagnostics (les erreurs, les avertissements, etc), la complétion de code, l'affichage d'informations au passage de la souris (*hover*), etc. On peut choisir de démarrer le serveur au moment voulu. Mais en général, on le démarre à l'ouverture d'un fichier avec

l'extension choisie. Quand le document ouvert est modifié, le client le notifie au serveur et lui envoie également le nouvel état du document et ce dernier choisit ce qu'il fait après ça. Il peut par exemple utiliser ce nouvel état du document pour envoyer de nouveaux diagnostics, de nouvelles informations à la nouvelle position de la souris, etc. À la fermeture du document, le client le notifie au serveur, et ce dernier choisit quoi faire après ça. Il peut choisir d'arrêter le serveur par exemple.

1.4. Contributions

Présentement, pour programmer en Typer, les options possibles sont le développement directement en ligne de commandes ou écrire dans un fichier et utiliser l'interpréteur de Typer pour l'exécution du fichier. On n'a donc aucune facilité d'écriture de code. Et pour une personne qui débute avec le langage par exemple, c'est assez difficile de programmer en Typer et apprendre le langage. L'objectif de notre travail est de faire en sorte de pouvoir programmer en Typer dans des outils existants en implémentant un serveur LSP pour le langage. Les contributions de ce projet sont :

(1) l'offre d'un serveur LSP au langage de programmation Typer afin de faciliter le développement en Typer avec les IDEs ou les éditeurs tels que Emacs. C'est la contribution principale de ce projet. Ce serveur fonctionne en se positionnant entre le compilateur et le client, et en utilisant les informations du compilateur qu'il fournit au client selon ce que ce dernier lui demande ou lui notifie.

(2) la révision de toutes les erreurs signalables pour les classifier correctement. En effet, on verra dans la suite de ce mémoire qu'on aura beaucoup de travaux et d'ajustements à faire pour pouvoir implémenter ce serveur parce que le langage n'est initialement pas forcément fait de façon à s'adapter au protocole LSP. Parmi ces changements, figurent les erreurs qui, pour certaines, doivent se manifester autrement dans le cadre du serveur LSP pour que l'élaboration puisse terminer et qu'on puisse ainsi afficher des messages d'erreurs utiles au programmeur.

(3) l'amélioration de la réification des types en un morceau de code équivalent. Les types ont habituellement une impression laide et affichent certaines parties non essentielles au programmeur. On travaille sur cet aspect pour améliorer l'impression des types en essayant de trouver du code équivalent pouvant améliorer l'affichage des types. Cette meilleure impression des types nécessite des recherches constantes dans le contexte de typage afin de pouvoir afficher les types de manière plus simplifiée et plus facilement compréhensible par l'utilisateur.

(4) le changement du langage cœur de Typer pour y ajouter une primitive dédiée à l'accès au champ d'un objet. En effet, implémenter certaines fonctionnalités nécessitait des changements majeurs. Pour implémenter certaines fonctionnalités, il a fallu changer

le cœur même du langage, c'est-à-dire la définition d'une lambda expression. Par exemple, le serveur se base sur l'arbre de syntaxe abstraite généré par la phase d'élaboration (Section 3.2) pour faire la plupart des fonctionnalités. Par contre, pour faire la complétion des méthodes d'un module en écrivant *List.foo* ou *List.map* par exemple, on avait soit une erreur (parce que **foo** n'existe pas parmi les méthodes du module **List**), soit pas les informations nécessaires pour offrir la fonctionnalité parce qu'ils ont été perdus pendant l'expansion des macros respectivement. On propose une solution à ce problème en étendant la définition des lambdas expressions du langage afin de préserver les informations nécessaires pour la fonctionnalité et par la même occasion gagner en vitesse en faisant une optimisation dans l'accès au champ d'un objet. On discute de ce point dans la Section 5.1.

Chapitre 2

État de l'art

Dans ce chapitre, nous allons dresser un portrait de ce qui se fait actuellement allant dans le sens de notre travail. Nous parlerons de *Tree-sitter*[**5**], un outil qui peut être utilisé pour résoudre des problèmes qui vont dans le même sens que le nôtre, de Merlin[**3**], un serveur de langue qui existait avant les serveurs LSP et d'*Idris*[**9**], un langage de programmation qui a des similarités avec Typer et qui a son propre IDE, *idris-mode*[**13**] sur Emacs.

2.1. Tree-sitter

Tree-sitter est une bibliothèque avec laquelle on peut faire des analyses syntaxiques incrémentales. L'outil est utilisé dans certains cas comme le nôtre où des arbres de syntaxe interviennent. L'outil permet d'analyser un texte et générer des arbres de syntaxe associés au texte au fur et à mesure que le texte change.

2.1.1. Une vue d'ensemble

Tree-sitter est un outil qui fait de l'analyse syntaxique et qui analyse du code de manière incrémentale. En d'autres termes, il peut construire un arbre syntaxique concret pour un fichier source donné et faire des mises à jour très rapides de l'arbre syntaxique au fur et à mesure qu'on édite le fichier source en question. La vitesse de Tree-sitter est un des arguments de la bibliothèque, elle est "instantanée" à l'échelle humaine à chaque mise à jour du code après l'analyse initiale. Ceci lui permet de fonctionner de manière synchrone contrairement à un outil comme LSP qui est asynchrone (avec LSP, une requête est envoyée et on n'attend pas la réponse d'une requête avant d'envoyer une autre requête. Les réponses aussi ne viennent pas forcément dans l'ordre d'envoi des requêtes). La bibliothèque est écrite en pur C. Tree-sitter se veut être assez général pour pouvoir être utilisé pour n'importe quel langage de programmation, rapide pour une mise à jour en temps réel de l'arbre de syntaxe pendant l'édition du code, robuste pour être capable de fournir des résultats même en cas d'éventuelles erreurs et sans dépendance pour pouvoir être intégré dans n'importe quelle

application. Pour la gestion des erreurs, l'idée est que Tree-sitter accepte toute chaîne de caractère, et si cette dernière ne correspond pas à la grammaire, l'analyseur le corrige pour qu'il corresponde à la grammaire. Et évidemment, il existe plusieurs combinaisons de corrections pour une erreur de syntaxe. Pour prendre une décision, l'analyseur suit les algorithmes des analyseurs GLR (GLR parser)[**19**] pour gérer les ambiguïtés. Quant aux analyseurs syntaxiques disponibles, ils sont assez complets et comptent la plupart des langages populaires tels que Java, Javascript, Python, PHP, C++, OCaml, etc. Tree-sitter est utilisé par des éditeurs comme *Atom* et *Neovim*.

2.1.2. Fonctionnalités

L'outil fournit les fonctionnalités suivantes :

- la coloration syntaxique en fonction du rôle qu'on donne aux mots dans le code
- le pliage de code et l'indentation automatique
- la navigation en termes de logique de code comme se déplacer au parent du nœud actuel ou encore transposer des arguments dans un appel de fonction, et plus généralement les fonctionnalités d'un éditeur de structure.

Il faut savoir que cette liste n'est pas exhaustive. On peut offrir toute fonctionnalité qu'on peut fournir en utilisant des arbres de syntaxes concrets. Par exemple, sur Github, il y a une nouvelle fonctionnalité d'une table des matières qui montre les fichiers changés, les fichiers ajoutés, les fichiers supprimés ainsi que les détails de ces changements dans les fichiers, à savoir les fonctions modifiées dans les pull request. Et pour faire cette dernière fonctionnalité, c'est-à-dire trouver les fonctions modifiées et les modifications dans le texte, ils ont dû faire des analyses et générer des arbres de syntaxe, et pour cela, ils ont utilisé Tree-sitter.

2.1.3. Spécificités et limites

Tree-sitter est inspiré de certains IDEs comme Eclipse ou encore Intellij. Ce que fait un IDE, c'est analyser un code et produire un arbre de syntaxe qui servira de base pour toutes les fonctionnalités présentes dans un IDE. La limite des IDEs est qu'ils sont mieux adaptés pour un langage spécifique et qu'ils sont lents par rapport à Tree-sitter. Les serveurs LSP, quant à eux, fonctionnent dans le principe où on utilise l'éditeur pour écrire le code et un serveur séparé pour l'analyser afin de fournir des fonctionnalités comme la complétion, aller à la définition, etc et toutes les facilités habituelles qu'on utilise pour programmer. Mais, comme pour les IDEs, à chaque mise à jour du code, de manière générale, tout le code est à nouveau analysé, et de plus, il transite à travers le JSON-RPC, ce qui rend les serveurs LSP un peu plus lents que Tree-sitter. Avec Tree-sitter également, les informations de l'arbre précédent sont préservées et on ne tient compte que des changements, cela amène à plus de rapidité. Par contre, Tree-sitter a une compréhension à l'échelle d'un fichier indépendant

(il ne fait que transformer des fichiers en arbres de syntaxes correspondants), mais n'a pas une logique de compréhension d'un projet complet comme peut le faire LSP. Donc il est mieux adapté pour certaines fonctionnalités et moins pour d'autres. Celles qui ont besoin de beaucoup de rapidité, comme la coloration syntaxique ou l'indentation automatique sont plus effectives avec Tree-sitter qu'avec LSP, et celles qui demandent un peu plus d'analyses comme les diagnostics sont mieux avec LSP. D'ailleurs, certains combinent les deux pour tirer le meilleur de chacun des deux outils.

2.2. Merlin

Merlin[**3**] est un outil qui fournit les fonctionnalités des éditeurs modernes pour le langage de programmation OCaml. Implémenté principalement par Frédéric Bour et Thomas Refis, sous le financement de l'entreprise Jane Street, le projet a vu le jour pour aider à programmer en OCaml.

2.2.1. Merlin brièvement

Merlin est un serveur de langue pour OCaml qui a commencé à être développé en 2013[**3**]. Merlin peut également être utilisé comme une bibliothèque. Il est fait pour fournir de l'aide aux programmeurs OCaml à écrire leur code. L'outil est disponible sur Opam[**18**], le gestionnaire de paquet du langage OCaml. L'outil peut être connecté à un éditeur pour fournir des fonctionnalités comme un retour sur les erreurs et avertissements, la complétion de code, le saut à la définition, etc, de manière incrémentale pendant qu'on édite du code OCaml. Pour supporter l'incrémentalité et gérer les erreurs, Merlin utilise l'analyseur lexical Ocamllex[**25**] et l'analyseur (parser) pour le langage OCaml Menhir[**20**]. Merlin ne prend en charge que deux éditeurs prêts à l'emploi dès l'installation : Vim et Emacs. Toutefois, il peut être utilisé pour certains autres éditeurs comme Atom et Sublime Text avec des configurations supplémentaires.

2.2.2. Fonctionnement

Merlin est un serveur de langue. Étant donné que l'outil a été développé en 2013 et qu'en 2013, le standard des serveurs LSP n'existait pas encore, Merlin a été implémenté en utilisant son propre protocole. Pour implémenter un serveur de langue, on a besoin de communiquer avec une interface de langage raisonnablement incrémentielle pour calculer des arbres typés et un éditeur pour échanger des informations avec le serveur à travers des requêtes qui vont demander des informations présentes dans l'arbre typé. Merlin ne fait pas exception et fonctionne ainsi. L'outil n'a pas été construit en implémentant un nouveau langage frontal, mais en modifiant des outils qui existaient déjà, mais qui n'étaient pas incrémentiels pour supporter l'incrémentalité.

2.2.3. Fonctionnalités

Merlin offre la plupart des fonctionnalités des éditeurs/IDE modernes. Mais, étant donné que Merlin utilise son propre protocole, il est supporté par moins de clients qu'un standard comme LSP qui est apparu après. Il offre entre autres :

Les diagnostics: Merlin peut détecter les erreurs de syntaxe, les erreurs de type (multiplier un nombre par une chaîne de caractère par exemple) et les avertissements (comme lorsqu'on écrit une correspondance de modèle et qu'on ne couvre pas toutes les possibilités du modèle)

Le saut à la définition: Merlin peut aussi fournir l'information de la position où est défini un identificateur à l'éditeur.

type de code sous le curseur: l'outil peut trouver le type des éléments à la position du curseur. Les développeurs de Merlin décrivent cette fonctionnalité comme "étonnamment délicate" et ce pour plusieurs raisons. D'abord, parce que l'élément sur lequel le curseur pointe n'est pas nécessairement clair. Si on écrit *List.length*, l'utilisateur pourrait vouloir le type de la fonction *length* ou du module *List*. Merlin donne le type du module si l'utilisateur place le curseur avant le point. Ceci signifie que le module et la fonction doivent avoir chacun leur position précise. Ensuite, si une réécriture de terme se fait, le même identifiant peut avoir le nom d'un autre identifiant, le nom d'un type ou encore le nom d'une étiquette dans un record. Donc pour rechercher le type d'un élément, Merlin utilise le contexte local pour éliminer les ambiguïtés. Dans le développement du serveur LSP de Typer, on a rencontré des problèmes aussi dans le développement de cette fonctionnalité dans la recherche du bon nœud dans l'arbre de syntaxe abstraite. Ce problème est lié à l'expansion des macros du langage et est abordé dans la Section 5.2.

La complétion: l'outil peut suggérer des possibilités de complétion pendant l'édition dans l'éditeur/IDE. La complétion a été également affectée par la réécriture de terme possible du langage OCaml qui affecte la précédente fonctionnalité. En implémentant la complétion pour notre serveur LSP, on a aussi rencontré des problèmes, mais d'ordre différent, liés à la perte, après l'élaboration, de données essentielles à l'implémentation de la fonctionnalité. Ce sujet est discuté dans la Section 5.1.

Ces fonctionnalités ne sont pas exhaustives. Les développeurs de Merlin ont espéré que leur retour d'expérience serve aux futurs développeurs de serveur de langue en réutilisant leurs choix de conception et/ou en se servant des leçons apprises lors des difficultés qu'ils ont rencontrées pour implémenter certaines fonctionnalités.

2.3. Idris, un langage aux desseins proches de Typer

Idris[**9**] est un langage de programmation purement fonctionnel. Idris est un point de référence pour notre projet pour plusieurs raisons : d'abord parce que le langage a été fait dans un contexte similaire à celui de Typer, c'est-à-dire avec l'idée de croiser des systèmes pour réunir les points forts de chacun; ensuite parce que le langage possède également son propre IDE sur Emacs, idris-mode[**13**] en utilisant un protocole alternatif à LSP spécialement développé pour Idris.

2.3.1. Qu'est-ce qu'Idris

Idris[**9**] est un langage de programmation fonctionnel pur conçu pour être un langage à usage général. Idris a été commencé en 2007 par Edwin Brady, et tire son nom d'un dragon chanteur du programme télévisé britannique pour enfants intitulé *Ivor the Engine.* Idris est conçu pour encourager le développement basé sur les types, une approche de programmation qui considère les types comme la base du code. Avec cette approche, on peut définir des spécifications dès le début et écrire du code potentiellement plus facile à maintenir, à tester et à étendre[**10**]. Donc l'IDE **idris-mode** fournit beaucoup d'aide par rapport à ça, il aide à connaître les types des différents éléments ou encore à compléter des parties du code en se basant sur les informations de typage. C'est un aspect important pour notre travail parce que, Typer, par son design, met beaucoup d'accent sur les types. Donc, il est important pour notre travail d'offrir de l'aide sur les types des éléments.

2.3.2. Idéologies similaires à Typer

Tout comme Typer, Idris est un croisement entre plusieurs systèmes informatiques. Le système de types d'Idris est similaire à Agda[**4**], et ses preuves similaires à Coq. Il se différencie principalement de Coq et Agda par sa gestion des effets de bords. Tout comme Typer, Idris peut être utilisé comme assistant à la preuve, à la différence que ce dernier est un vérificateur total contrairement à Typer actuellement (la vérification de la totalité dans Typer n'est actuellement pas implémenté). En effet, pour que le langage Typer puisse être utilisé pour écrire et manipuler des preuves, il doit garantir de toujours terminer, et le langage ne l'assure pas. C'est pour cela qu'on dit que le langage n'inclut pas une vérification de la totalité dans son implémentation.

2.3.3. Idris et interactions

La partie interactive d'Idris mérite un coup d'œil étant donné que notre projet a pour but d'offrir à Typer justement des facilités d'écriture aussi, mais avec une autre technologie qui est un serveur LSP. Pour offrir des possibilités interactives à *Idris*, **Hannes Mehnert**

et **David Christiansen** ont développé en 2014 un IDE dans Emacs pour Idris qu'il ont appelé `idris-mode`[**13**]. Pour ce faire, ils ont implémenté une extension du compilateur d'Idris nommé *ideslave*[**13**] qui fournit un protocole grâce auquel ils ont fourni les fonctionnalités de `idris-mode`[**13**] dans Emacs. Ce protocole est une alternative à LSP développé spécialement pour Idris. Bien qu'il soit développé pour Idris, ce protocole est aussi suffisamment général pour prendre en charge une variété d'autres outils et modèles d'interaction[**13**]. Les requêtes fournies par le protocole d'*ideslave* incluent des requêtes sur l'état du compilateur. Grâce à ça, ils peuvent demander par exemple le type d'un identifiant, la documentation associée à un nom, etc. `idris-mode` fournit aussi la coloration sémantique qui aide les utilisateurs à distinguer les constantes, constructeurs de type, constructeurs de données et les variables liées[**13**]. Il permet aussi de faire le saut à partir d'un message d'erreur à sa source. `Idris-mode` gère aussi la gestion des trous (parties du programme qui ne sont pas encore écrites). C'est-à-dire qu'il peut vérifier le contenu d'un trou, tenter de remplir automatiquement un trou et afficher le type. Par exemple, considérons le code suivant:

```
1 main: IO ()
2 main  = putStrLn ?greeting
```

Listing 2.1. Idris program.

Dans le Listing 2.1, la syntaxe *?greeting* introduit un nouveau trou. Et si on vérifie le type de *greeting*, on aura comme résultat **greeting : String**, ce qui nous dit que *greeting* est de type chaîne de caractère. En Typer, les trous existent aussi avec la même syntaxe et notre serveur LSP peut fournir le type d'un trou comme *idris-mode* le fait avec *Idris*. L'IDE prend également en charge la compilation et l'exécution de programmes. Bien qu'*idris-mode* offre des fonctionnalités qui aident déjà beaucoup à la programmation en Idris sur Emacs, il n'offre pas encore des fonctionnalités classiques comme le surlignement des erreurs ou l'allée à la définition d'une fonction. L'analyse incrémentale (pendant l'écriture du code) n'est pas non plus fournie.

Chapitre 3

LSP et Typer

Dans ce chapitre, nous présentons le protocole LSP un peu plus en détails, ainsi que la phase d'élaboration de Typer étant donné que les fonctionnalités de notre serveur dépendent de cette phase.

3.1. Le protocole LSP

Le protocole LSP est un protocole utilisé entre un éditeur/IDE qui joue le rôle de client, et un serveur de langue qui fournit des informations qu'il prend d'une source, comme le compilateur d'un langage par exemple, pour les distribuer au client, à sa demande ou non. Quand le client envoie des informations au serveur, et demande en retour une réponse, on parle dans ce cas de figure d'un envoi de requête au serveur. Si le client ne demande pas de réponse, on dit qu'il a envoyé des notifications au serveur. Il existe également des notifications du serveur au client. Quand le client envoie des notifications au serveur, celui-ci peut décider de faire des actions à la suite de ces derniers. Les informations que le serveur envoie sont utilisées pour fournir des fonctionnalités au client, comme la complétion de code, la documentation au survol, le surlignement des erreurs, etc. Le serveur et le client peuvent communiquer de différentes façons :

stdio: en utilisant *stdio* comme canal de communication.

pipe: en se servant des *pipes* pour windows ou des *fichiers socket* pour Linux et Mac OS comme canal de communication.

socket: également, en recourant à un *socket* comme canal de communication.

node-ipc: et pour finir, en établissant une communication *IPC* comme une connection TCP.

Le serveur peut être écrit dans tout langage de programmation souhaité, puisque la communication entre le serveur et le client se fait par des échanges de messages textes au format JSON, tant qu'il peut communiquer d'une manière ou d'une autre avec le compilateur (ou la source où le serveur va aller chercher les informations). Le client, quant à lui, est écrit

dans la langue spécifique à l'outil de développement utilisé. Plus concrètement, tous les échanges entre le serveur et le client transitent par un protocole appelé JSON-RPC[**12**].

Le protocole donne beaucoup de liberté aux concepteurs de serveurs de langues ainsi qu'aux clients. C'est effectivement nécessaire, car certains langages ne peuvent pas supporter certaines fonctionnalités, tout comme certains éditeurs ne supportent pas forcément toutes les fonctionnalités qu'offre le protocole LSP. Les éditeurs, comme les concepteurs de serveur de langue, sont libres de déterminer les fonctionnalités qu'ils vont supporter. Également, le protocole laisse la liberté aux serveurs de langue de travailler avec un système de projet ou un système à fichiers uniques. Cela dépend totalement du langage pour lequel on conçoit le serveur. Notre serveur, par exemple, travaille seulement avec des fichiers uniques parce que Typer ne supporte pas une notion de projet actuellement. Le langage Typer a une notion de compilation séparée, mais très limitée pour l'instant et utilisée uniquement en interne pour les fichiers initiaux de Typer. Aussi, il n'y a pas de notion de taille limite définie par le protocole, cette notion est totalement gérée par le langage pour lequel on implémente le serveur. Il faut toutefois noter que les clients aussi peuvent avoir des tailles limites de support. À titre d'exemple, *Vs Code* a comme limite un fichier de *20MB*, soit *300 000* lignes de code.

Étant donné que le protocole fonctionne de manière asynchrone, des inconsistances d'état entre le client et le serveur peuvent subvenir. C'est-à-dire qu'après avoir reçu la réponse d'une requête du serveur par exemple, l'état du document peut avoir changé entre temps. Cet aspect est géré par le client. Ce dernier va déterminer si l'état a changé ou pas et le notifier au serveur. Le serveur ne gère pas ça et se contente d'envoyer des informations au client par rapport aux requêtes et aux notifications que le client lui envoie. De manière générale, quand les clients reçoivent la réponse à une requête, mais que l'état a changé entre temps, ce dernier ne l'utilise pas et envoie tout simplement une autre requête pour avoir des informations à jour. C'est aussi pour minimiser ce genre de problèmes que le serveur a besoin d'être rapide dans la réponse aux requêtes et l'envoi d'informations au client.

3.1.1. Le JSON-RPC

Le *JSON-RPC (JavaScript Object Notation Remote Procedure Call)*[**12**] est un protocole codé en JSON[**8**] d'appel de procédure à distance qui a vu le jour pour la première fois en 2005 avec la version *1.0*. Le protocole permet l'envoie de notifications et de requêtes à un serveur. Les requêtes peuvent être répondues de manière asynchrone. Le protocole LSP utilise toujours la version *2.0* du protocole qui a vu le jour en 2009, puis révisé en 2010. Les requêtes sont en fait des appels de méthodes spécifiques fournies par un système distant. C'est d'ailleurs pour cette raison que, dans le cadre du protocole LSP, les serveurs et les clients ne sont pas obligés d'être dans le même espace mémoire (Le serveur peut par exemple

```
1
2 --> {"jsonrpc": "2.0", "method": "multiply", "params": [4, 3], "id": 1}
3 <-- {"jsonrpc": "2.0", "result": 12, "id": 1}
4
5 --> {"jsonrpc": "2.0", "method": "multiply", "params": {"x": 5, "y": 4}, "id": 3}
6 <-- {"jsonrpc": "2.0", "result": 20, "id": 3}
7
```

Fig. 3.1. Une requête JSON-RPC.

```
1
2 --> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
3
```

Fig. 3.2. Une notification JSON-RPC.

être dans une machine à part comme on l'a dit dans la Section 1.1.3). Une requête est un objet JSON qui peut contenir trois membres :

- id : une chaîne (ou un nombre) utilisé pour des raisons de correspondance entre la demande et la réponse qu'elle reçoit. Il faut noter donc que l'id peut être omis lorsqu'on n'attend pas de réponse de la part du serveur (dans le cas d'une notification).
- method : une chaîne de caractère qui est le nom de la méthode qu'on appelle.
- params : un objet ou une liste contenant des valeurs qu'il faut passer à la méthode qu'on appelle dans la requête. Dans le cas où elle ne prend pas de paramètres, ce membre peut ne pas être fourni.

La réponse à une requête est également un objet JSON qui peut contenir les membres suivants :

- id : représente l'identifiant de la requête à laquelle elle répond.
- result : ce sont les résultats des données renvoyées par la méthode appelée. Lorsqu'une erreur a lieu, ce membre n'existe pas.
- error : ce membre existe seulement lorsqu'on a une erreur et contient le message d'erreur.

Il y a un autre membre possible dans le cas d'une requête, le membre **jsonrpc** qui contient la version du protocole utilisé. Dans le cadre du protocole LSP, c'est toujours la valeur **2.0**. On peut voir un exemple d'une requête JSON-RPC à la Figure 3.1.

Comme on peut le voir, la requête a demandé d'appeler la méthode *multiply* avec les paramètres *4* et *3*, et elle a reçu *12* dans le premier échange. Dans le second échange, la requête appelle la méthode *multiply* avec des paramètres nommés x et y pour le résultat de 20. On remarque qu'on a deux types de paramètres possibles. Ils sont soit sous forme d'un objet ou d'une liste comme on l'a dit précédemment. Une notification se présente comme à la Figure 3.2.

La notification ci-dessous demande au serveur d'appeler la fonction *update* avec les paramètres *1,2,3,4,5*.

```
Content-Length: ...\r\n
\r\n
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "textDocument/didOpen",
    "params": {
        ...
    }
}
```

Fig. 3.3. Une requête d'un serveur LSP.

3.1.2. Comment se présentent les requêtes des serveurs LSP?

Les requêtes des serveurs LSP sont un tout petit peu différentes des requêtes JSON-RPC. Elle se compose d'un en-tête et d'un contenu séparés par un '\r\n'. L'en-tête est composé par les membres suivants :

- Content-Length : la longueur de la partie contenue en octet. Cet en-tête est obligatoire et est un nombre.
- Content-Type : le type de la partie contenue. Le type par défaut est **application/vscode-jsonrpc;charset=utf-8**

Quant au contenu, il suit le protocole JSON-RPC, donc il est exactement comme on l'a décrit précédemment. À la Figure 3.3, on peut observer une illustration de la présentation d'une requête d'un serveur LSP.

La Figure 3.4 montre quant à elle une requête concrète de notre serveur sur Emacs et sa réponse.

On remarque que l'affichage n'est pas la même. En fait, c'est juste la manière dont elle est présentée par l'éditeur au niveau de l'affichage. On a envoyé une requête **textDocument/hover** avec des paramètres **textDocument** qui contient l'**uri** du document, et le paramètre **position** qui indique la position du curseur, la ligne et la colonne. On a comme réponse un *range* qui nous indique l'intervalle de début et de fin de positionnement de l'élément sur lequel le serveur envoie une réponse (ligne de début et ligne de fin, caractère de début et caractère de fin de l'élément en question), et un **contents** qui est en fait, une chaîne de caractère contenant l'information de type retrouvée à cette position. Dans notre cas précis, c'est un entier. L'**id** de la requête envoyée est *72*.

3.1.3. Fonctionnement

Le protocole LSP met en jeu 3 choses importantes : le **client**, qui est un éditeur ou un IDE, le **JSON-RPC** qui est le moyen de communication, et le **serveur de langue** qui

```
[Trace - 09:40:10 ] Sending request 'textDocument/hover - (72)'.
Params: {
  "textDocument": {
    "uri": "file:///home/soilih/Bureau/exemple2.typer"
  },
  "position": {
    "line": 8,
    "character": 1
  }
}


[Trace - 09:40:10 ] Received response 'textDocument/hover - (72)' in 123ms.
Result: {
  "range": {
    "end": {
      "character": 4,
      "line": 9
    },
    "start": {
      "character": 0,
      "line": 9
    }
  },
  "contents": "Int"
}

```

Fig. 3.4. Une requête LSP de notre serveur sur Emacs.

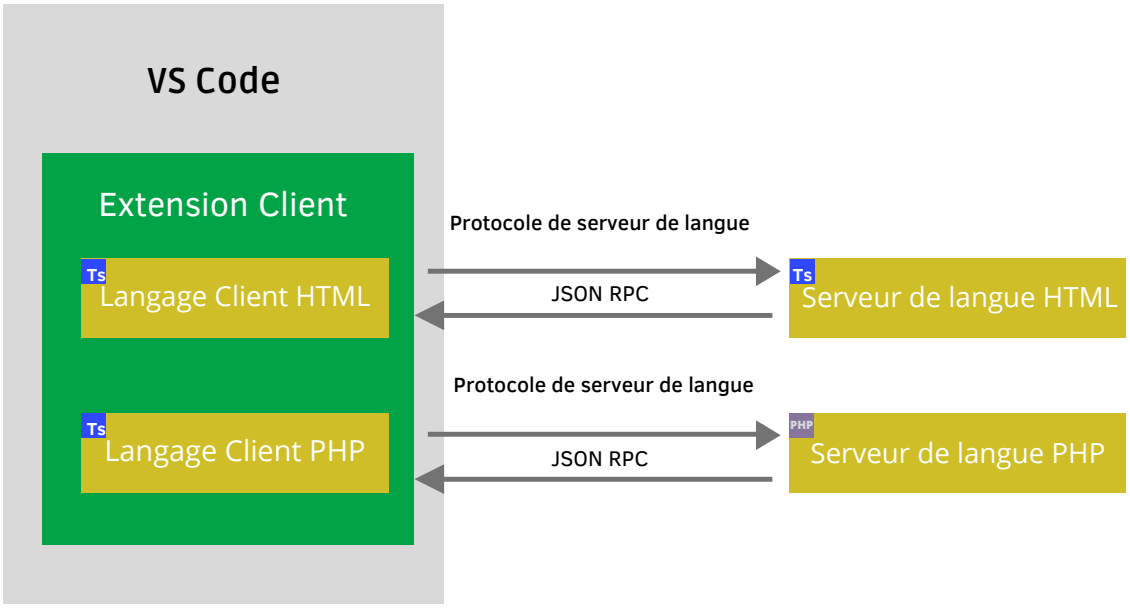

Fig. 3.5. Le protocole LSP.

s'occupe d'envoyer les informations que le client demande au serveur en s'appuyant sur les informations qu'il possède. Le serveur peut par exemple trouver ces informations en utilisant un compilateur. Grâce au contenu d'un document que le client envoie au serveur à travers une notification de modification d'un document par exemple, ce dernier va pouvoir compiler le document et générer des informations sur le document qu'il va pouvoir communiquer

au client selon ce que ce dernier lui demande ou lui notifie. Ces informations sont par exemple les *diagnostics* du document (les informations d'erreurs, les avertissements, etc) et l'arbre de syntaxe abstraite généré par l'élaboration qu'il va pouvoir utiliser pour fournir des informations nécessaires pour offrir des fonctionnalités telles que la complétion de code. Observons la Figure 3.5 pour illustrer nos dires[1]. La Figure 3.5 montre une vue d'ensemble du fonctionnement du protocole. Dans notre schéma, on a 3 axes principaux:

l'hôte: l'éditeur de texte ou IDE joue ce rôle dans ce schéma. Dans notre cas, l'hôte (client) est VS Code.

le protocole JSON-RPC: le protocole par lequel toutes les communications entre le serveur et le client se font.

le serveur de langue: c'est le serveur qui s'occupe d'aller chercher les informations de l'arbre de syntaxe généré par la compilation (la phase d'élaboration pour notre cas) pour les communiquer au client en fonction de ce qu'il demande. Notons toutefois que le serveur LSP n'est pas obligé de communiquer avec un compilateur, mais peut le faire avec n'importe quelle source détenant les informations nécessaires au client.

On remarque que l'éditeur ou l'IDE peut jouer le rôle de client de plusieurs serveurs LSP à la fois. Il faut noter aussi qu'il faudra écrire quelques configurations pour l'éditeur/IDE afin qu'il puisse communiquer avec le serveur. C'est dans ces configurations qu'on programmera par exemple que le serveur doit se lancer si on ouvre un fichier avec une extension spécifique. On peut observer aussi que les deux configurations du client, quant à elles, sont écrites en TypeScript[**2**]. Effectivement, chaque éditeur a un nombre de langages limités dans lequel on peut écrire les configurations du client pour un serveur de langue. VS Code supporte seulement des configurations clients écrites en TypeScript ou en JavaScript[**6**].

3.1.4. Déroulement des requêtes

D'un point de vue concret, les communications entre le client et le serveur sont fait avec plein de requêtes et de réponses à la suite. Le client n'est pas obligé d'attendre la réponse à une requête pour en envoyer une autre. Il peut en enchaîner à la suite et le serveur ne lui répond pas forcément par ordre d'arrivée, mais dès que l'information est prête. C'est pour cette raison qu'on dit que les serveurs LSP fonctionnent de manière asynchrone. On peut observer dans la Figure 3.6[2] un flux d'échanges entre le serveur et le client pour illustrer un schéma possible d'échanges.

La Figure 3.6 présente un exemple de déroulement des requêtes. On va faire abstraction du protocole JSON-RPC au milieu par lequel transitent tout échange entre le serveur de langue et le client.

[1]image tirée du site code.visualstudio.com en présentant le protocole LSP
[2]image tirée du site microsoft.github.io présentant le protocole LSP

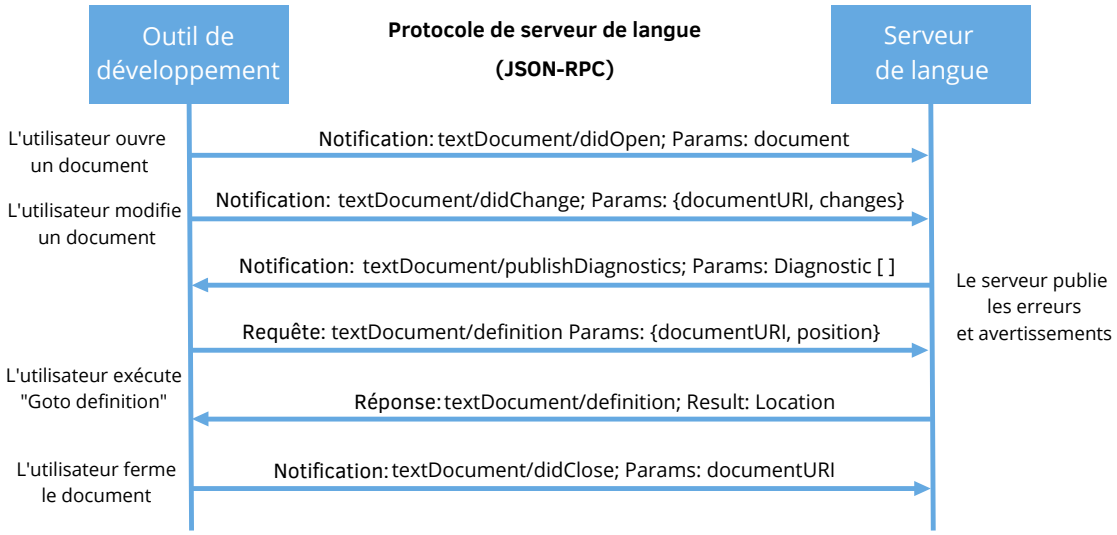

Fig. 3.6. Déroulement des requêtes.

Le client ouvre un document: le serveur se lance et la requête d'initialisation qui établit la connexion entre le serveur et le client s'effectue. Le client notifie ensuite au serveur qu'un document est ouvert et lui envoie tout le contenu du document.

Le document est modifié: quand l'utilisateur modifie le document, le client notifie au serveur qu'un document a été modifié et lui envoie le nouvel état du document. Celui-ci correspondant au précédant état (le document à l'ouverture) modifié, mais pas tout le document de nouveau. Le serveur prend note de cette information et envoie au client les nouveaux diagnostics (les erreurs, les warnings, les informations, etc). Notons que les diagnostics sont envoyés une première fois à l'ouverture du document.

Le client exécute une requête: il peut s'agir d'une requête quelconque. Dans notre cas, le client demande un saut à la définition d'un élément dans le document. Le serveur lui répond en lui envoyant les informations de positionnement de l'élément qui définit celui sur lequel le curseur se trouve. Ces informations sont la ligne de début, la ligne de fin, la colonne de début et la colonne de fin qui localisent l'élément qui définit l'élément sur lequel le curseur était quand il a envoyé la requête.

Le client ferme le document: une fois le document fermé, le client notifie au serveur qu'il a fermé le document. Le serveur peut donc décider des actions à effectuer comme s'arrêter par exemple.

3.1.5. Fonctionnalités

D'un point de vue fonctionnalités, les serveurs LSP sont assez complets et offrent toutes les fonctionnalités de facilité d'écriture de code habituelles. Le protocole offre plus de 40 fonctionnalités, et de plus, il est extensible. Le protocole propose les fonctionnalités suivantes:

Initialize: c'est la requête envoyée par le client pour demander au serveur de démarrer. Elle est suivie de la notification **Initialized** quand le client reçoit une réponse du serveur.

Shutdown: C'est une requête envoyée par le client pour demander au serveur de s'éteindre.

DidOpenTextDocument: c'est une notification que le client envoie au serveur pour lui notifier qu'un document est ouvert. Une notification d'ouverture ne peut pas être envoyée plus d'une fois sans qu'une notification de fermeture n'ait suivi avant. En d'autres termes, les notifications d'ouverture et de fermeture doivent être équilibrées. Donc un même document ne peut être ouvert plus d'une fois, à moins d'être fermé avant.

DidChangeTextDocument: cette notification est envoyée du client au serveur pour signaler les modifications apportées à un document texte. Dans le cas de notre serveur, on recompile à chaque fois que l'éditeur nous signale que l'état du document a changé. Ce changement dépend d'un éditeur à l'autre. Par exemple, certains éditeurs vont considérer que l'état a changé quand on modifie et enregistre le document, d'autres quand le texte est modifié, d'autres si les informations reçues dans l'échange précédent ne sont plus à jour, etc. Donc c'est l'éditeur qui décide quand est-ce qu'un document change d'état, et s'il nous le notifie, on recompile le document avec le serveur pour générer de nouvelles informations, à savoir un nouvel arbre de syntaxe et de nouveaux diagnostics.

DidCloseTextDocument: cette notification fait savoir au serveur qu'un document a été fermé.

PublishDiagnostics: ce sont les notifications envoyées du serveur au client pour signaler les diagnostics d'un document, c'est-à-dire les erreurs, les avertissements, les informations, etc. Les diagnostics sont détenus par le serveur, et c'est ce dernier qui décide quand est-ce qu'il les transmet au serveur. Le serveur peut choisir par exemple d'envoyer de nouveaux diagnostics à chaque modification du document, comme c'est le cas pour notre serveur LSP pour Typer.

Completion: le client appelle une requête de complétion pour trouver les éléments de complétion à une position donnée, et le serveur les lui fournit. Pour notre cas, on a rajouté aussi le type d'un élément de chaque complétion de sorte qu'on voit aussi

le type de l'item qu'on choisit pour la complétion dans la réponse qu'on fournit au client.

hover: c'est une requête que le client utilise pour demander des informations sur un texte à une position donné de la souris. Le serveur peut retourner les informations qu'il souhaite. Dans la plupart des cas, le serveur retourne le type de l'élément à la position de la souris, parce que c'est plus pertinent et pratique pour les programmeurs pour mieux écrire leur code, et pour rectifier des erreurs que si on envoyait à la place, des informations comme le fichier dans lequel l'élément a été initialement défini par exemple. On peut aussi associer une documentation en plus de cette information de type dans les détails de la réponse communiquée au client. Pour notre cas, on envoie comme information le type d'un élément parce qu'on n'a pas vraiment à disposition la documentation associée à un élément à l'état actuel du code Typer.

Go to Definition: cette requête est envoyée du client au serveur pour demander l'emplacement de la définition d'un élément à une position donnée dans le document. Le serveur peut alors lui renvoyer cet emplacement en se basant sur les informations qu'il possède. Dans notre cas, le serveur la trouve grâce à l'arbre de syntaxe généré par l'élaboration de ce même document avec Typer.

Document Highlights: la demande de surbrillance (*highlight*) est une requête envoyée du client au serveur pour trouver les *highlights* du document à une position donnée. Dans l'usage pour les langages de programmation, une surbrillance est faite à toutes les références de l'élément sur lequel on est positionné dans tout le document. Grâce aux *highlights*, on peut voir où une certaine variable est utilisée dans le texte.

Document References: le client envoie cette requête au serveur pour trouver toutes les références à un élément dans le document à une position donnée. La différence avec les *highlights* est que les highlights sont demandés pour un seul document alors que, pour les références, c'est à l'échelle du projet. Dans les éditeurs/IDE en général, à la demande de cette requête, les références sont listées et on peut passer d'une référence à une autre en cliquant sur l'élément de la liste qui nous intéresse sans souci d'ordre.

Document Symbols: le client envoie cette requête au serveur pour lui demander la liste des symboles présents dans un document. Le serveur peut répondre soit par :

- une liste des symboles présents dans le document.
- une hiérarchie des symboles trouvés dans un document.

La spécification des serveurs LSP recommande aux serveurs de renvoyer la deuxième option, car c'est la structure de donnée la plus riche. Grâce à cette hiérarchie, les serveurs LSP offre une fonctionnalité habituellement appelée *breadcrumbs*. Les *breadcrumbs* désignent le chemin emprunté pour se retrouver à l'endroit où on se situe. Si, par exemple, on est dans un document nommé *code.typer*, dans une fonction nommée

afficher, sur une variable nommée *msg*, le *breadcrumb* qu'on pourrait avoir dans ce cas de figure peut être **"code.typer > afficher > msg"**. Notons toutefois que tous les serveurs ne vont pas dans autant de détails, certains se contentent par exemple de juste nommer la fonction, ce qui nous donnerait **"code.typer > afficher"** comme *breadcrumb*. C'est au serveur de décider de comment il organise la fonctionnalité.

Évidemment, cette liste n'est pas exhaustive, les serveurs LSP peuvent offrir beaucoup plus de fonctionnalités. Aussi, le protocole est extensible, ce qui signifie qu'il y a toujours des possibilités d'offrir plus de fonctionnalités que toutes celles offertes actuellement. Il y a toutefois une chose importante à savoir, les serveurs LSP ne sont pas obligés d'offrir toutes les fonctionnalités possibles pour les serveurs LSP. On peut très bien faire un serveur qui supporte seulement les *diagnostics* et le *hover* et il fonctionnera. Par contre, la requête d'initialisation est obligatoire. Les notifications d'ouverture, modification et fermeture d'un document sont aussi généralement toujours présentes et constituent le serveur minimal. Aussi, c'est dans la requête d'initialisation qu'on spécifie les fonctionnalités du serveur. Par exemple, c'est dans cette dernière qu'on spécifiera que notre serveur supporte le *hover*. Et si on ne fournit pas cette fonctionnalité, des erreurs surviendront. A contrario, si on implémente une fonctionnalité et qu'on ne spécifie pas que le serveur la prend en charge dans la requête d'initialisation, cette dernière ne fonctionnera pas dans le client, comme si elle était inexistante.

3.2. La phase d'élaboration de Typer

Dans cette section, nous allons présenter la phase d'élaboration de Typer étant donné l'importance de cette dernière pour notre serveur. La phase d'élaboration de Typer est très importante pour notre projet parce que tout notre travail repose entièrement sur les résultats de cette dernière. Dans la globalité, on élabore un code, et on utilise les diagnostics et l'arbre de syntaxe abstraite résultants de l'élaboration pour fournir toutes les fonctionnalités de notre serveur comme le surlignement des erreurs, la complétion, le saut à la définition, etc avec notre serveur LSP. À chaque modification du code source, l'élaboration est faite de nouveau, tout le processus recommence et le serveur a de nouvelles informations qu'il peut communiquer selon ce que demande le client.

3.2.1. La phase d'élaboration

La phase d'élaboration s'occupe principalement de 3 choses[**26**] :

- elle finit l'analyse syntaxique suite à quoi on peut reconnaître l'appel d'une *fonction*, d'une *macro*, et les autres constructions du langage comme les déclarations dans des scopes locales avec un *let* ou une analyse de correspondance avec un *case*.
- elle fait l'inférence et la vérification des types.

- elle effectue également l'expansion des macros.

A la fin de la phase d'élaboration, tous les **sexp** (s-expressions ou expression symbolique) sont transformés en **lexp** typés à chaque sous-expression. En réalité, cette phase est une des étapes parmi lesquelles passe un code Typer, du code écrit à l'évaluation par l'interpréteur[**7**]. Un code Typer traverse les étapes suivantes :

(1) **File à Pretoken :** dans cette étape, les espaces blancs sont supprimés.
(2) **Pretoken à Sexp :** ici, les *Pretoken* sont transformés en arbre de syntaxe, autrement dit, en expressions symboliques (sexp).
(3) **Sexp à Lexp :** c'est la fameuse phase d'élaboration où on transforme les expressions symboliques (sexp) en lambdas expressions (lexp).
(4) **Lexp à Elexp :** ici, toute information dont l'évaluation n'a pas besoin est effacée. D'ailleurs *elexp* signifie *erased lexp*.
(5) **Elexp à Vexp :** dans cette dernière étape, l'évaluation du code est effectuée.

Comme mentionné plus haut, l'élaboration s'occupe de l'inférence et la vérification des types ainsi que de l'expansion des macros. Notre travail commence à partir de la fin de cette phase en y utilisant ses résultats, à savoir l'arbre de syntaxe abstraite généré par cette phase et les diagnostics également. Étant donné qu'on ne veut pas exécuter du code avec notre serveur, la partie *Elexp* à *Vexp* est généralement omise dans le cadre de notre travail. La partie principale qu'on utilise est la phase d'élaboration, donc le passage des *Sexp* à *Lexp*. Ensuite, ce sont les résultats de cette phase qu'on utilise pour notre serveur. Pendant ce passage a lieu aussi l'expansion des macros. En effet, l'inférence et la vérification des types font l'élaboration. Et c'est aussi pendant cette inférence et cette vérification qu'a lieu l'expansion des macros.

3.2.2. Inférence et vérification des types

L'élaboration se fait avec deux fonctions, la fonction *check* et la fonction *infer*. *infer* s'occupe de faire l'inférence des types. En fait, pour simplifier la tâche au programmeur, le langage ne le force pas à toujours spécifier les types. Le langage infère lui-même les types manquants grâce à deux systèmes d'inférence de types : d'abord de manière locale, puis en se servant de la totalité du code source. C'est la fonction *infer* qui fait ce travail. Cette fonction prend une expression de type **sexp**, le contexte d'élaboration et renvoie l'expression de la **sexp** après la phase d'élaboration ainsi que son type. Les deux éléments de sortie sont des **lexp** quant à eux. La fonction *check*, quant à elle, fait la vérification si le type attendu est bien le type qu'on a. Elle prend une expression de type **sexp**, le contexte d'élaboration, le type attendu et vérifie si le type attendu et le type de l'expression sont logiques. L'inférence et la vérification des types font donc l'élaboration, et pendant cette dernière, l'expansion des macros a également lieu. Mais plus concrètement, il n'y a pas vraiment d'ordre à ces

tâches, mais tout ça se fait dans une seule et unique phase. Alors, évidemment, il y a un ordre logique puisque s'il y a une expansion de macros à faire, elle doit être faite d'abord, puis ensuite, on peut prendre la *sexp* résultante de l'expansion et faire l'inférence ou la vérification selon la convenance. Mais tout ça se fait dans une même phase, il n'y a pas une phase qui vient avant une autre. Tout est géré par une seule grande fonction récursive qui appelle parfois *check* et parfois *infer*, et *check* et *infer* appellent eux-mêmes parfois *check* et parfois *infer*. L'expansion des macros est également faite par ces deux fonctions *check* et *infer*. En fait, *check* et *infer* sont mutuellement récursives. Si on appelle *infer* et que cette dernière ne sait pas comment faire l'inférence, car il n'a pas les informations nécessaires, elle crée une méta-variable et présume que c'est le type qu'il faut, et vérifie avec *check* si c'est effectivement le type attendu. Quant à la fonction *check*, si elle s'occupe d'un cas qui peut être inféré, elle fait l'inférence du type, puis vérifie que le type inféré et le type attendu sont les mêmes. Donc certains cas sont gérés dans *check* et d'autres dans *infer*, et les deux s'appellent dans les endroits qui conviennent le mieux.

À la fin de l'élaboration, tout est typé, et les types s'affichent tel que résultants de l'évaluation avec la partie implicite (s'il y a lieu) et la partie explicite. Les paramètres implicites ici désignent les paramètres de types que le programmeur n'écrit pas, mais qui sont retrouvés par l'élaboration. Dans l'exemple du code qui suit, on pourra voir des paramètres implicites :

```
1 multiply x y z = _*_ x y ;
2
3 %% multiply : (ℓ : ##TypeLevel) ≡> (τ : (##Type_ ℓ))  ≡>
4 %%             (x : ##Int) -> (y : ##Int) -> (z : τ) -> Int
```

Cet exemple montre une fonction qui prend 3 arguments et multiplie les deux premiers arguments puis laisse inutilisé le troisième argument. L'inférence de type détecte grâce à l'utilisation de la multiplication que **x** et **y** sont des entiers. Pour le type de **z**, l'inférence lui a donné comme type τ, qui lui-même est de type *Type* parce qu'on n'a pas utilisé **z** et donc on n'a pas plus de précisions. En Typer, $\equiv>$ est utilisé pour les arguments implicites effaçables. ℓ et τ sont des arguments implicites. Ici, notre fonction prend comme arguments implicites un ℓ qui a comme type le type des niveaux, **##TypeLevel**, et un τ qui a comme type l'application de **##Type_** à ℓ, ce qui donne à τ le type *Type*. Et c'est ce τ là qui est donné comme type à **z**. On voit donc ici qu'on n'a pas spécifié les types des arguments et que l'inférence trouve leurs types par rapport à leur utilisation dans le code.

3.2.3. L'expansion des macros

C'est à l'expansion des macros que les macros sont remplacées par du code qui est le résultat de leurs évaluations. Un appel de macro en Typer n'a pas une forme syntaxique

particulière par rapport à un appel de fonction. Seule une distinction de types est faite pour distinguer un appel de fonction d'un appel de macro. Les fonctions qui définissent des macros doivent prendre une liste de **sexp** en entrée et renvoyer une **sexp** comme on a pu le voir dans le Listing 1.7. L'expansion se fait pendant la vérification et l'inférence des types pour qu'on puisse avoir toutes les informations concernant les types. Comme on l'a vu dans le Listing 1.7, la fonction prise par le constructeur **macro** doit prendre en paramètres une liste de **sexp** et renvoyer une **sexp**. Et c'est cette **sexp** là qui sera utilisée pour la transformation en **lexp** durant la phase d'élaboration. Il faut savoir que ces macros sont des macros procédurales qui peuvent exécuter tout type d'opération, comme lire ou écrire des fichiers, ou même ne pas terminer. Et ceci a pour conséquence que, l'ordre du code après l'expansion ne reflète pas forcément celui du code source, et que même son contenu ne garantit pas de ressembler au code source. Cet aspect des macros a un impact significatif dans le cadre de notre travail qu'on verra un peu plus en détails dans le chapitre 5 à la Section 5.2.

Chapitre 4

Implémentation

Dans ce chapitre, nous allons présenter la partie d'implémentation de notre serveur LSP, allant de la vue d'ensemble du projet, des technologies utilisées jusqu'aux fonctionnalités du serveur.

4.1. Présentation du langage d'implémentation OCaml

OCaml est un langage de programmation paru pour la première fois en 1996 publié par l'institut de recherche Inria en France. Écrit en C et en OCaml, le langage est l'une des implémentations du langage Caml[**27**] descendant lui-même du langage ML. OCaml est également de la famille ML. Le langage est multiparadigme. Il supporte en effet la programmation impérative, fonctionnelle et orientée objet. Le langage est portable et performant. Il est utilisé dans des projets divers comme dans l'assistant de preuve *Coq* ou encore la version web de l'application *Facebook Messenger*.

4.1.1. Aperçu syntaxique

On ne va pas présenter ici tout le langage OCaml, mais on va plutôt se concentrer sur les notations qu'on pourra occasionnellement retrouver par la suite.

```
1 (*Définir des variables*)
2 let str = "une chaine"
3 let flt = 2.3
4 let tup = (str,flt)
5 let int_list = [1;2;3;4]
6
7 (*Faire des opérations mathématiques*)
8 let fl = 2.5 +. 2.8 (* +., -., *., /. sont utilisés pour les
     flottants*)
9 let mul = 2 * 4 (* +, -, *, / sont utilisés pour les entiers*)
```

```
10
11 (*Définir une fonction*)
12
13 (* Notons qu'en OCaml, il n'y a pas d'instruction return, la derni
     ère expression écrite sera renvoyée. *)
14
15 (*On peut spécifier le type des arguments et le type de retour comme
     suit. Ceci signifie que la fonction somme prend en paramètre deux
     entiers et renvoie un entier. rec signifie que la fonction est r
     écursive*)
16 let rec somme (x:int) (y:int) : int =
17   if x = 0 then y
18   else 1 + somme (x - 1) y
19
20 (* On peut aussi spécifier le type d'une fonction comme suit. Ceci
     signifie également que la fonction prend en paramètre deux
     entiers et renvoie un entier. *)
21 let rec produit x y : int -> int -> int =
22   if x = 1 then y
23   else y + produit (x - 1) y
24
25 (*On peut aussi ne pas spécifier le type de la fonction. let...in
     signifie qu'on définit une expression localement. Le match...with
     permet de faire des correspondances de modèle. '::' sépare le
     premier élément d'une liste du reste de la liste.*)
26 let list_remove lst el =
27   let rec foo lst el =
28     match lst with
29     | [] -> []
30     | hd::tl -> if (el = hd) then tl
31                 else hd::(foo tl el)
32   in
33 foo lst el
34
35 (*On peut définir un nouveau type avec type.*)
36
37 (*Définir un tuple de deux entiers. '*' est utilisé pour les tuples
     dans les définitions de types.*)
38 type tup_ent = int * int
39
```

```
40 (*Définition d'un record*)
41 type rgb = {r : int; g : int; b : int}
42
43 (*Un élément de type rgb*)
44 let my_color = {r=55; g=28; b=45}
45
46 (* On peut accéder à un élément d'un record avec un '.'*)
47 print_int my_color.r (*55*)
48
49 (* Un nouveau type peut avoir plusieurs constructeurs et chaque
      constructeur peut utiliser ou non n'importe quel type. Par
      exemple ici, on utilise le record précédemment défini*)
50 type colour =
51   | Red
52   | Green
53   | Blue
54   | RGB of rgb
```

Listing 4.1. Aperçu d'OCaml.

4.1.2. Système de types

OCaml est typé statiquement. Toutefois, il n'est pas nécessaire de préciser le type des expressions. En effet, le langage utilise un algorithme pour inférer automatiquement les types en fonction du contexte d'utilisation des expressions dans le code. OCaml supporte le polymorphisme paramétrique. Autrement dit, au moment de la définition, certaines parties peuvent être indéterminées. Par exemple, la fonction **map** du module **List** a comme type: **('a -> 'b) -> 'a list -> 'b list**. **'a** et **'b** peuvent être de n'importe que type. La fonction **map** prend donc en paramètre une fonction qui prend en paramètre un type **'a** et renvoie un type **'b**, une liste d'éléments de type **'a** et renvoie une liste d'éléments de type **'b**. On peut comprendre donc que c'est à l'utilisation de la fonction **map** qu'on spécifiera les types de **'a** et **'b** qui peuvent être des entiers, des flottants ou même des listes. Donc le langage supporte le polymorphisme des paramètres.

4.1.3. La programmation modulaire

OCaml supporte la programmation modulaire. Cette dernière est la possibilité de séparation d'un programme en différents modules. En général, chaque module définit de nouveaux types de données et des fonctions propres au module. Par exemple, **List** est un module définissant un nouveau type **t** défini comme suit:

```
type 'a t = 'a list =
|[]
|(::) of 'a * 'a list
```

Le module **List** contient des fonctions comme **hd** qui renvoie le premier élément d'une liste qu'elle prend en paramètre, ou encore **length** qui renvoie la longueur d'une liste donnée. Pour accéder à un élément d'un module, on peut utiliser un **'.'**. Pour accéder par exemple à la fonction hd du module **List**, on peut écrire **List.hd**. La lettre **t** est, par convention, utilisée pour définir le type du module. Par exemple, pour le module **String**, le type **t** du module représente le type **string**. Donc **String.t** est le type **string**, **Int.t** le type **int**, etc. Les modules commencent par une lettre majuscule dans le code en OCaml. Aussi, en OCaml, un fichier est tout simplement un module. Mais ce dernier commence par une minuscule même si on l'utilisera avec une majuscule dans le code. En effet, on ne peut pas créer un fichier OCaml commençant par une majuscule, à moins que tout le nom du fichier soit en majuscule. Étant donné qu'en OCaml, un fichier est un module, les fichiers dans le même dossier sont automatiquement connectés. Si on a dans le même dossier un fichier nommé **premier.ml** contenant un variable **a**, on peut accéder à la variable **a** dans un autre fichier du même dossier en écrivant tout simplement **Premier.a**. On peut créer des modules à l'intérieur des modules, des sous-modules. La logique d'accès reste la même. Par exemple, si notre fichier **premier.ml** a un sous-module nommé **Sub** dans lequel est définie une variable **b**, pour accéder à **b**, il faut écrire **Premier.Sub.b**. Notons qu'on peut ouvrir un module dans un autre, en portée globale ou locale, pour ne pas avoir à spécifier le nom du module quand on veut utiliser une variable de ce module et juste écrire le nom de la variable.

4.2. Vue d'ensemble

Deux communications sont nécessaires pour que notre serveur soit fonctionnel, d'un côté avec le compilateur pour pouvoir générer les informations dont le client a besoin, et d'un autre avec le client pour pouvoir lui envoyer les informations qu'il demande ou non selon le contexte. Le serveur a donc besoin de deux connexions. Le client notifie l'état du document ou envoie des requêtes avec des informations pour une demande particulière au serveur à un état précis du document. Ensuite, ce dernier communique avec le compilateur de Typer qui compile le code et transmet des informations au serveur qui les transmet à son tour au client selon ses nécessités. La vue d'ensemble du projet ressemble à l'image de la Figure 4.1.

4.3. Technologies utilisées

Le serveur est entièrement écrit en OCaml. Pour implémenter le serveur d'un langage, il n'est pas requis de l'écrire dans le langage en question, ou même de l'écrire dans le langage

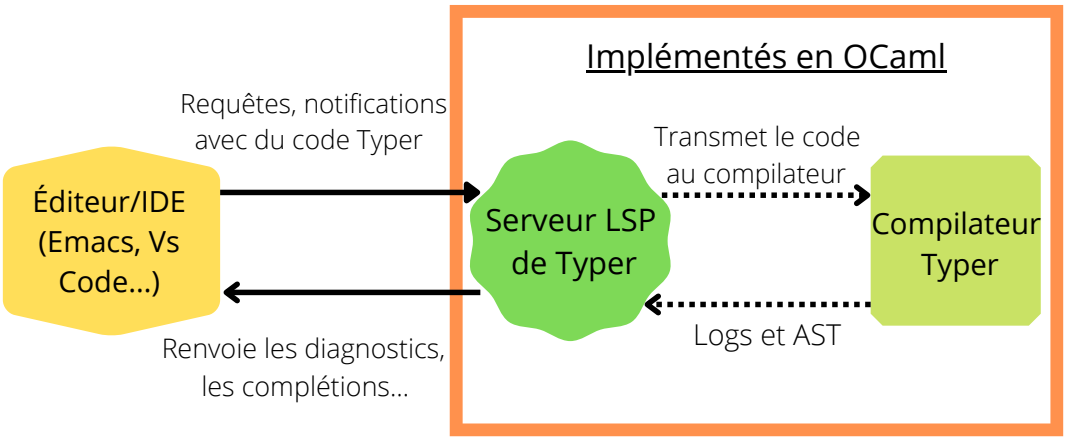

Fig. 4.1. Vue d'ensemble.

où est écrit le langage, mais on l'a fait en OCaml pour des raisons de connexion qu'on verra à la Section 4.4. Les serveurs LSP sont récents, ils n'existent que depuis 2016. Ainsi, les implémentations de serveurs de langues le sont aussi. Le premier serveur LSP pour le langage OCaml lui-même n'était pas écrit en OCaml, mais en TypeScript pour OCaml et Reason, il y a de cela plus de 4 ans, et c'était la seule implémentation existante d'un serveur de langue pour OCaml. Plus tard, une équipe liée aux développeurs d'OCaml même a écrit le premier serveur LSP écrit en OCaml pour OCaml. En effet, le 15-oct-2020, l'équipe d'OCaml Labs Consultancy (OCLC) a publié la première version d'un serveur LSP pour OCaml écrit en OCaml sur Opam, le gestionnaire de paquet d'OCaml. Ce travail a été basé sur Merlin qu'on a vu à la Section2.2 qui a été utilisé comme bibliothèque pour le projet. Pour implémenter leur serveur, ils ont dû développer une bibliothèque *Lsp* qu'ils ont aussi publié sur Opam. D'autres bibliothèques dont *Lsp* a besoin pour fonctionner ont été créées et intégrées au projet comme *Jsonrpc*. Dans le projet pour faire le serveur LSP d'OCaml, ils ont aussi utilisé la bibliothèque *Fiber*. Deux personnes de cette équipe ont créé une autre bibliothèque appelée *Linol*, intégrée à Opam aussi, qui utilise la bibliothèque *Lsp*, mais qui se concentre sur l'essentiel qu'offre la bibliothèque sans les autres dépendances spécifiquement utilisées pour le projet du serveur LSP d'OCaml. On a donc décidé d'utiliser cette bibliothèque-là,

tout en se référant à la bibliothèque *Lsp* au besoin pour avoir certaines informations sur comment marchent certaines fonctionnalités, ce qu'ils attendent comme informations et ce qu'ils envoient comme informations. En définitif, les technologies utilisées sont :

- le langage OCaml
- la bibliothèque Linol sur Opam
- la bibliothèque LSP comme documentation
- l'éditeur Emacs

4.4. Les connexions

Comme nous l'avons énoncé à la Section 4.2, nous avons besoin de faire deux connexions. La première est la connexion avec le compilateur du langage Typer. C'est pour cette tâche qu'on a utilisé le langage OCaml étant donné que Typer est écrit en OCaml. Le serveur est en fait un fichier OCaml ajouté à Typer. L'idée est que le serveur et le compilateur soient dans le même processus, c'est-à-dire dans le même langage de programmation. Ainsi, grâce aux modules du langage OCaml, le serveur peut donc se connecter naturellement à tous les fichiers sources de Typer. Comme ça, on peut utiliser des variables, faire des appels de fonctions entre le serveur et les autres fichiers composants le langage. Aussi, on n'a pas besoin de gérer des problèmes de synchronisation et de sérialisation/désérialisation. La seconde connexion concerne le serveur de langue et l'outil de développement. Les serveurs LSP supportent différents types de connexions comme les connexions par sockets, stdio, pipe, etc. Pour notre projet, on a utilisé une connexion pipe. Pour connecter le serveur à un outil de développement, il faut que l'outil soit un client supportant le protocole LSP. Sur Emacs par exemple, il y a le paquet *lsp-mode* et *eglot* qui sont fait pour que l'éditeur supporte des clients LSP. Il faut également faire quelques configurations selon ses préférences dépendamment de l'éditeur/IDE pour indiquer au client quoi démarrer comme serveur, quand le démarrer, quoi offrir comme fonctionnalités, comment les offrir selon ses préférences, etc. Pour notre cas, on a utilisé le fichier de configuration de l'éditeur Emacs pour écrire nos configurations pour les besoins du projet.

4.5. Choix d'un client

En général, le serveur devrait fonctionner pour n'importe quel client LSP presque sans travail supplémentaire. La seule chose fondamentale est de dire à l'éditeur quel exécutable utiliser pour lancer le serveur lorsqu'un fichier d'extension *.typer* est ouvert. Les configurations supplémentaires dépendent du choix de l'utilisateur. Par exemple, il peut décider de configurer l'affichage de la complétion d'une certaine manière comme on peut le faire sur Emacs en utilisant *company-mode*. Toutefois, pour notre projet, on a beaucoup utilisé

Emacs comme client et ce, pour plusieurs raisons. Typer étant encore en phase de développement, les utilisateurs actuels du langage se servent de l'éditeur Emacs ou le font avec un terminal. Ils utilisent Emacs parce que Typer dispose d'un fichier qui définit le mode Typer (*typer-mode*) sur Emacs, et qui fournit la coloration syntaxique. Le protocole LSP n'est pas adapté pour la coloration syntaxique, parce qu'il est un peu lent pour cette fonctionnalité. Donc pour tout le projet, on a utilisé l'éditeur Emacs avec *typer-mode* pour les besoins du projet et étendu en quelque sorte les fonctionnalités de *typer-mode* sur Emacs. On a également défini une commande pour démarrer le serveur qu'on a rajouté à *typer-mode*. La commande est **typer-lsp-mode**. La commande s'exécute donc avec le mode Typer, ce qui fait que le serveur se lance à l'ouverture d'un fichier d'extension *.typer* puisque ce dernier lance *typer-mode*.

4.6. Structure du serveur

Comme on l'a énoncé avant, le serveur utilise la bibliothèque *Linol*. Pour ce faire, on s'est basé sur le modèle simple de la bibliothèque que les développeurs ont mis à disposition pour apprendre à utiliser *Linol* pour structurer notre serveur.

```
1 type syntax_tree = (vname * Lexp.lexp * Lexp.ltype) list list
2
3 type syntax_tree_elab_ctx = (syntax_tree * Debruijn.elab_context)
4
5 type document_state = (log_entry list
6                         * (syntax_tree_elab_ctx option)
7                         * ((int * int) option ref))
```

Listing 4.2. Le type de l'état du document.

Comme on le sait, le client envoie continuellement des mises à jour sur des documents au serveur. Il notifie au serveur qu'un tel document est ouvert ou modifié, et lui envoie le contenu du document et son *uri*. Ensuite, le serveur utilise ce contenu pour l'élaborer afin de construire des informations sur le document pour ce contenu spécifique du code. Et ce sont ces informations qui sont utilisées pour répondre aux requêtes du client et lui envoyer les notifications de diagnostics, toujours pour ce contenu du code à ce moment précis. Et à chaque modification du document, ce processus se répète. Notons toutefois que plusieurs documents peuvent être ouverts ou modifiés à la fois. Pour gérer tous ces aspects, on s'est référé au modèle de Linol car il propose une bonne structure pour construire son serveur. Notre serveur possède donc les 3 éléments suivants primordiaux :

le type d'un état du document: ce type représente un état quelconque du document. Référons-nous au Listing 4.2 pour mieux comprendre. On a défini un état

particulier du document qu'on a appelé **document_state**. Comme on peut le voir, ce type est un tuple de 3 éléments :

- la **log_entry list** qui correspond à une liste d'informations sur le document enregistrées après la fin de l'élaboration. Cette liste est la liste contenant les diagnostics du document pour cet état précis du document.
- la **syntax_tree_elab_ctx option** qui est une option d'un tuple contenant la **syntax_tree**, qui représente l'arbre de syntaxe abstraite et le contexte d'élaboration. On l'a en option parce que cela peut ne pas se générer si l'élaboration échoue. Ce tuple de deux éléments est du type de ce que renvoie l'élaboration si elle se passe bien. Pour notre serveur, on n'utilise pas ce contexte d'élaboration. Par contre, l'arbre de syntaxe abstraite que génère l'élaboration, lui, on l'utilise pour construire la plupart de nos fonctionnalités.
- la dernière position connue du curseur (de type **(int * int) option ref**) pour enregistrer la dernière position du curseur trouvée. Cette information, on en a besoin pour des raisons techniques liées à l'implémentation des *breadcrums* qu'on verra à la Section 4.7.9. **int * int** nous donne la position de la ligne et la colonne, l'**option** c'est parce qu'on ne connaît pas toujours la position du curseur, et **ref** parce que cette position peut changer même si le document n'est pas modifié, et donc qu'on ne change pas d'état du document.

la fonction compile_string: cette fonction effectue l'élaboration d'un document sous format textuel en utilisant des fonctions présentes dans Typer pour cet effet. Ce choix s'est fait parce que les notifications d'ouverture et de modification envoient au serveur le contenu du document sous format textuel (chaîne de caractère) accompagné de son *uri*. la fonction **compile_string** renvoie une valeur de type **document_state** qui correspond à l'état spécifique du document. À chaque modification du code source, cette fonction sera appelée et renverra le nouvel état du document.

la table buffers : cette table de hachage est importante étant donné comment fonctionne les serveurs LSP. En effet, le client peut envoyer des informations sur plusieurs fichiers en même temps, parce que ce n'est pas forcément un seul fichier qui est ouvert. Et il envoie les informations de chaque document avec l'**uri** du document en question. Pour gérer les informations de tous ces fichiers, on a donc eu recours à une table de hachage, cette table **buffers**, qui associe chaque **uri** d'un document à l'état du document en question. Ainsi, on peut gérer l'ouverture et la modification de plusieurs fichiers parallèlement et leurs états respectifs.

4.6.1. Quelques fonctions utiles

Implémenter notre serveur demande d'écrire beaucoup de code dans lequel il y a beaucoup de fonctions. On présente ici certaines fonctions qui sont essentielles à comprendre pour la suite :

la fonction diagnostics: la fonction *diagnostics* est celle qu'on utilise pour construire les informations de diagnostics d'un document. Observons le Listing **??**.

```
let diagnostics : document_state ->
                  Lsp.Types.Diagnostic.t list
```

Listing 4.3. La fonction diagnostics.

Cette fonction prend un paramètre de type **document_state** et renvoie une liste d'éléments de type **Lsp.Types.Diagnostic.t**. Dans la bibliothèque **Lsp** d'**OCaml**, ce type désigne le type d'une information qu'attend le client d'une notification de type **PublishDiagnostics** [1] (On peut voir plus d'informations sur ce type dans le Listing 4.11).

Une fois que l'élaboration est faite, la fonction *diagnostics* peut être utilisée pour construire les diagnostics du document avec leurs sévérités.

la fonction browse_lexp: cette fonction est l'une de celles qui font le plus de tâches. Voyons tout d'abord la signature de la fonction avec le Listing 4.4.

```
let rec browse_lexp : Debruijn.lexp_context -> Lexp.lexp ->
                          Source.Location.t -> vname list ->
     Debruijn.lexp_context *
     (((Source.Location.t * string option) * int) option) *
     Lexp.lexp *
     Lexp.lexp option *
     Source.Location.t *
     vname list

(* En Typer, une position dans un fichier a le type:
 Source.Location.t
 Source.Location.t = {
      file : string;
      start_line : int;
      start_column : int;
      end_line : int;
      end_column : int;
 }
```

[1] Voir la Section 3.1.5.

```
17 *)
```

Listing 4.4. La fonction browse_lexp.

La fonction **browse_lexp** est multitâche. C'est une fonction récursive qui prend en paramètre un contexte, une **lexp**, une position (la position du curseur) et une liste de **vname** (variable name). La fonction effectue comme première tâche de renvoyer le contexte à la position dans le fichier prise en paramètre comme premier élément du tuple de retour. Ce contexte servira notamment pour construire des fonctionnalités telles que la *complétion*. Le deuxième travail que cette fonction exécute est de renvoyer une variable et sa position si le curseur est sur une variable comme second constituant du tuple renvoyé. En effet, *vname* est de type : **vname : (Source.Location.t * string option)**. La fonction renvoie la variable sur laquelle le curseur se positionne ainsi que sa position dans le contexte. Cette position est importante parce qu'en Typer, les variables sont représentées par position, plutôt que par nom. En effet, pour la définition des variables, Typer utilise les indexes de *De Bruijn*. Une variable est définie par la distance qui la sépare de la précédente déclaration. C'est cette distance qu'on appelle *indexe de De Bruijn*. Ces informations seront utiles pour trouver la position de la définition d'une variable. Le troisième constituant du tuple que renvoie notre fonction est une *lexp*. Une *lexp* est concrètement un nœud de l'arbre de syntaxe abstraite généré par l'élaboration. On recherche ce nœud pour pouvoir trouver le type de l'élément à la position d'entrée (la position du curseur). Pour retrouver ce type, la fonction traverse tout l'arbre et recherche récursivement le **nœud** ou le **sous-nœud** sur lequel le curseur se positionne. Elle renvoie ensuite le type de l'expression représentée par ce nœud de l'arbre. En général, ce type ne fait pas partie de l'arbre de syntaxe abstraite, c'est-à-dire qu'il n'est pas nécessairement présent dans le code source ni même dans la *lexp* après l'élaboration. Ce type est construit ou reconstruit pendant qu'on traverse le code récursivement dans notre fonction. Ce type sera utilisé pour fournir la fonctionnalité de *hover*. Le quatrième élément du tuple est une *lexp* particulière qui servira pour certains cas de la complétion. Le cinquième élément est l'information de positionnement du nœud trouvé pendant la traversée précédente pour trouver le nœud sur lequel se positionne le curseur. Il sera utile pour trouver le *hover* et d'autres fonctionnalités également. Le dernier élément du tuple est une liste de *vname*. Cette dernière sera nécessaire pour construire les *breadcrumbs*.

la fonction browse_defs: cette fonction vérifie si le curseur est positionné sur une variable. Jetons un œil au Listing 4.5.

```
let rec browse_defs : Lexp.lexp -> Source.Location.t ->
                          (Lexp.lexp * vname * int) option
```

Listing 4.5. La fonction browse_defs.

Notre fonction prend en paramètres une *lexp* et une position (la position du curseur). Elle vérifie ensuite si le curseur est sur une variable. Si c'est le cas, elle renvoie un tuple de 3 éléments : la *lexp* où cette variable peut apparaître, le nom de la variable ainsi que son numéro à ce moment précis. Ces informations serviront à retrouver toutes les références à la variable sur laquelle on se positionne.

la fonction find_references: cette fonction s'occupe de retrouver toutes les références à une variable. Examinons le Listing 4.6.

```
let rec find_references : Lexp.lexp -> vname -> int ->
                            vname list -> vname list
```

Listing 4.6. La fonction find_references.

Cette fonction prend en paramètre une **Lexp.lexp**, un **vname**, un entier et une liste de **vname**. On remarque que les 3 premiers arguments correspondent respectivement en termes de type aux 3 éléments de retour de la fonction *browse_defs*(sans l'option évidemment). En fait, cette fonction complète *browse_defs*. *browse_defs* est utilisée pour trouver la variable sur laquelle se positionne la souris, et *find_references* va chercher toutes les références à cette variable. Ces références, on les cherche dans la *lexp* que renvoie *browse_defs*. Cette recherche se fait par indexe, qui est notre 3ème argument. Et chaque référence trouvée, on vérifie si le nom de la référence correspond au nom de notre variable pour nous assurer d'avoir la bonne information. Ces références sont collectées récursivement dans la liste de **vname**, notre 4ème argument, qu'on renvoie par la suite à la fin des recherches.

4.6.2. Quelques mots sur Linol

Linol est une bibliothèque créée par Simon Cruanes et Guillaume Bury, deux personnes de l'équipe qui a développé le serveur LSP du langage OCaml. Ils ont conçu la bibliothèque pour qu'elle soit utilisée pour implémenter des serveurs LSP. La bibliothèque utilise elle-même la bibliothèque Lsp. Linol possède une classe surchargeable appelée *server*. Cette classe possède des méthodes à redéfinir qui sont les méthodes que le client appelle à travers les requêtes qu'il envoie au serveur comme la méthode **on_req_hover** qui sera appelée lorsque le client enverra une requête *hover*. Il y a également des méthodes à redéfinir qui sont appelées lorsque le client envoie une certaine notification comme la méthode **on_notif_doc_did_open** qui est appelé quand un document est ouvert. On a donc un ensemble de méthodes à surcharger pour chaque fonctionnalité qu'on souhaite prendre en charge avec notre serveur. Toutefois,

la bibliothèque a un nombre de méthodes limité à surcharger (elle ne supporte pas par défaut la requête qui construit les highlights par exemple), mais offre la possibilité aux utilisateurs de la bibliothèque d'étendre les fonctionnalités à leur guise à travers une méthode utilisée pour prendre en charge les requêtes non fournies par la bibliothèque.

4.7. Fonctionnalités du serveur

Ici, nous allons voir les requêtes et notifications que notre serveur supporte.

4.7.1. Initialize

La requête d'initialisation est la première requête à être exécutée par le client. En effet, si le serveur reçoit comme première requête une autre que celle-là, il doit renvoyer une erreur. Il en est de même pour les notifications à l'exception de la notification *Exit* qui peut demander au serveur de s'arrêter. Comme toutes les autres requêtes, elle nécessite une réponse. Pour la requête *Initialize*, la réponse peut être *null* s'il y a une erreur, ou une autre réponse qui informe le client des services que le serveur offre. Concrètement, la réponse renvoyée par le serveur informe le client que le serveur supporte des fonctionnalités précises telles que le *hover* et la *complétion*. Il est nécessaire que le serveur renvoie les informations complètes des fonctionnalités qu'il offre, parce que, sinon, la requête ne fonctionne pas (le client ne l'envoie jamais) même si elle a été implémentée dans le serveur. Cette requête permet également de démarrer le serveur. Conventionnellement, on démarre le serveur quand un fichier d'une extension précise est ouvert. Pour notre cas à nous, on lance le serveur quand *typer-mode* (le mode Typer sur Emacs) est actif, et ce dernier s'active quand un fichier d'extension *.typer* est ouvert. Donc, indirectement, on suit la convention. Observons le Listing 4.7 pour mieux comprendre. Toutes les requêtes que le client envoie au serveur sont, en réalité, des appels de méthodes. Pour appeler ces méthodes, il fournit donc les paramètres de la méthode qu'il appelle et cette dernière renvoie des résultats. Chaque requête envoie un seul paramètre. La bibliothèque *Lsp* qu'utilise *Linol* représente chaque paramètre par un record. On va se concentrer ici sur les types des paramètres que la requête du client envoie et ce qu'elle reçoit.

```
1 (* informations de la requête client *)
2
3 InitializeParams.t = {
4   processId : int option;
5   clientInfo : InitializeParams.clientInfo option;
6   rootPath : string option option;
7   rootUri : string option;
8   initializationOptions : Yojson.Safe.t option;
9   capabilities : ClientCapabilities.t;
```

```
10    trace : [ `Messages | `Off | `Verbose ] option;
11    workspaceFolders : WorkspaceFolder.t list option option;
12  }
13
14  (* réponse du serveur *)
15
16  InitializeResult.t Lsp.Client_request.t
17
18  InitializeResult.t = {
19    capabilities : ServerCapabilities.t;
20    serverInfo : InitializeResult.serverInfo option;
21  }
22
23  (* type de la méthode du serveur *)
24
25  InitializeParams.t -> InitializeResult.t Lsp.Client_request.t
```

Listing 4.7. La requête Initialize.

Le client envoie un paramètre de type **InitializeParams.t** et reçoit la réponse du serveur qui lui renvoie une information de type **InitializeResult.t** qui contient les **capabilities** (les services que le serveur offre).

4.7.2. DidOpenTextDocument

Comme on l'a expliqué dans la Section 4.6, cette notification ne se fait pas seulement avec un fichier, mais peut se faire avec plusieurs fichiers à la fois. Ceci permet et oblige le serveur à gérer plusieurs fichiers. Voyons dans le Listing 4.8 qui montre le contenu de cette notification.

```
1   (* informations de la notification client *)
2
3   DidOpenTextDocumentParams.t = {
4   textDocument : TextDocumentItem.t
5   }
6
7   TextDocumentItem. t = {
8       uri : DocumentUri.t;
9       languageId : string;
10      version : Integer.t;
11      text : string
12  }
```

Listing 4.8. La notification DidOpenTextDocument.

Quand un fichier d'extension *.typer* est ouvert, le client notifie ça au serveur en lui envoyant des informations telles que l'*uri* du document et le contenu textuel du document. La bibliothèque *Linol* nous fournit une méthode à redéfinir qui sera appelée à l'ouverture d'un fichier. On utilise ici la fonction **compile_string** puis la fonction **diagnostics** avec la valeur de retour de **compile_string** pour construire les diagnostics nécessaires envoyés au client. On envoie ensuite ces notifications au client pour qu'il puisse les afficher. Chaque document ouvert est rajouté à une table de hachage qui a comme clé l'**uri** du document et comme valeur l'état du document de type **document_state**. Cette table nous permet de gérer l'ouverture de plusieurs fichiers parallèlement.

4.7.3. DidChangeTextDocument

À chaque modification, le client nous le notifie également avec des informations spécifiques. Le Listing 4.9 montrent concrètement ces informations.

```
1  (* informations de la notification client *)
2
3  DidChangeTextDocumentParams.t = {
4      textDocument : VersionedTextDocumentIdentifier.t; contentChanges
       : TextDocumentContentChangeEvent.t list
5  }
6
7  VersionedTextDocumentIdentifier.t = {
8      uri : DocumentUri.t;
9      version : Integer.t
10 }
11
12 TextDocumentContentChangeEvent.t = {
13     range : Range.t Json.Nullable_option.t;
14     rangeLength : int Json.Nullable_option.t;
15     text : string
16 }
```

Listing 4.9. La notification DidChangeTextDocument.

Comment la notification fonctionne intérieurement, elle ne traite pas totalement à nouveau le contenu textuel, mais elle agit sur les changements. La liste des changements est envoyée au serveur. Linol gère ça de manière interne et fournit dans une méthode à redéfinir lors d'une notification de type *DidChangeTextDocument* l'ancien contenu textuel et le nouveau

contenu textuel du document parmi les paramètres de cette dernière. Cette notification est très importante, car c'est grâce à cette notification que notre serveur peut marcher de manière incrémentale. On utilise le terme incrémental ici pour signifier que les informations se mettrons à jour pendant que le programmeur écrit son code, mais pas dans le sens où on élabore seulement la partie modifiée. Quand le fichier est modifié, on refait le même processus qu'avant, c'est-à-dire qu'on utilise la fonction **compile_string** pour constituer un nouvel état du document, puis la fonction **diagnostics** pour construire les nouveaux diagnostics. Ces diagnostics sont ensuite transmis au client. Quant au nouvel état, on l'utilise pour mettre à jour la table de hachage. Autrement dit, la valeur correspondante à l'**uri** du document est modifiée au nouvel état du document.

4.7.4. DidCloseTextDocument

Cette dernière notification complète le cycle de vie d'un document. On peut observer dans le Listing 4.10 un aperçu des informations que le client envoie au serveur.

```
1 (* informations de la notification client *)
2
3 DidCloseTextDocumentParams.t = {
4     textDocument : TextDocumentIdentifier.t
5 }
6
7 TextDocumentIdentifier.t = {
8     uri : DocumentUri.t
9 }
```

Listing 4.10. La notification DidCloseTextDocument.

Le client envoie l'*uri* du document fermé au serveur. Linol nous fournit également une méthode à redéfinir qui est appelée quand un fichier est fermé. Après la fermeture d'un fichier, on enlève les informations sur le fichier de notre table de hachage **buffers** pour ne pas utiliser inutilement de la mémoire.

4.7.5. PublishDiagnostics

Les diagnostics représentent les erreurs, les avertissements, les informations et les indices (hints) d'un document. Ces notifications sont contrôlées par le serveur, et c'est ce dernier qui décide quand est-ce qu'il veut les transmettre au client. Voyons un peu à quoi ressemblent les informations de diagnostics que le serveur transmet au client dans le Listing 4.11.

```
1
2 (* informations de la notification serveur *)
```

```
3
4  Diagnostic.t = [ {
5      range : Range.t;
6      severity : DiagnosticSeverity.t Json.Nullable_option.t;
7      code : code_pvar Json.Nullable_option.t;
8      codeDescription : CodeDescription.t Json.Nullable_option.t;
9      source : string Json.Nullable_option.t;
10     message : string;
11     tags : DiagnosticTag.t list Json.Nullable_option.t;
12     relatedInformation : DiagnosticRelatedInformation.t list
                        Json.Nullable_option.t;
13     data : Json.t option
14 } ]
```

Listing 4.11. La notification PublishDiagnostics.

Les diagnostics, on les crée avec notre fonction **diagnostics**. On peut voir un exemple de l'affichage des diagnostics à la Figure 4.2 dans notre éditeur.

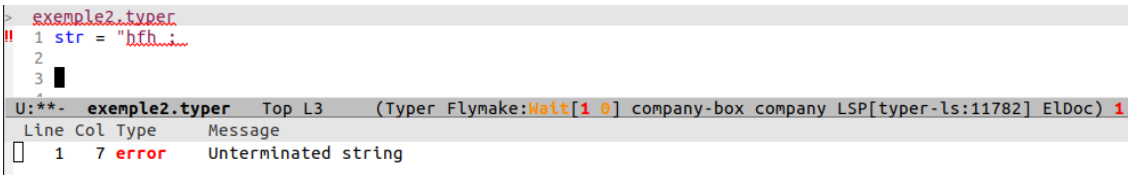

Fig. 4.2. Diagnostics.

Comme on peut le voir, on a déclaré une variable de type *string* et on n'a pas fermé les *doubles quotes*, le serveur renvoie un diagnostic de sévérité **Error** pour indiquer qu'il y a une chaîne de caractère non terminée à la ligne 1 et la colonne 7. Notons que le serveur envoie en réalité des intervalles de positionnement, c'est-à-dire la ligne de début et la ligne de fin, la colonne du début et celle de fin pour chaque diagnostic. C'est cette information qui permet au client de pouvoir surligner les diagnostics dans l'éditeur. Dans l'implémentation des diagnostics, on a rencontré comme enjeu principal la révision et le changement de la gestion des erreurs du compilateur. Quand l'élaboration se passait mal, des erreurs fatales (les erreurs internes au compilateur) pouvait se déclencher. Celles-ci ont pour conséquence d'interrompre la génération des **logs** (informations sur le document contenant les diagnostics) et de l'arbre de syntaxe abstraite. Ceci, par conséquent, empêche la construction des diagnostics à envoyer à l'éditeur. Cette génération ou non des *logs* et l'arbre de syntaxe, dans un usage habituel, ne fait pas de différence pour l'utilisateur parce que le code ne sera pas exécuté dans tous les cas. Mais dans le cadre du serveur LSP, on a besoin de les gérer pour ne pas interrompre le programme et générer des informations utiles à fournir à l'utilisateur comme diagnostics.

On a fait un travail pour gérer au cas par cas chacun des cas de figure où la compilation ne finissait pas pour pouvoir afficher les diagnostics. On l'a fait de deux manières :

- la première est d'éviter les cas qui ne sont pas nécessaires pour notre serveur et susceptibles de déclencher des erreurs fatales. Par exemple, notre serveur ne s'occupe pas de vérifier si l'élaboration est correcte, car on considère ça comme acquis. Et ces vérifications peuvent enclencher des erreurs fatales. Donc quand on démarre le serveur, par défaut, on ne fait pas ces vérifications. Toutefois, on a défini une option dans la commande pour démarrer le serveur qui peut activer la vérification de l'élaboration, bien qu'on ne gère pas ce cas. C'est l'option **–check** qu'il faut rajouter quand on démarre le serveur pour activer la vérification.
- La deuxième est de regarder les autres endroits restants et les gérer de manière à ce que l'élaboration finisse. En fait, ces problèmes ne sont pas inhérents à LSP, mais sont fondamentalement des corrections de classification d'erreurs dans Typer. Ces classifications sont beaucoup moins graves pour un usage habituel du compilateur en ligne de commande. En effet, une erreur implique que le code ne sera pas exécutable, donc qu'un arbre de syntaxe se génère ou pas, cela ne fait pas de différence pour l'utilisateur. En revanche, dans le cadre du développement du serveur LSP, on a besoin impérativement qu'un arbre de syntaxe se génère, et donc on a dû apporter certaines corrections. Par exemple, changer certains endroits où l'élaboration se passait mal et qui ne nécessitaient pas forcément des erreurs fatales, mais des erreurs normales, parce que ce sont seulement les erreurs fatales qui empêchent la génération des *logs* et de l'arbre. Ou encore, renvoyer un **Lexp.impossible** (la lambda expression qu'on envoie quand on ne peut pas en envoyer une de plus logique) à d'autres endroits où on ne peut pas avoir de **lexp** pour dire qu'on n'a pas pu en avoir une, ce qui veut dire que quelque chose s'est mal passée pendant l'élaboration. Mais cette dernière finit quand même afin de pouvoir avoir des informations qui aideront l'utilisateur à trouver la source de l'erreur en question.

Le deuxième enjeu des diagnostics, un peu plus mineur, est un problème d'affichage. En effet, Typer s'utilisait jusqu'ici principalement avec un terminal, donc était optimisé pour ce fait. L'affichage des diagnostics comportait des couleurs dans le terminal. Mais ces derniers, le client ne pouvait pas forcément les comprendre pour les afficher et donc affichait des messages parfois difficiles à lire, et surtout, il n'en a pas besoin, car les outils ont déjà leur manière d'imprimer les informations. Donc, on a défini des affichages spécifiques au serveur excluant toute couleur afin d'avoir des messages propres à transmettre à l'éditeur.

4.7.6. Hover

Le passage de la souris sur un élément est l'une des fonctionnalités qui ont été les plus délicates à faire. Ce qu'on affiche au passage de la souris sur un élément est son type. Voyons un peu à quoi ressemblent les informations échangées dans le Listing 4.12.

```
(* informations de la requête client *)

HoverParams.t = {
    textDocument : TextDocumentIdentifier.t;
    position : Position.t;
    workDoneToken : ProgressToken.t Json.Nullable_option.t
}

(* réponse du serveur *)

Hover.t = {
    contents : contents_pvar;
    range : Range.t Json.Nullable_option.t
}

(* type de la méthode du serveur *)

HoverParams.t -> Hover.t option Lsp.Client_request.t
```

Listing 4.12. La requête Hover.

Le client envoie une requête au serveur contenant la position de la souris et l'identifiant du document. Le serveur lui répond avec une information de positionnement et du contenu.

On peut voir dans notre éditeur comment le *hover* se présente sur Emacs avec la Figure 4.3.

Pour retrouver le type sous la souris, on utilise notre fonction **browse_lexp** qu'on a vu à la Section 4.6.1.

Quand on positionne la souris sur notre fonction **foo**, le client envoie une requête *hover* à laquelle notre serveur répond en fournissant le type de la fonction **foo** ainsi que ses informations de positionnement. Le client peut alors entreprendre des actions comme sélectionner la fonction et imprimer son type.

Dans notre exemple, la fonction **foo** prend en paramètres un *Integer*, une *liste de Sexp* et renvoie un *IO Sexp* et c'est ce que le serveur renvoie comme information de type avec les informations de positionnements de la fonction.

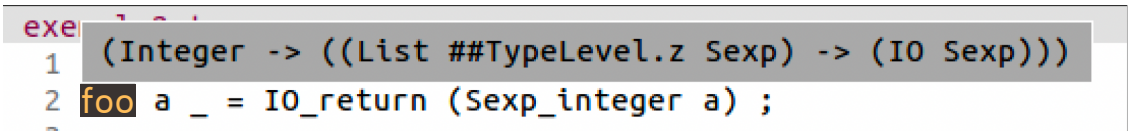

Fig. 4.3. Hover.

Pour faire cette fonctionnalité, on transforme d'abord tout le code en un gros **lexp**. Puis, on envoie cette **lexp**, le contexte initial et la position de la souris fournie par le client à notre fonction **browse_lexp**. Nous rappelons que **browse_lexp** doit traverser tout l'arbre dans la recherche du nœud approprié pour pouvoir renvoyer le type de l'expression représentée par ce nœud. Ceci est dû à l'expansion des macros qui peuvent réordonner le code de manière totalement arbitraire.

Le type que nous renvoie **browse_lexp** est la valeur résultante d'une suite de réductions faite pendant l'élaboration. Dans l'affichage, ce n'est pas toujours beau à voir, donc on cherche, après le renvoie du type par **browse_lexp**, à trouver une forme plus propre à envoyer au client (cet aspect est détaillé un peu plus dans le chapitre suivant à la Section 5.3).

Une fois qu'on a le type et sa position renvoyée par notre fonction **browse_lexp**, ainsi qu'une forme élégante pour imprimer le type, le serveur peut renvoyer une réponse au client. Et on le fait en redéfinissant la fonction que Linol fournit pour répondre aux requêtes de type *hover*.

4.7.7. Goto Definition

Quand on écrit du code, on peut avoir besoin de retrouver où un certain identificateur a été défini. Le serveur prend en charge cette fonctionnalité de retrouver la position de définition d'un identificateur. Voyons les informations de type de cette requête dans le Listing 4.13.

```
1 (* informations de la requête client *)
2
3 DefinitionParams.t = {
```

```
4       textDocument : TextDocumentIdentifier.t;
5       position : Position.t;
6       workDoneToken : ProgressToken.t Json.Nullable_option.t;
      partialResultToken : ProgressToken.t Json.Nullable_option.t
7   }
8
9   (* réponse du serveur *)
10
11  Locations.t = [
12      `Location of Location.t list
13      | `LocationLink of LocationLink.t list
14  ]
15
16  (* type de la méthode du serveur *)
17
18  DefinitionParams.t -> Locations.t option Lsp.Client_request.t
```

Listing 4.13. La requête Goto Definition.

Le client envoie une requête de type *Goto Definition* avec des informations de positions du curseur et l'*uri* du document. Le serveur lui répond en lui envoyant une location précise. On a une liste ici parce qu'on peut avoir plus d'une définition. Dans le cas de Typer, la position de la définition est unique. Voyons une illustration de comment se présente cette fonctionnalité sur Emacs à la Figure 4.4.

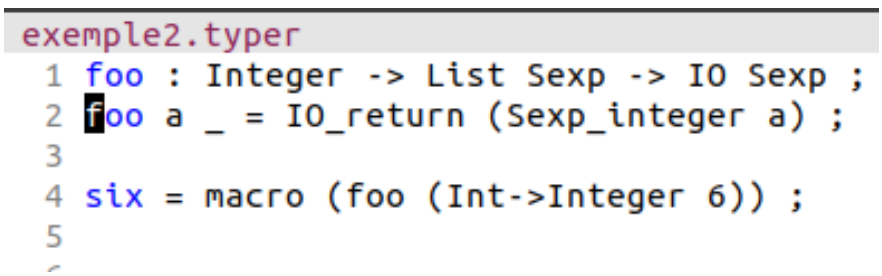

Fig. 4.4. GoToDefinition.

Le curseur était initialement positionné à la ligne 4 sur la fonction **foo**. Quand on fait un clic droit dessus et qu'on demande d'aller à la définition, une requête de type *Goto Definition* est envoyée au serveur et ce dernier répond avec le positionnement à la ligne 2 et colonne 1. L'éditeur entreprend donc des actions avec l'information reçue. Sur Emacs, le curseur se déplace à cette position, donc à la ligne 2 sur la fonction **foo**, qui est l'endroit où elle a été définie.

Pour offrir cette fonctionnalité, on utilise encore notre fonction **browse_lexp** pour chercher l'information. Quand le curseur est sur une variable, elle renvoie cette variable là.

browse_lexp renvoie également le contexte à la position du fichier. L'information de position de la définition est directement disponible dans le contexte. Pour la retrouver, on utilise une fonction en Typer appelée **lctx_lookup** . Cette fonction prend un contexte et une variable en paramètre et renvoie l'information associée à cette variable: la position de sa définition, son type et sa définition (s'il y en a une).

Une fois que l'information est retrouvée, on la transmet au client à travers la fonction de Linol qui répond aux requêtes de *GoToDefinition*.

4.7.8. Completion

La complétion est l'une des fonctionnalités les plus essentielles à mettre en place. Le code 4.14 nous donne un peu d'informations sur les types échangés.

```
(* informations de la requête client *)

CompletionParams.t = {
    textDocument : TextDocumentIdentifier.t;
    position : Position.t;
    workDoneToken : ProgressToken.t Json.Nullable_option.t;
   partialResultToken : ProgressToken.t
                          Json.Nullable_option.t;
    context : CompletionContext.t Json.Nullable_option.t
}

(* réponse du serveur *)

CompletionList.t = {
    isIncomplete : bool;
    items : CompletionItem.t list
}

||

CompletionItem.t = [ {
    label : string;
    kind : CompletionItemKind.t Json.Nullable_option.t;
    tags : CompletionItemTag.t list Json.Nullable_option.t;
    detail : string Json.Nullable_option.t;
    documentation : documentation_pvar Json.Nullable_option.t;
   deprecated : bool Json.Nullable_option.t;
    preselect : bool Json.Nullable_option.t;
```

```
27      sortText : string Json.Nullable_option.t;
28      filterText : string Json.Nullable_option.t;
29      insertText : string Json.Nullable_option.t;
30      insertTextFormat : InsertTextFormat.t
                    Json.Nullable_option.t;
31      insertTextMode : InsertTextMode.t Json.Nullable_option.t;
    textEdit : textEdit_pvar Json.Nullable_option.t;
    additionalTextEdits : TextEdit.t list
                      Json.Nullable_option.t;
32      commitCharacters : string list Json.Nullable_option.t;
33      command : Command.t Json.Nullable_option.t;
34      data : Json.t option
35  } ]
36
37  (* type de la méthode du serveur *)
38
39  CompletionParams.t -> [
40      `CompletionList of CompletionList.t
41      | `List of CompletionItem.t list
42  ] option Lsp.Client_request.t
```

Listing 4.14. La requête Completion.

L'éditeur nous fournit l'*uri* du document ainsi que la position du curseur à travers une requête de complétion à laquelle le serveur répond en fournissant la liste des complétions. La complétion permet de faire beaucoup de choses, comme nous aider à écrire les variables déjà déclarées correctement. La complétion nous aide à compléter pour nous ce qu'on essaie d'écrire. Elle nous permet de coder plus rapidement et correctement, nous fait des suggestions sur ce dont on n'est plus sûr de l'écriture exacte, ainsi que les autres suggestions qu'on ne connaît pas forcément. La complétion peut aussi servir de documentation. En effet, on peut trouver toutes les fonctions d'un module particulier par exemple. Un autre aspect important de la complétion est qu'il ne nous suggère pas seulement du texte brut, mais il nous suggère des éléments de complétion avec son détail pour savoir ce qu'on utilise précisément (dans le record de retour **CompletionItem.t**, cela correspond au champ *detail*). Pour le cas de notre serveur, chaque élément de complétion est accompagné de son type comme *detail*, ainsi que de son *kind* (le champ *detail* de **CompletionItem.t**) qui va nous préciser si on est face à une fonction, une variable, etc. On a illustré ce qu'on a visuellement avec la complétion avec la Figure 4.5. Comme nous pouvons le constater, notre fichier ne contient qu'une seule déclaration: la variable **my_str**. Dans l'éditeur, on vient de saisir *my*. Quand on commence à écrire, l'éditeur nous propose des suggestions à compléter. Comme on peut le constater,

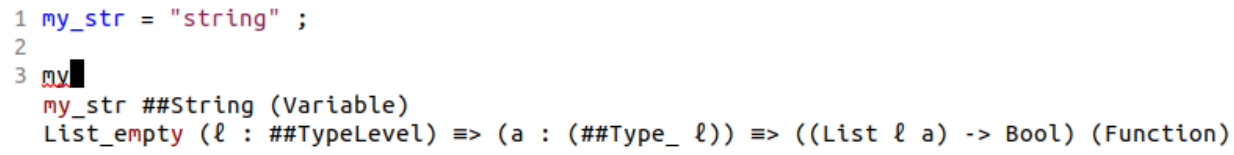

Fig. 4.5. Completion.

il nous suggère **my_str**, notre variable de type **String** qu'on a déclaré à la ligne 1, ainsi que la fonction **List_empty**, une fonction qui vérifie si une liste est vide, parce que ce sont les deux seuls éléments que le serveur a trouvés qui contiennent les lettres **m** et **y** dans cet ordre suite à ce qu'on a saisi. Pour implémenter la complétion, on utilise encore une fois la fonction **browse_lexp**. On a besoin en parallèle de retrouver le mot qu'on veut compléter. Et pour ce faire, étant donné qu'on a la position du curseur et le code du fichier ouvert en chaîne de caractère, il nous suffit de reculer à partir de cette position et collecter caractère par caractère vers la gauche jusqu'à trouver un séparateur comme un **';'**, une **','**, etc. Ces séparateurs sont définis dans le langage Typer et on les a répertoriés dans une liste pour les besoins de notre serveur. Parallèlement, on utilise **browse_lexp** qui nous renvoie le contexte qui s'applique à la position du curseur. On utilise ce contexte pour construire les complétions en général. Il existe d'autres cas où les complétions se trouvent autrement. Si on a un code source comme **List.length**, la liste des complétions sera retrouvée dans le type du module *List*. Ces complétions se cherchent de cette manière pour des raisons qu'on verra à la Section 5.1. Une fois qu'on a toutes les complétions qu'il nous faut, on les filtre par rapport au mot qu'on avait retrouvé au tout début. On garde tout d'abord les complétions qui contiennent notre mot dans un premier filtre, et puis on filtre encore une deuxième fois par rapport aux complétions qui contiennent les caractères du mot dans l'ordre. Ensuite, on envoie la liste des complétions filtrées au client.

4.7.9. Document Symbols

Les symboles, communément appelés *breadcrumbs*, indiquent le chemin emprunté pour se retrouver à l'endroit où on se situe. Voyons le Listing 4.15.

```
1 (* informations de la requête client *)
2
3 DocumentSymbolParams.t = {
4     workDoneToken : ProgressToken.t Json.Nullable_option.t;
    partialResultToken : ProgressToken.t
                    Json.Nullable_option.t;
5     textDocument : TextDocumentIdentifier.t
6 }
7
```

```
8  (* réponse du serveur *)
9
10 DocumentSymbol.t = [ {
11     name : string;
12     detail : string Json.Nullable_option.t;
13     kind : SymbolKind.t;
14     tags : SymbolTag.t list Json.Nullable_option.t;
15     deprecated : bool Json.Nullable_option.t;
16     range : Range.t;
17     selectionRange : Range.t;
18     children : t list Json.Nullable_option.t
19 } ]
20
21 ||
22
23 SymbolInformation.t = [ {
24     name : string;
25     kind : SymbolKind.t;
26     tags : SymbolTag.t list Json.Nullable_option.t;
27     deprecated : bool Json.Nullable_option.t;
28     location : Location.t;
29     containerName : string Json.Nullable_option.t
30 } ]
31
32 (* type de la méthode du serveur *)
33
34 DocumentSymbolParams.t -> [
35     `DocumentSymbol of DocumentSymbol.t list
36     | `SymbolInformation of SymbolInformation.t list
37 ] option Lsp.Client_request.t
```

Listing 4.15. La requête Document Symbols.

La requête *Document Symbols* est envoyée par le client pour trouver les symboles d'un document. À travers la requête, le client communique au serveur des informations comme l'*uri*. Toutefois, le client ne nous fournit pas la position du curseur. Ceci est problématique parce qu'on en a besoin pour trouver les symboles, étant donné qu'en Typer, il y a des symboles en fonction de la position du curseur. On a donc dû trouver une alternative pour trouver la position du curseur. C'est la raison pour laquelle le type **document_state** a un tuple de deux entier comme troisième élément qui le compose. Ce tuple enregistre la dernière position du curseur trouvée. Cette information nous vient des autres requêtes que le client envoie

au serveur telles que *Document Highlights* ou encore *Completion*. Cette fonctionnalité nous permet de savoir dans quelle fonction on est et/ou sur quelle variable on est. À la Figure 4.6,

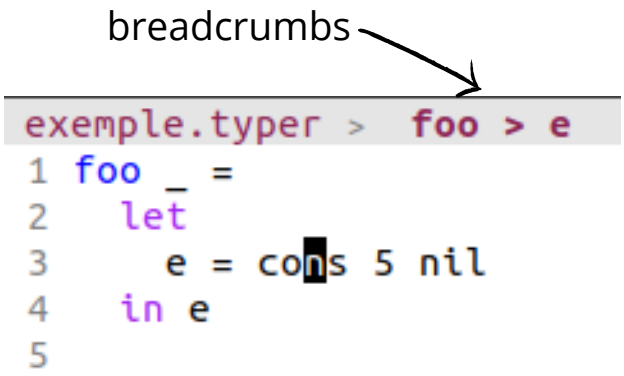

Fig. 4.6. Breadcrumbs.

on peut avoir un aperçu de cette fonctionnalité. Comme on peut le voir, le curseur se positionne dans la liste *e*. Le serveur nous renvoie les *breadcrumbs* retrouvés à cette position du curseur, à savoir **foo > e**. Le serveur a trouvé qu'on est à l'intérieur d'une fonction nommée *foo* dans une liste nommée *e*. Le serveur envoie cette information au client, et ce dernier l'affiche en général en haut du fichier comme c'est le cas pour Emacs.

Pour retrouver les *breadcrumbs*, on utilise encore une fois notre fonction **browse_lexp**. Cette dernière recherche récursivement dans quelles variables/fonctions le curseur est, puis collectionne ces informations dans une liste jusqu'à la position du curseur. C'est cette liste le dernier élément du tuple renvoyé par **browse_lexp**. Les informations à l'intérieur de cette liste sont ensuite utilisées pour construire la liste des symboles à renvoyer à l'éditeur.

4.7.10. Document Highlights

Cette fonctionnalité sélectionne toutes les occurrences du même élément dans le même document. Observons le Listing 4.16.

```
1 (* informations de la requête client *)
2
3 DocumentHighlightParams.t = {
4     textDocument : TextDocumentIdentifier.t;
5     position : Position.t;
```

```
6       workDoneToken : ProgressToken.t Json.Nullable_option.t;
    partialResultToken : ProgressToken.t Json.Nullable_option.t
7   }
8
9   (* réponse du serveur *)
10
11  DocumentHighlight.t = [ {
12      range : Range.t;
13      kind : DocumentHighlightKind.t Json.Nullable_option.t
14  } ]
15
16  (* type de la méthode du serveur *)
17
18  DocumentHighlightParams.t -> DocumentHighlight.t list option
```

Listing 4.16. La requête Document Highlights.

Le client envoie une requête *Document Highlights* avec l'*uri* et la position du curseur. Le serveur lui répond en lui envoyant les *highlights* du document à la position du curseur. Autrement dit, il fournit au client toutes les positions des occurrences de la variable (si on est sur une variable) sur lequel le curseur se positionne. Le client va en général marquer ces occurrences ainsi que la variable par un surlignement en même temps.

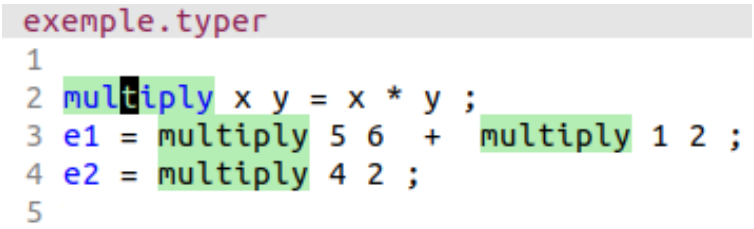

Fig. 4.7. Highlights.

La Figure 4.7 nous en offre une illustration. Dans cet exemple, la fonction **multiply** définie à la ligne 2 a été utilisée à la ligne 3 et la ligne 4. Notre serveur détecte ces informations et les renvoie au client en réponse à sa requête *Document Highlights*. Le client peut alors sélectionner tous les endroits où la fonction **multiply** apparait. La recherche des *highlights* se fait en deux étapes. Avec la fonction **browse_defs** qu'on a vu à la Section 4.6.1, le serveur trouve la variable sur laquelle le curseur se positionne s'il est sur une variable ainsi que le nœud où elle peut apparaître, puis avec la fonction **find_references** (également vu à la Section 4.6.1), le serveur trouve toutes les références à la variable trouvée avec **browse_defs**. Ainsi, il peut envoyer ces informations-là au client comme réponse à la requête *Document Highlights*.

4.7.11. Find References

Cette fonctionnalité ressemble un peu à *Document Highlights*, mais à une portée différente. En effet, les *highlight* concernent un seul fichier et les *références* concernent tout le projet. Voyons dans le Listing 4.17 pour voir les types des informations échangées.

```
(* informations de la requête client *)

ReferenceParams.t = {
    textDocument : TextDocumentIdentifier.t;
    position : Position.t;
    workDoneToken : ProgressToken.t Json.Nullable_option.t;
   partialResultToken : ProgressToken.t
                  Json.Nullable_option.t;
    context : ReferenceContext.t
}

(* réponse du serveur *)

Locations.t = [
        `Location of Location.t list
        | `LocationLink of LocationLink.t list
]

(* type de la méthode du serveur *)

ReferenceParams.t -> Location.t list option Lsp.Client_request.t
```

Listing 4.17. La requête Find References.

Le client envoie une requête de type *Find References* au serveur avec des informations telles que l'*uri* et la position du curseur. Le serveur renvoie au client les emplacements de toutes les références dans une liste. En Typer, on ne peut pas, pour l'instant, trouver les références à une échelle d'un projet, mais seulement d'un fichier. Ceci fait que les *highlight* et les *références* dans notre serveur ont des informations équivalentes. Toutefois, dans les éditeurs, ces deux fonctionnalités ne sont pas présentées de la même manière d'une manière générale.

En effet, les **Highlights** sélectionnent toutes les occurrences d'un élément, de l'endroit où il est déclaré aux endroits où il est utilisé dans le document. Les références, quant à elles, permettent de naviguer entre ces endroits-là. On peut naviguer de l'endroit où une variable a été introduite, par exemple, à tous les endroits où on l'utilise. Notons qu'on n'a pas de sens ni d'ordre imposé. On peut passer de la première référence à l'avant-dernière, puis finir

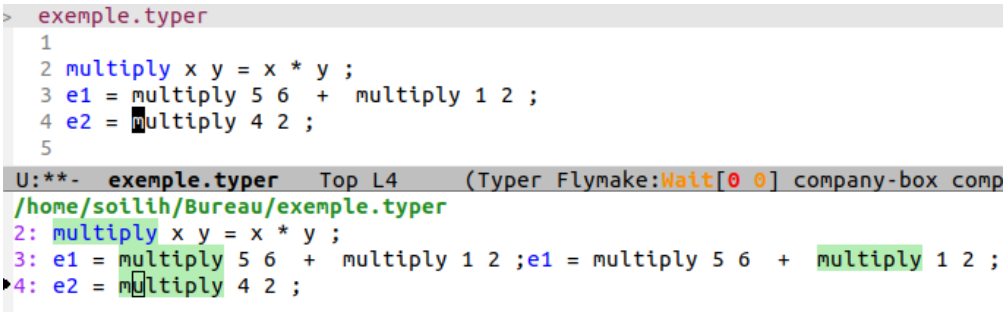

Fig. 4.8. References.

sur la deuxième. La raison qui fait ça est que les occurrences se présentent sous la forme d'une liste cliquable où chaque clic sur une référence te ramène à l'emplacement précis où elle a été utilisée.

On peut voir une démonstration dans la Figure 4.8. Comme on peut le voir sur la Figure 4.8, la fonction **multiply** est définie à la ligne 2, puis utilisée à la ligne 3 et 4. Quand on demande les références de *multiply*, le serveur lui renvoie une liste qu'on peut voir de 4 éléments (incluant la définition). Dans notre exemple, on a cliqué sur l'élément à la 4ème ligne de la liste, et l'éditeur place le curseur à l'endroit de l'utilisation de la référence à **multiply**, à la ligne 4 dans le code. Le travail fait pour trouver les références est le même que celui des *highlights*. En effet, ces deux fonctionnalités demandent des informations similaires dans notre cas. Donc, on utilise nos deux fonctions **browse_defs** et **find_references** vues à la Section 4.6.1 pour retrouver les informations nécessaires à renvoyer au client.

Chapitre 5

Problèmes rencontrés

Dans l'implémentation du serveur, on a rencontré beaucoup de problèmes. On en a résolu certains complètement, et d'autres de manière partielle. Certains de ces problèmes se généralisent, et d'autres sont spécifiques à Typer.

5.1. Cas de la complétion

Pour fournir la plupart des fonctionnalités du serveur, on utilise l'arbre de syntaxe abstraite généré par l'élaborateur. Pour la complétion, on est tombé sur un cas qu'on observe souvent quand on programme avec des macros.

5.1.1. Spécificité du problème

Pour mieux comprendre le problème, jetons un œil au code suivant:

```
1 %%  Code Typer:
2 List.map
3 %% Code après expansion:
4 Case List | tuple (map := x) => x
5 %% Code généré:
6 Case List | tuple x1 x2 x3 x4 x5 x6 x7 x8 ...
7                 => x5 (celui qui correspond à map)
```

Listing 5.1. Code généré avant Proj.

Quand on écrivait **List.map** par exemple, on passait par deux étapes. D'abord, l'expansion des macros transformait le **List.map** au code de la ligne 4 qui dit fondamentalement dans notre exemple qu'on va attendre dans la variable **x** la valeur du champ qui s'appelle **map**. On peut remarquer que jusqu'à cette étape, le code est constant. Ensuite, venait la phase d'élaboration qui transformait le code en un nœud **Case** qui, sachant que **List** est en fait un tuple de taille N, extrayait tous les champs pour finalement n'en renvoyer qu'un (celui

qui contient la valeur **map**). Dans notre exemple, le champ correspondant à **map** est **x5**. Mais en réalité, ces champs sont des variables, et étant donné que les variables en Typer sont représentées par des indexes de De Bruijn, on nous renvoie un certain numéro correspondant à la variable **x5**.
On a deux problèmes fondamentaux ici :

- le premier est que si on a un champ qui s'appelle **mapi**, qui est une complétion possible, à partir du code généré, on ne peut pas le retrouver.
- le second est qu'on perd totalement l'information du fait que c'était **map** qui était écrit.

Aussi, quand on avait comme code Typer *List.foo* alors que le champ *foo* n'existe pas, l'élaboration échouait tout simplement, et donc on n'avait pas d'arbre de syntaxe généré, ce qui est légitime pour du code Typer, mais dans le cadre d'un serveur LSP, ça nous coinçait totalement. Donc, on ne pouvait pas offrir la complétion avec ce code généré pour toutes ces raisons. Quand on programme avec des macros, on peut rencontrer ce genre de problèmes où les informations dont on aurait besoin pour effectuer une certaine tâche ne sont pas disponibles après l'expansion parce qu'elles ont été utilisées de manière interne pendant l'expansion et ne sont plus disponibles après. On pourrait citer comme exemple le langage Racket[**17**], un langage connu pour son puissant système de macros. Avec Racket, on peut créer des langages dédiés au domaine avec des macros. Si on prend un de ces langages dédiés, et qu'on fait l'expansion, et qu'à partir du résultat de l'expansion, on cherche à trouver le genre d'informations qui nous serviraient à faire de la complétion, on ne pourrait pas parce que certains identificateurs essentiels à la fonctionnalité ont été utilisés de manière interne pendant l'expansion et ne sont plus disponibles avec le code généré. Dans le même ordre d'idées, pour un exemple plus concret, si on a une macro qui fait une analyse syntaxique, comme le fait l'outil Lex (souvent utilisé avec Yacc qui fait le parsing)[**16**], le code généré, serait une table contenant un ensemble de tableaux correspondant aux différents états de la machine à état qui serait généré. Donc à partir de cette table-là, on ne pourrait pas fournir des fonctionnalités comme la complétion, parce que les informations qu'il faudrait pour le faire apparaissent dans la macro de départ, mais pas dans le code généré. Donc des problèmes de cet ordre peuvent souvent apparaître quand on programme avec des macros. Quand on faisait la complétion, on est tombé sur un problème similaire où les informations qu'il nous faudrait pour fournir la complétion, à savoir par exemple les fonctions spécifiques à un module donné, n'étaient pas disponibles avec l'arbre de syntaxe généré.

5.1.2. Solution proposée

Des problèmes similaires, à savoir préserver les informations qu'on avait avant l'expansion des macros, ou garder assez d'informations pour fournir la fonctionnalité dans le cas où

l'élaboration échoue, ne sont pas triviales à résoudre. Les articles 'Fortifying macros'[**23**] et 'Taming macros'[**24**] apportent des débuts de solutions possibles dans ce genre de problèmes. En effet, pour résoudre les problèmes des systèmes de macros existants en Racket, à savoir que leurs applications n'étaient pas forcément correctes, que leurs définitions et leurs spécifications ne se ressemblaient pas forcément, et pour le souci d'écrire des macros robustes (*Fortifying macros*) ou encore pour apporter des solutions à déboguer les constructions de programmation que sont les macros en Scheme (*Taming macros*), l'auteur propose dans ces articles des approches de résolution qui changent la manière de définir les macros. De ce fait, pendant, leur expansion, on peut savoir si on a "une expression", "une déclaration", etc. Des solutions similaires pourraient nous aider dans un cas comme le nôtre. Ils pourraient nous permettre de savoir si on complète juste une variable ou le nom d'un module par exemple. Étant donné que c'est avant l'expansion qu'on a ces informations, cela pourrait être utile dans le cas où l'élaboration échouerait à cause d'une erreur dans le code ou d'un code incomplet comme on l'a vu dans la section précédente. Toutefois, dans ce cas-là, on n'aurait pas des informations comme le contexte de typage pour fournir la complétion avec, étant donné que l'élaboration échouerait justement. Donc ces articles offrent des pistes intéressantes pour résoudre ce genre de problèmes de manière générale, mais ils n'offrent pas vraiment une solution à notre problème spécifique. Par contre, ces travaux pourraient être intéressants à être adaptés au système de macros de Typer dans l'espoir de fournir de meilleures informations au serveur LSP. Le serveur sera facilement adaptable pour en tirer profit. Cependant, on a recouru à une autre approche pour résoudre le problème. On a rajouté une extension au constituant cœur du langage, les **lexp**. On a rajouté un nouveau constructeur aux **lexp**, le constructeur **Proj**, *Proj* étant un diminutif de projection. Notre nouveau constructeur est défini comme suit:

```
1 | Proj of U.location * lexp * label
```

Listing 5.2. Constructeur Proj.

Ce constructeur interviendra à chaque fois qu'il y aura un **'.'** dans le code, que ce soit avec un module ou autre chose, c'est-à-dire n'importe quel tuple/record/struct. Le constructeur a tout d'abord une *location* pour connaître l'emplacement de la **lexp**, une **lexp** qu'on utilisera pour retrouver les complétions, et un label qui sera la chaîne écrite après le **'.'** qui, elle-même, a sa location spécifique. Regardons le code suivant qui est l'évolution du Listing 5.1 mais avec notre nouveau constructeur:

```
1 %% Code Typer:
2 List.map
3 %% Code généré:
4 Proj List map
```

Listing 5.3. Code généré avec Proj.

Avec le nouveau constructeur **Proj**, pour le code *List.map*, on a la trace du **map** dans le label et on peut chercher le type de **List** grâce à quoi on pourra fournir les complétions. Donc, on a tout ce qu'il faut pour offrir la fonctionnalité. Et en cas d'erreur, des erreurs où on écrirait par exemple *List.foo*, alors que la fonction *foo* n'existe pas dans le module **List**, pour le constructeur **Proj**, ça ne change rien. On aura encore le module **List** et le label **foo**. Cette solution résout donc totalement notre problème. Et par la même occasion, cette solution n'est pas seulement essentielle au serveur LSP, mais elle l'est aussi pour le langage en général. Pour le code précédant utilisant le constructeur **Case**, le nombre de champs du tuple **List** avait une influence sur la vitesse par exemple. Si le tuple avait 100 champs, et qu'on va chercher le 80ème, on mentionne tous les champs parce qu'on le fait par position plutôt que par nom, pour finalement n'en renvoyer qu'un seul, le 80ème. Donc plus un tuple avait de champs, plus le code devenait plus grand également. Mais avec le nouveau code avec le constructeur **Proj**, on a du code beaucoup plus concis, et plus efficace. La taille du code reste inchangée peu importe le nombre de champs du tuple.

5.2. Cas de la recherche du type d'un élément à la position de la souris avec le *hover*

Ce problème affecte toutes les fonctionnalités qui utilisent la fonction **browse_lexp** telle que la complétion, mais le *hover* en est plus affecté à cause de la précision de l'information qu'il demande. Le *hover* est une des fonctionnalités de notre serveur, et l'une des plus importantes, car elle fournit des informations sur les types et Typer accorde beaucoup d'importance aux types. On a pu voir comment elle fonctionne à la Section 4.7.6. Durant l'implémentation de cette fonctionnalité, nous avons rencontré certains problèmes dont celui de la recherche du type de l'élément sur lequel se positionne la souris dans l'arbre de syntaxe généré par la phase d'élaboration.

5.2.1. Spécificité du problème

Quand la souris se positionne sur un élément, on cherche à renvoyer le type de l'élément en question dans le cadre de la fonctionnalité de *hover*. Pour cette fonctionnalité, on a été confronté à un problème principal qui est spécifique au langage Typer. Le problème est que, l'expansion des macros en Typer peut réordonner le code de manière complètement arbitraire entre le code source et la forme élaborée à cause, notamment, de l'expansion des macros comme on l'a vu à la Section 3.2.3. Donc, on ne peut pas se fier à l'ordre des

déclarations dans le code source pour faire des recherches plus optimisées. Nos recherches de type sont donc plus difficiles et potentiellement plus lentes.

5.2.2. Solution proposée avec la fonction browse_lexp

browse_lexp est notre fonction qui fait la recherche du type de l'élément sur la souris ou le plus proche comme on l'a vu à la Figure 4.3. À cause du problème qu'on a mentionné avec les macros, la manière correcte est de traverser l'ensemble du code. On pourrait faire mieux en termes de recherche, comme ne considérer que la définition qui englobe la définition, puis traverser uniquement cette **lexp** pour retrouver la **sous-lexp** qui nous intéresse. Si on fait cela, ce sera presque toujours correct, mais on ne peut pas le garantir parce que l'expansion des macros peut fausser cette approche en plaçant les définitions dans le désordre par exemple. On pourrait également utiliser comme approche de traverser tout le code, et garder trace de toutes les positions traversées et des informations qui y sont, de manière à ce qu'à la deuxième recherche (et les recherches suivantes), on ait une information auxiliaire qui nous guide. Mais on ne procède pas de cette manière principalement parce qu'à chaque modification de l'état du code, on refait l'élaboration (par rapport à la structure du serveur) et qu'on n'a pas rencontré de réels enjeux demandant cette amélioration. Toutefois, cette piste est très prometteuse pour une future amélioration. Notre solution pour que ça marche toujours de manière correcte est donc de traverser tout l'arbre. Ce problème affecte aussi d'autres fonctionnalités. La liste exhaustive est : la *complétion*, la *définition* et les *breadcrumbs*. Ceci est parce que les informations nécessaires pour construire ces fonctionnalités sont fournies par une seule grande fonction, **browse_lexp**, qui renvoie un tuple avec les informations nécessaires pour chacune des fonctionnalités citées plus haut. Toutefois, on mentionne ce problème ici dans le *hover* parce que ce dernier en est plus affecté que les autres fonctionnalités. En effet, le *hover* affiche le type d'un élément, qui est, en soi, une **lexp** renvoyée par la fonction. Ce dernier demande plus de précisions que le contexte mis à jour qu'on utilise pour la complétion par exemple. Donc notre fonction fonctionne ainsi, elle traverse tout l'arbre et renvoie un tuple dans lequel se retrouve le type de l'élément le plus proche de la souris. Les autres éléments du tuple comme le contexte à un endroit précis du code (l'endroit où se positionne la souris) sont utilisés pour offrir d'autres fonctionnalités comme le saut à la définition d'un élément ou encore la complétion.

5.3. L'impression des types pour le Hover

Après l'élaboration, les types qu'on a à imprimer peuvent ne pas se retrouver dans le code source parce qu'ils sont utilisés de manière interne. Ici, on cherche donc à trouver une manière propre d'afficher les types.

5.3.1. Spécificité du problème

L'impression des types nous fournit un affichage laid et on aimerait ici apporter une amélioration à ce niveau. Les types qu'on doit imprimer peuvent ne pas apparaître dans le code source. L'élaboration peut retrouver le type grâce à une série de réductions pour inférer les types, mais la valeur renvoyée ne ressemble pas forcément à ce qu'on voudrait imprimer. Prenons comme exemple la fonction *trier* suivante qui fait un tri sur une liste d'entiers :

```
1  inserer : ? ;
2  inserer elem liste =
3      case liste
4      | nil => cons elem nil
5      | cons hd tl => if (Int_<= elem hd)
6                      then (cons elem liste)
7                      else (cons hd (inserer elem tl)) ;
8  trier : ? ;
9  trier lst =
10     case lst
11     | nil => nil
12     | cons hd tl => inserer hd (trier tl) ;
```

Listing 5.4. Constructeur Proj.

Comme on peut le remarquer, cette fonction travaille avec une liste d'entiers. Donc idéalement, on aimerait imprimer comme type de la liste *lst* **List Int** avec le *hover*. L'élaboration peut retrouver cette information, mais présentée différemment selon comment l'inférence a été faite. Pour notre cas à nous, elle nous renvoie comme type **(List ##Int)**. L'affichage peut être pire que ça. On peut par exemple avoir **List ##TypeLevel.z ##Int** ou encore **typecons (List) (nil) (cons ##Int (List ##TypeLevel.z ##Int))** qui correspond à "dérouler" la définition de **List** une fois. **##Int** est la valeur renvoyée par l'élaboration, car c'est la représentation du type des entiers. Ce type est, en soi, la valeur d'une variable nommée **Int** comme on l'a vu à la Section 1.2.5. L'utilisateur n'a pas besoin de voir les ##, parce que les réductions et représentations internes ne sont pas des informations nécessaires à l'utilisateur, et ce dernier veut juste voir quelque chose de simple et compréhensible. Mais, en soi, l'élaboration prend une **sexp** en paramètre et nous renvoie une **lexp** qui représente un type ressemblant au type précédent avec les ##. Ce qu'on cherche à imprimer comme forme propre et simplifiée est, en fait, une **sexp** dont l'élaboration est égale à la **lexp** de l'élaboration. Idéalement, c'est le but. Mais cette information n'est pas forcément disponible dans le code, et pas non plus après l'élaboration dans la forme souhaitée.

5.3.2. Solution proposée

L'élaboration nous renvoie une **lexp** à partir d'une **sexp**. Ce qu'on cherche comme forme propre est la **sexp** qui correspondrait à la **lexp** envoyé par l'élaboration. Donc, idéalement, il faudrait faire l'évaluation inverse parce que les types ne sont pas forcément affichés dans le code source, donc on ne peut pas les chercher là-bas. Dans notre exemple précédent, par exemple, à aucun endroit dans le code source, on a écrit **List Int**. Mais cette solution est en général impossible et demande beaucoup de travail qui sort un peu du cadre du projet, qui est d'implémenter un serveur LSP pour le langage Typer. Donc, on a résolu partiellement ce problème. Notre solution consiste à aller chercher dans le contexte une forme plus propre à imprimer. On avait par exemple le type d'un entier qui était **##Int** au lieu de **Int** tout simplement. En fait, **##Int** est la valeur de la variable **Int** résultante d'une suite de réductions. Ce qu'on a fait pour pallier ce problème, c'est de rechercher dans le contexte le nom d'une variable qui a la valeur correspondante au type qu'on a. En d'autres termes, on recherche une variable dans le contexte qui contient ce qu'on veut et si on en trouve une, on remplace le **##Int** par une référence à cette variable.

Si on retrouve la variable et qu'elle est redéfinie, c'est-à-dire, qu'une autre variable est définie avec le même nom plus tard dans le code, on utilise la nouvelle variable ainsi redéfinie. Ensuite, on a juste à utiliser cette variable enregistrée et sa position dans le contexte pour construire une forme plus propre à l'affichage de la variable, et utiliser ça comme type. Ceci permettra de faire passer les **##Int** à **Int** par exemple. Si on ne retrouve rien dans le contexte comme variable avec la valeur correspondante à la **lexp** que nous a renvoyé notre fonction **browse_lexp**, on imprime tout simplement la valeur de retour de **browse_lexp**, donc cette **lexp** telle quelle, et elle est, en soi, telle que résultante de l'élaboration. Notre solution est donc partielle.

Chapitre 6

Évaluation du serveur

Dans ce chapitre, nous parlons de l'évaluation du code et son efficacité. On a implémenté un système et on réunit ici les données des délais de réponses du serveur LSP selon la taille du fichier et les résultats si on fait les tests sur les différents fichiers échantillons présents dans Typer. Dans un premier temps, on fait des tests sur des fichiers qui ont le même code, mais dupliqué pour être de taille différente afin d'observer l'évolution de la vitesse par rapport à la taille du fichier. Pour ce faire, on a choisi de faire des tests sur des fichiers des tailles suivantes pour chaque fonctionnalité : 10KB, 30KB, 100KB, 300KB, 1MB. Grâce à l'évolution de la vitesse sur ces derniers, on peut avoir une idée de comment les délais évolueront par rapport à la taille du fichier. On s'attend à ce que l'évolution soit linéaire à la taille du fichier. Ce sont des fichiers de 480, 1450, 4800, 14520 et 48500 lignes respectivement pour avoir un ordre d'idées. Pour faire du code qui est le même en complexité, mais de taille différente, on a fait des duplications du même fichier. Il s'agit du fichier *math.typer* (un fichier de 5.1KB) dupliqué pour atteindre les tailles souhaitées (ce fichier se trouve dans le dossier "samples" du projet Typer sur Gitlab avec le numéro de révision : ffd27741d831). Et dans un second lieu, on fait les tests sur les échantillons (les fichiers se trouvant dans le dossier "samples" du projet Typer sur Gitlab également) pour voir dans quelle mesure la vitesse dépend non seulement de la taille du code, mais aussi du code lui-même. On veut également voir si la vitesse est acceptable pour des fichiers plus communs que des fichiers dont le contenu est dupliqué. Les fichiers échantillons et leurs tailles sont répertoriés dans le Tableau 6.1. On attend d'un serveur de langue comme le nôtre qu'il fournisse des informations au client (notification ou réponse à une requête) dans une **centaine de millisecondes**. On va utiliser la valeur de **100 millisecondes** comme repère. Chaque mesure observable est la moyenne de 1000 mesures calculées par des scripts automatiques pour avoir des données justes et acceptables. Pour chaque mesure, on ne fait pas l'élaboration à nouveau. C'est-à-dire qu'on fait l'élaboration une fois et on commence à compter à partir du résultat de l'élaboration pour faire chaque mesure. On fait ça à cause de la façon dont fonctionne notre serveur LSP.

Nom	taille
pervasive.typer	0.417KB
plain_let_test.typer	0.523KB
acc.typer	0.748KB
dependent.typer	0.85KB
nat.typer	0.857KB
myers.typer	0.996KB
decltype.typer	1.1KB
do_test.typer	1.2KB
bool.typer	1.4KB
defmacro	1.4KB
tuple_test.typer	1.6KB
batch_test.typer	1.7KB
array_test.typer	2.2KB
hurkens.typer	2.2KB
bbst.typer	2.5KB
hott.typer	2.9KB
polyfun_test.typer	2.9KB
autodiff.typer	3.5KB
math.typer	5.1KB
elabctx_test.typer	6.8KB
table.typer	7.9KB
case_test.typer	11.3KB

Tableau 6.1. Noms et tailles des échantillons

En effet, ce dernier élabore le code une fois, et utilise le résultat de l'élaboration pour fournir toutes les fonctionnalités à un état précis du code. Et quand le code est modifié, l'élaboration se fait à nouveau et toutes les fonctionnalités sont fournies à ce nouvel état du code. Donc dans un premier temps, on fait toutes nos mesures sans l'élaboration, et puis dans un second temps, on mesure indépendamment l'élaboration parce qu'elle fait partie du processus et que l'utilisateur de notre serveur LSP y sera confronté. On a également considéré le temps d'envoi des informations au client négligeable. En effet, une fois que les informations sont là, la transition de ces informations au client est presque immédiate. Et étant donné que le client peut différer, et donc que le temps peut légèrement varier d'un client à un autre, on a décidé d'exclure le client parce que c'est hors de notre contrôle. Nos mesures concernent donc le temps que prend notre serveur LSP à renvoyer ce qu'il faut passer comme informations au client. Par exemple, pour la complétion, ces mesures concernent le temps que le serveur met pour trouver toutes les complétions possibles sans l'envoi de ces complétions au client. Ces mesures ont été effectuées avec un ordinateur fixe pour la stabilité de leurs processeurs sur le serveur 'baro'. *Baro* est un serveur avec 48GB de RAM et un processeur Xeon X5650 de fréquence 2.67GHz avec 12 cœurs, mais notre code n'a utilisé qu'un cœur.

6.1. Évaluation avec des fichiers de tailles différentes et les échantillons

Dans cette première section, on va voir les résultats avec des fichiers de même contenu, mais de tailles différentes. On a choisi des tailles grandissantes pour voir comment ça se comporte avec le même code en complexité, mais que la taille est progressive. L'objectif est de savoir si on est linéaire à la taille du fichier et aussi si on n'a pas de surprise de dysfonctionnement.

6.1.1. Les diagnostics

Les mesures ont été faites avec des scripts pour automatiser le processus, mais aussi, pour plus d'exactitude. Chaque mesure est la moyenne de 1000 mesures de la même chose, et c'est seulement cette moyenne-là qui apparaît comme résultat pour chaque point de chaque figure.

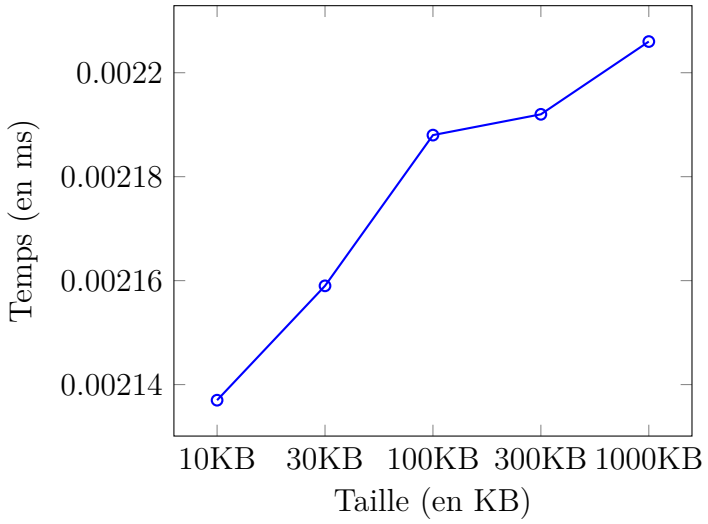

Fig. 6.1. Les fichiers de taille évolutive pour les diagnostics

On peut remarquer à la Figure 6.1 que le temps par rapport à la taille du fichier est essentiellement constant. Ce qui est ce qu'on attendait. Pour ce qui est des résultats des tests des échantillons (Figure 6.2), on a atteint un maximum de **0.00135 millisecondes**. Pour le fichier d'*1MB*, on a un temps de **0.002206 millisecondes** en moyenne. Donc les résultats sont largement supérieurs à ce qu'il faut pour un serveur stable.

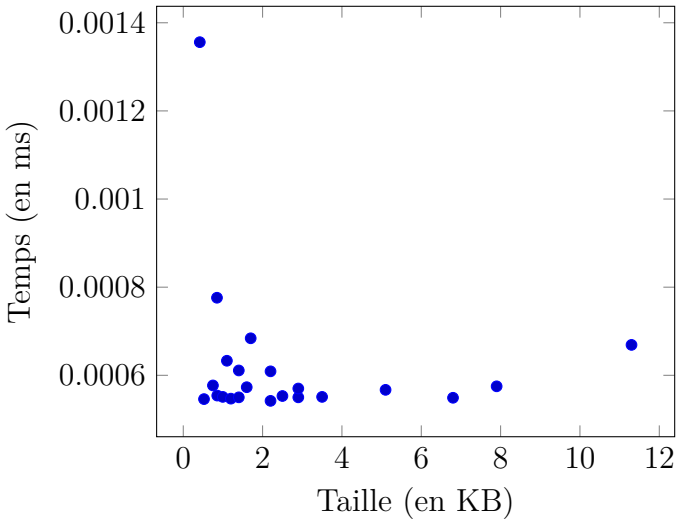

Fig. 6.2. Les échantillons pour les diagnostics

6.1.2. Le hover

Pour faire les mesures pour le *hover*, on fait des mesures sur 1000 tests afin de calculer la moyenne pour avoir des résultats plus précis comme pour les diagnostics.

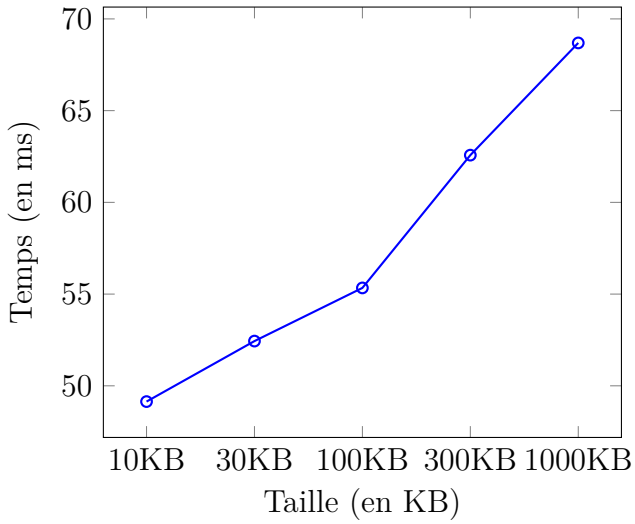

Fig. 6.3. Les fichiers de taille évolutive pour le hover

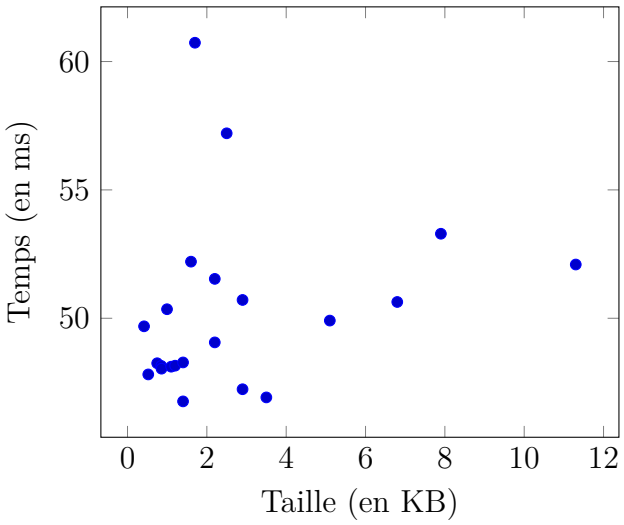

Fig. 6.4. Les échantillons pour le hover

La Figure 6.3 montre une progression linéaire à la taille du fichier de la vitesse. Les mesures pour le *hover* conviennent à nos exigences. D'autre part, pour ce qui est de la Figure 6.4, on atteint un maximum de **60 millisecondes**. Pour le grand fichier d'*1MB*, on a une moyenne de **69 millisecondes**. On répond donc aussi aux exigences en termes de vitesse. Toutefois, beaucoup de temps est mis dans la recherche d'une manière plus propre d'imprimer le type. Une solution comme celle proposée à la Section 5.2.2 qui propose de garder trace de toutes les positions traversées dans la recherche d'un nœud de l'arbre de syntaxe abstraite pourrait profiter à cette fonctionnalité en vitesse.

6.1.3. La complétion

Pour la complétion également, on part sur les mêmes fichiers ainsi que les mêmes standards.

On peut observer à la Figure 6.5 que le délai de réponse de la complétion évolue par rapport à la taille du fichier. On a encore une fois ce qu'on attendait comme résultat. Pour ce qui est de la Figure 6.6, on remarque que la complétion varie très peu. On a atteint comme valeur de délai maximal de réponse **2.822 millisecondes**. On a une moyenne de **18 millisecondes** pour les grands fichiers d'*1MB*. Ce qui nous indique une performance satisfaisante en termes de vitesse.

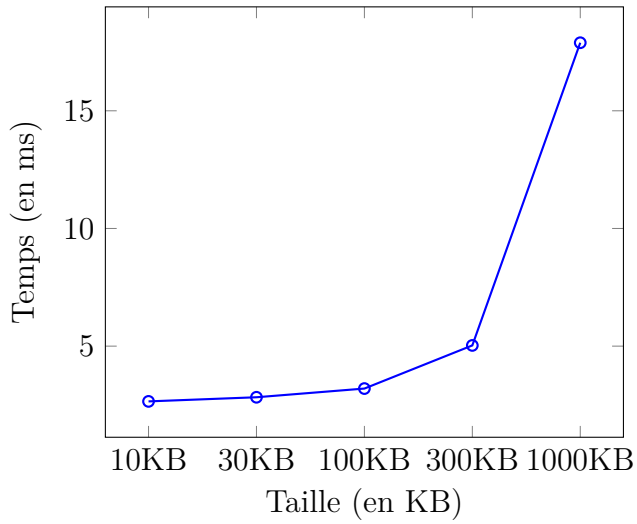

Fig. 6.5. Les fichiers de taille évolutive pour la complétion

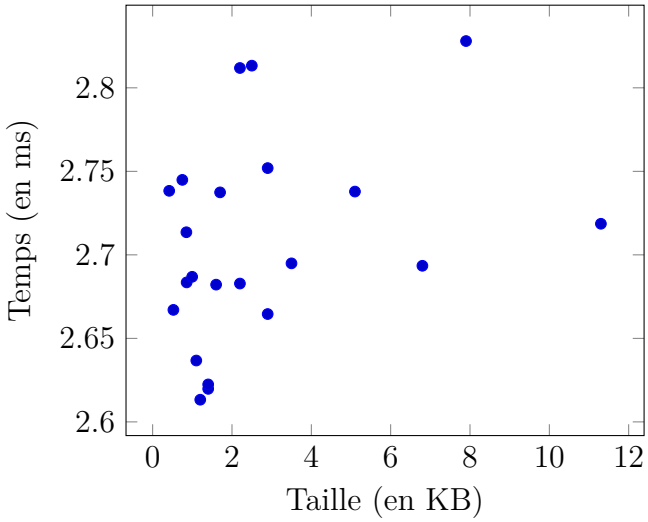

Fig. 6.6. Les échantillons pour la complétion

6.1.4. Le saut à la définition

Pour le saut à la définition aussi, on part sur les mêmes principes.

On peut interpréter à la Figure 6.7 que le serveur est linéaire à la taille du fichier. Donc, on est satisfait pour ce côté-là. D'un autre côté, pour l'évaluation des échantillons à la Figure 6.7, la valeur maximale atteinte est **0.304 millisecondes**. Pour les grands fichiers avoisinant le méga-octet, on a une moyenne de **2.5 millisecondes**. Là encore, notre serveur est assez rapide et on n'a pas de surprise.

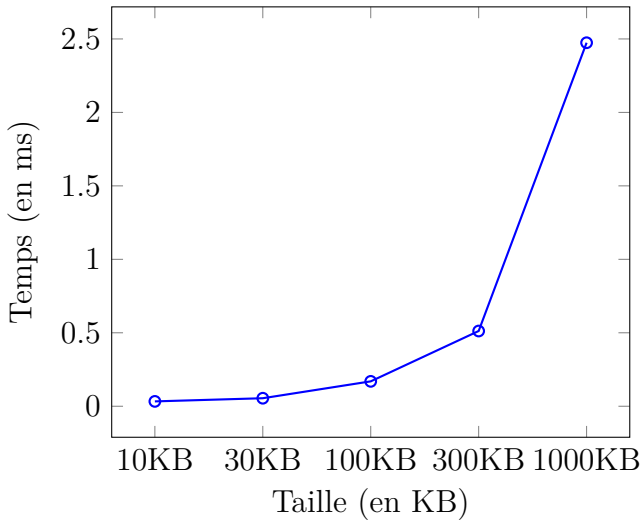

Fig. 6.7. Les fichiers de taille évolutive pour le saut à la définition

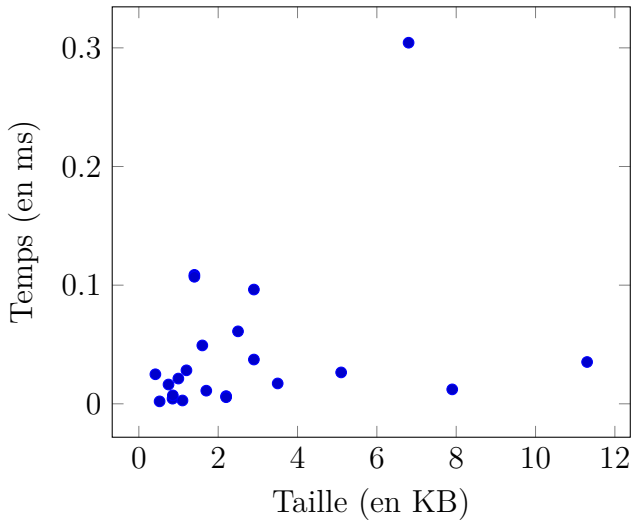

Fig. 6.8. Les échantillons pour le saut à la définition

6.1.5. Les highlights et les références

On a tout aussi mesuré les *highlights* et les *références* comme tous les autres. On a joint ces deux fonctionnalités étant donné qu'elles demandent, dans notre cas, les mêmes informations. Les informations qu'elles envoient au client sont les mêmes et prennent par conséquent le même temps.

La Figure 6.9 montre une évolution de la vitesse linéaire à la taille du fichier. On a

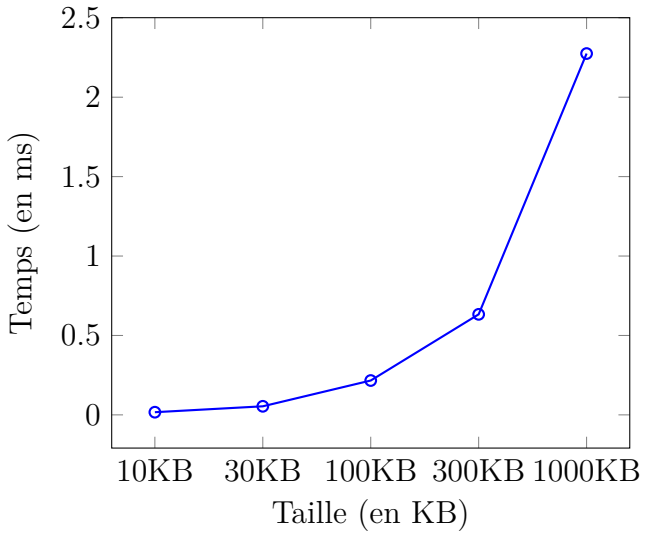

Fig. 6.9. Les fichiers de taille évolutive pour les highlights et références

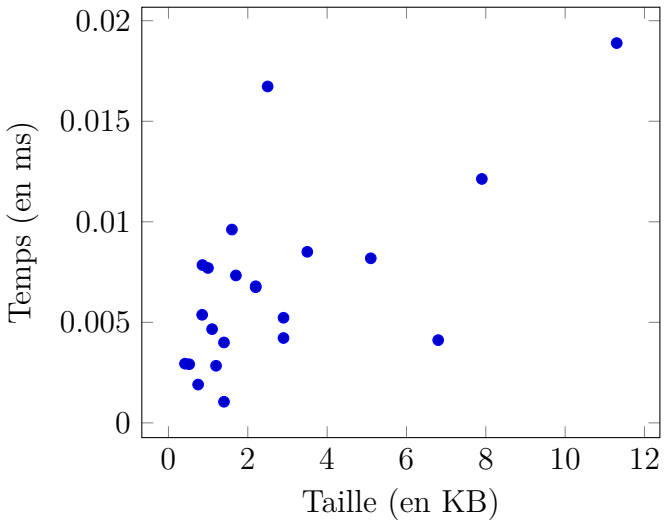

Fig. 6.10. Les échantillons pour les highlights et références

également un maximum de **0.018 millisecondes** dans l'évaluation des échantillons. Et aussi **2.3 millisecondes** comme moyenne pour les délais de réponse des fichiers d'une taille avoisinant *1MB*. On a donc une vitesse acceptable.

6.1.6. Les breadcrumbs

Pour finir, avec les breadcrumbs, c'est également avec les mêmes standards que les mesures se sont faites. On se réfère toujours à notre valeur de *100 millisecondes* comme repère.

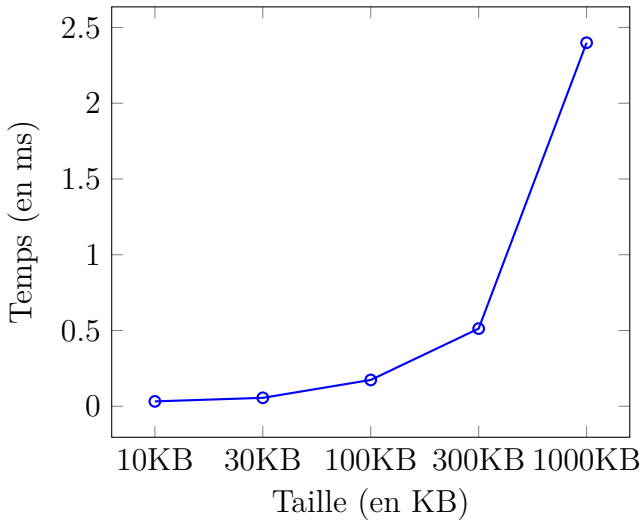

Fig. 6.11. Les fichiers de taille évolutive pour les breadcrumbs

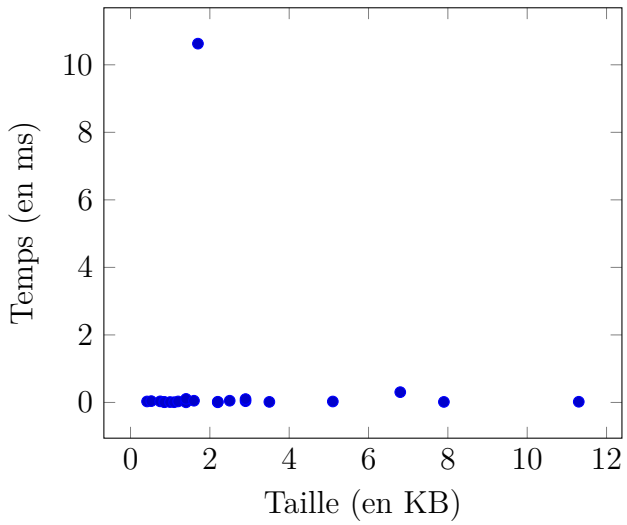

Fig. 6.12. Les échantillons pour les breadcrumbs

On remarque à la Figure 6.11 que pour les *breadcrumbs*, la vitesse régresse en fonction de l'augmentation de la taille du fichier. Ceci est donc comme on l'attendait. Pour la Figure 6.12, on peut remarquer que la plus grande valeur qu'on a atteinte pour le temps mis est de **2.791 millisecondes**. En ce qui concerne les fichiers de l'ordre d'*1MB*, on a une moyenne de temps de **2.4 millisecondes**. Pour le côté de la vitesse, la performance est convenable.

6.2. Le temps de l'élaboration

Le temps de l'élaboration ne relève pas de notre responsabilité directe étant donné qu'on n'a pas implémenté l'élaborateur dans le cadre de ce projet. Mais étant donné que l'utilisateur y sera confronté dans l'utilisation du serveur, ces données sont importantes à mesurer. On réunit donc les données de l'élaboration de chacun des fichiers qu'on a utilisés dans la Table 6.2. On part toujours sur le principe de la moyenne de 1000 mesures pour chaque résultat pour plus d'exactitude. On peut tirer, à travers les données qu'on a, la conclusion que l'élaboration est assez rapide pour ne pas avoir d'impact négatif significatif sur notre serveur.

Nom (taille)	temps d'élaboration (en ms)
pervasive.typer (0.417KB)	0.068624
plain_let_test.typer (0.523KB)	0.07408
acc.typer (0.748KB)	0.070884
dependent.typer (0.85KB)	0.076823
nat.typer (0.857KB)	0.069143
myers.typer (0.996KB)	0.116017
decltype.typer (1.1KB)	0.074137
do_test.typer (1.2KB)	0.067045
bool.typer (1.4KB)	0.076173
defmacro.typer (1.4KB)	0.066815
tuple_test.typer (1.6KB)	0.086057
batch_test.typer (1.7KB)	0.495689
array_test.typer (2.2KB)	0.100345
hurkens.typer (2.2KB)	0.088934
bbst.typer (2.5KB)	0.510666
hott.typer (2.9KB)	0.063731
polyfun_test.typer (2.9KB)	0.101832
autodiff.typer (3.5KB)	0.071897
math.typer (5.1KB)	0.092778
elabctx_test.typer (6.8KB)	0.093219
table.typer (7.9KB)	0.133177
case_test.typer (11.3KB)	0.260191
10KB.typer (10KB)	0.103934
100KB.typer (100KB)	0.496741
1MB.typer (1MB)	4.694625

Tableau 6.2. Noms et temps d'élaboration de tous nos fichiers

6.3. Interprétation des résultats

Pour qu'un serveur de langue fonctionne bien, il faut qu'il puisse répondre aux requêtes dans une moyenne de **100 millisecondes**. Notre serveur est globalement rapide. Pour

l'envoi des diagnostics, notre serveur est constant et prend une fraction de millisecondes. Ceci nous convient parfaitement. En ce qui concerne le *Hover*, on reste dans nos exigences de répondre aux requêtes en moins de **100 millisecondes**, mais c'est notre fonctionnalité qui prend le plus de temps. En fait, retrouver l'information de type est très rapide (le type de l'expression représentée par un nœud de l'arbre de syntaxe abstraite), mais le temps est essentiellement passé à améliorer l'affichage des types dans des formes plus agréables et plus facilement compréhensibles par l'utilisateur. Une amélioration de l'affichage des types en Typer pourrait faire totalement disparaître ce problème et accélérer nettement le serveur. Une autre manière d'améliorer cette fonctionnalité serait d'ajouter des caches, c'est-à-dire de garder trace de l'information présente dans les positions traversées. Ceci pourrait améliorer cette fonctionnalité en termes de vitesse, parce que dans une deuxième recherche, les informations précédentes pourraient nous servir de guide et beaucoup nous faire gagner en vitesse. Les résultats nous montrent toutefois qu'on n'a pas besoin de rajouter des caches parce que la vitesse est convenable, mais c'est une possibilité envisageable pour une amélioration. La préoccupation de la vitesse du serveur était majeure étant donné que, à cause de l'expansion des macros du langage, on est obligé de traverser tout le code, ce qui, en théorie, peut prendre du temps. Mais les résultats de l'expérience sont prometteurs en termes de vitesse, et le serveur reste rapide même à traversant tout le code, et ceci, même pour les fichiers de grande taille. Pour les autres fonctionnalités comme la *complétion*, les *références* ou encore le saut à la définition, leurs vitesses sont convenables et peuvent assurer un bon fonctionnement et une stabilité du serveur.

Conclusion

Notre objectif principal était d'offrir un serveur LSP au langage Typer pour aider les programmeurs. Et pour y arriver, il faut communiquer continuellement avec le compilateur pour avoir les informations nécessaires à transmettre au client. L'implémentation du langage en lui-même n'a pas été forcément conçue pour accueillir un serveur LSP, donc on a fait beaucoup de travaux d'adaptation à ce niveau pour que le serveur fonctionne, des adaptations comme pour l'affichage du type de l'élément sur lequel se place la souris pour afficher les types de manière élégante. Les macros du langage ont occasionné une traversée complète de l'arbre de syntaxe abstraite dans la recherche d'un nœud pour trouver le type de l'élément sur lequel la souris se positionne, étant donné que ces derniers réordonnent le code de manière arbitraire. La perte de certaines informations nécessaires pour construire la fonctionnalité de *complétion* pendant la phase d'élaboration nous a amené à étendre la définition d'une lambda expression, le cœur du langage, avec un nouveau constructeur pour garder ces informations et aussi améliorer le code pour en avoir un qui est plus rapide et plus efficace. Notre travail apporte des solutions partielles ou totales à tous ces problèmes. Le langage Typer a donc désormais un serveur LSP fonctionnel. L'évaluation du serveur en termes de vitesse nous a aussi donné des résultats prometteurs. On espère que ce serveur va servir les programmeurs Typer pour coder plus rapidement et efficacement dans leurs éditeurs/IDE préférés. On souhaite que notre outil aide les programmeurs Typer à coder avec moins d'effort pour plus de productivité et d'efficacité. Des pistes comme garder trace des positions traversées dans l'arbre de syntaxe abstraite dans la recherche d'un nœud peuvent être explorées pour améliorer la vitesse globale du serveur.

Références bibliographiques

[1] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand et H. Abelson : Revised^5 report on the algorithmic language Scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

[2] Hejlsberg Anders : Introducing TypeScript. *Microsoft Channel*, 9, 2012.

[3] Frédéric Bour, Thomas Refis et Gabriel Scherer : Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–15, 2018.

[4] Ana Bove, Peter Dybjer et Ulf Norell : A brief overview of agda–a functional language with dependent types. *In International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[5] Max Brunsfeld : Tree-sitter. `https://github.com/tree-sitter/tree-sitter`, 2019.

[6] Flanagan David et Matilainen Pasi : *JavaScript*. Anaya Multimedia, 2007.

[7] Pierre Delaunay : Implémentation d'un langage fonctionnel orienté vers la méta programmation. Mémoire de D.E.A., Université de Montréal, 2017.

[8] Crockford Douglas : JSON. *ECMA International*, 2012.

[9] Brady Edwin : Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[10] Brady Edwin : *Type-driven development with Idris*. Simon and Schuster, 2017.

[11] Huet Gérard, Kahn Gilles et Paulin-Mohring Christine : The Coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[12] JSON-RPC Working Group *et al.* : JSON-RPC 2.0 specification. *Online]: https://www. jsonrpc. org/specification (Accessed 2021-08-22)*, 2013.

[13] Mehnert Hannes et Christiansen David : Tool demonstration: An IDE for programming and proving in Idris. *Proceedings of Vienna Summer of Logic, VSL*, 14(2), 2014.

[14] Geuvers Herman : Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[15] McCarthy John, Abrahams Paul W, Edwards Daniel J, Hart Timothy P et Levin Michael I : *LISP 1.5 programmer's manual*. MIT press, 1962.

[16] Levine, John R, Mason ans John, Levine, John R, Mason, Tony, Brown, Doug, Levine, John R, Levine et Paul : *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

[17] Flatt Matthew et Findler Robert Bruce : Racket guide. `https://download.racket-lang.org/releases/7.6/pdf-doc/guide.pdf`, 2014.

[18] OCamlPro : The opam manual. `https://opam.ocaml.org/doc/Manual.html`, 2013.

[19] Kipps James R : Glr parsing in time o (n 3). *In Generalized LR parsing*, pages 43–59. Springer, 1991.

[20] François Pottier Yann Régis-Gianas : Menhir reference manual. 2016.

[21] Harper Robert, MacQueen David et Milner Robin : *Standard ML*. Department of Computer Science, University of Edinburgh, 1986.

[22] Milner Robin : A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[23] Culpepper Ryan : Fortifying macros. *Journal of functional programming*, 22(4-5):439–476, 2012.

[24] Culpepper Ryan et Felleisen Matthias : Taming macros. *In International Conference on Generative Programming and Component Engineering*, pages 225–243. Springer, 2004.

[25] Joshua B Smith : Ocamllex and ocamlyacc. *Practical OCaml*, pages 193–211, 2007.

[26] Monnier Stefan : Typer: ML boosted with type theory and Scheme. *Journées Francophones des Langages Applicatifs*, pages 193–208, 2019.

[27] Leroy Xavier et Weis Pierre : *Manuel de référence du langage Caml*. InterEditions, 1993.

[28] Minsky Yaron, Madhavapeddy Anil et Hickey Jason : *Real World OCaml: Functional Programming for the Masses*. " O'Reilly Media, Inc.", 2013.