Rapport de stage à Deducteam (ENSIIE 1A) “Interface moderne pour la preuve interactive de théorèmes”

Francois Lefoulon

▶ **To cite this version:**

Francois Lefoulon. Rapport de stage à Deducteam (ENSIIE 1A) “Interface moderne pour la preuve interactive de théorèmes”. Informatique [cs]. 2020. hal-02971929

HAL Id: hal-02971929
https://inria.hal.science/hal-02971929

Submitted on 7 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École nationale supérieure d'informatique pour l'industrie et l'entreprise

Rapport de stage 1A

«Interface moderne pour la preuve interactive de théorèmes»

Élève : **François Lefoulon**
Tuteur académique à l'ENSIIE : **Guillaume Burel**

Laboratoire : **Deducteam** (Gif-sur-Yvette, 91)
Maîtres de stage : **Frédéric Blanqui, Emilio Gallego**

Stage 1A du 1er juin au 31 juillet

Remerciements

Merci à Tristan qui m'a fait découvrir Deducteam et donc permis d'obtenir ce stage.

Merci à Amélie qui m'a beaucoup aidé à comprendre les concepts et la syntaxe de LambdaPi et Coq, tout en me montrant CoqIDE.

Merci à Frédéric pour ses recommandations et conseils toutes les réunions, et à Emilio pour son aide précieuse sur certaines portions ardues du code OCaml.

Merci à toute l'équipe pour les séminaires extrêmement riches.

Deducteam

FIGURE – *Le logo de Deducteam*

Deducteam est une équipe de l'Institut National de Recherche en Informatique et en Automatique (Inria), dont les travaux de recherche portent sur les prouveurs automatiques et les assistants de preuve.

Elle est située au Laboratoire de Spécification et Vérification (LSV) de l'ENS Paris-Saclay, et dirigée par Gilles Dowek. Elle comporte actuellement 6 membres permanents, 5 post-doctorants et ingénieurs, ainsi que 6 étudiants en doctorat. 8 stagiaires étaient présents cet été.

Un projet central de l'équipe est la création d'un hub pour les assistants de preuve, avec le logiciel Dedukti et la librairie de preuves Logipedia, ainsi que de nombreux logiciels de conversion de preuves entre Dedukti et des assistants de preuves.

Liste des sigles et abréviations

Voici les sigles et abréviations utilisés dans ce rapport :

* **LP** : LambdaPi, aussi appelé Dedukti 3
* **LSP** : Language Server Protocol
* **IDE** : Integrated Development Environment
* **API** : Application Programming Interface
* **VSCode** : Visual Studio Code

Table des matières

1 Introduction **1**

2 Contexte du stage **2**
2.1 Cadre de travail 2
2.2 Logique et preuve informatisée 2
2.3 Travaux de l'équipe, langage Dedukti 2

3 Définition des objectifs et solutions **4**
3.1 Approche du problème : développement d'une extension pour un éditeur de texte 4
3.2 Modèle client-serveur 4
3.3 Implémentation existante 5
3.4 Features souhaitées 5
3.5 Features additionelles 6

4 Extension Visual Studio Code **7**
4.1 Utilisation du VSCode Extension API 7
4.1.1 Fonctionnement général 7
4.1.2 Syntax highlighting 7
4.1.3 Snippets 9
4.1.4 Commandes 9
4.2 Implémentation des commandes et du client 10
4.2.1 En place à mon arrivée 10
4.2.2 Mes ajouts 10

5 Serveur LambdaPi **12**
5.1 Implémentation existante du protocole dans le serveur LP 12
5.2 Traitements des requêtes "hover" et "definition" 12
5.2.1 Précision des problèmes rencontrés 12
5.2.2 Récupération des informations issues du parsing 13
5.2.3 Range map pour les tokens 13

6 Conclusion **15**

Appendices

A Exemple : message JSON-RPC envoyé du client au serveur **i**

B Exemple : coloration syntaxique **ii**

C Exemple : snippet **iii**

D Exemple de navigation de preuve **iv**

E Exemple : hover et definition **v**

F Annexe : Développement durable et responsabilité sociétale **vi**

G Références **vii**

1. Introduction

Le domaine de la logique et de la preuve assistée par ordinateur requiert une expertise conséquente pour formaliser des preuves mathématiques, ou des preuves de correction de logiciel, en un langage compréhensible par une machine. Alléger cette difficulté est un moyen de faciliter le développement de logiciels sûrs, dont les enjeux sont critiques dans l'industrie des transports notamment, et la recherche en mathématiques, dont les applications sont très nombreuses.

L'objet du stage est de fournir une interface graphique agréable et pratique pour Dedukti 3, qui a l'ambition de centraliser les preuves informatiques.

2. Contexte du stage

2.1 Cadre de travail

En raison de l'épidémie de COVID-19, le stage se déroule en télétravail dans mon appartement étudiant.

Je suis encadré par Frédéric Blanqui, membre permanent de Deducteam, et Emilio Gallego, chercheur de l'équipe πr^2, travaillant également sur Dedukti 3. Nous nous réunissons sur Skype tous les mercredi. J'envoie toutes les semaines un compte-rendu de mon travail.

Nous assistons tous les jeudis à des séminaires en ligne présentés par des membres de l'équipe, dans le but d'élargir nos connaissances à partir des travaux d'un membre de l'équipe.

Une unique réunion d'équipe en présentiel est organisée à l'ENS Paris-Saclay pour que chaque stagiaire fasse une soutenance de stage blanche.

2.2 Logique et preuve informatisée

Une logique repose sur la définition de règles de déduction qui à partir d'axiomes permettent de déduire un ensemble de propositions. L'exemple canonique est la logique classique utilisée en mathématiques. Il existe d'autre logiques comme la logique intuitioniste, où les propositions ne sont pas nécessairement vraies ou fausses.

Les preuves informatisées utilisent des représentations machine des axiomes, propositions et règles ; elles nécessitent un formalisme plus précis qu'une preuve en langage naturel, qui leur donne un haut niveau de fiabilité.

Les prouveurs automatiques font une grande partie du travail de vérification (d'un logiciel le plus souvent) seuls, tandis que les assistants de preuve se destinent à un utilisateur qui souhaite écrire des preuves. Un bon assistant de preuve fournit des outils à l'utilisateur pour rendre l'écriture de preuves plus facile.

2.3 Travaux de l'équipe, langage Dedukti

L'équipe Deducteam développe depuis 2011 un langage permettant d'implémenter de nombreuses logiques, et ainsi de vérifier des preuves dans une logique voulue. C'est ce qu'on appelle un logical framework.

Le langage se nomme Dedukti, et est fondé sur la théorie du $\lambda\Pi$-calcul modulo réécriture. Intuitivement, écrire une preuve consiste à décomposer un théorème en une arborescence de propositions, jusqu'à retomber sur des propositions et définitions connues. On peut en sus considérer que les objets mathématiques utilisés dans la preuve sont des symboles qu'il faut réécrire, d'où l'application de certaines tactiques appelées règles de réécriture. Ces règles sont analogues à des fonctions, qui prennent en entrée un symbole d'un certain type et renvoient un autre symbole d'un certain type en sortie. Elles sont particulièrement utiles pour définir des théories équationelles ou calculatoires, puisqu'elles sont appliquées automatiquement par l'ordinateur pour effectuer des calculs et tester des égalités.

Dedukti permet de définir des symboles avec un certain type, ainsi que des règles de réécriture. À chaque étape d'une preuve Dedukti, on applique une certaine tactique (par exemple une règle de réécriture) ; un fichier LP est valide s'il "type-check", c'est à dire si les types des symboles restent cohérents tout au long du programme et qu'il n'y a pas d'erreur de syntaxe. Comme tout langage, LP dispose d'un parser, c'est à dire un programme qui analyse la syntaxe afin d'identifier les symboles, définitions, types...

Jusqu'à la version 2, Dedukti ne permet pas de visualiser l'état d'avancement de la preuve lors de l'application d'une tactique, ce qui rend complexe l'écriture de preuves.

La version 3 de Dedukti, nommée LambdaPi (ou LP), s'accompagne d'une refonte plus simple de la syntaxe, mais surtout de l'ajout de fonctionnalités d'assistant de preuve au langage : à chaque étape de la preuve, un parser LambdaPi identifie un certain but dans le raisonnement. L'idée pour permettre à l'utilisateur de construire une preuve est d'afficher ces buts au fur et à mesure qu'il avance dans l'écriture de cette preuve.

L'objectif de mon stage est de créer une interface graphique pour l'utilisation de LambdaPi en tant qu'assistant de preuve.

3. Définition des objectifs et solutions

3.1 Approche du problème : développement d'une extension pour un éditeur de texte

Il existe déjà des assistants de preuve avec une interface ergonomique et un affichage progressif des buts. CoqIDE propose cela pour le langage Coq, tandis que ProofGeneral, une extension emacs, se veut adaptable à tout langage d'assistant de preuve ; cependant, cette adaptation demanderait un travail considérable puisqu'il n'y a pas de manière simple d'utiliser, avec ProofGeneral, le Language Server Protocol, que je présenterai et dont j'aurai besoin dans la suite.

L'idée serait plutôt de créer une telle interface au sein de différents éditeurs de texte couramment utilisés aujourd'hui, à l'instar d'Atom, Vim, Emacs ou VSCode, au travers d'extensions dédiées. Cette méthode est plus attractive pour les utilisateurs, qui souhaitent en général disposer d'un unique éditeur de texte pour toutes leurs tâches de programmation.

Une interface sommaire a été développée pour Vim. Une extension pour Atom était en projet au sein de l'Inria, mais a été abandonnée au profit d'Emacs et VSCode, pour lesquels il est plus simple de développer l'extension voulue. Durant ce stage, un collègue ingénieur travaille sur une extension Emacs tandis que je m'occupe d'une extension VSCode.

Cet éditeur dispose en effet d'une riche API permettant le développement d'extensions permettant de modifier l'affichage, d'exécuter des commandes...

Maintenant, sachant qu'il faut implémenter notamment, dans ces extensions, le parsing et le type-checking du texte (2.3), est-il vraiment judicieux de coder un parser dans l'extension VSCode ? Ne pourrait-on pas réutiliser le parser de LambdaPi, et factoriser du code qui pourrait servir à différentes extensions ?

3.2 Modèle client-serveur

La solution proposée par mes maîtres de stage est de considérer l'éditeur de texte comme une application "client", qui envoie des requêtes à une application "serveur" utilisant le parser de LambdaPi. Le problème est alors de permettre au client et au serveur de communiquer.

C'est à ce problème que répond le LSP (Language Server Protocol), lancé par Microsoft en 2016. Il s'agit d'un protocole de communication entre un "language client" et un "language

server" basé sur un autre protocole très simple, JSON-RPC.

D'après la spécification JSON-RPC, la syntaxe des requêtes du client et des réponses du serveur est la suivante : un message est en fait un objet JSON, donc une liste d'attributs entre crochets, dont les attributs sont

* jsonrpc : la version du protocole json-rpc, peu important ici
* method : l'objet de la requête/réponse, ce qu'elle doit permettre de faire
* params : les paramètres de la requête
* id : l'identifiant de la requête

Exemple en annexe. A

Le rôle du Language Server Protocol est principalement de fournir une liste de méthodes (attribut "method") spécifiques à une interface, ou un IDE, pour un langage de programmation, où l'utilisateur code à l'aide d'une interface (un éditeur de texte, par exemple) qui est le client du protocole ; le client demande des informations au serveur et les rend visibles par l'utilisateur.

3.3 Implémentation existante

À mon arrivée sur sur le stage, il existait déjà une extension "client" VSCode et un language server.

Ceux-ci utilisaient (et utilisent encore) la méthode "textDocument/publishDiagnostics" afin de vérifier le type checking du texte et générer un outline.

Une utilisation approximative des méthodes "hover" et "definition" permettait de voir le type d'un symbole LP ou d'aller à sa définition, uniquement si le type et la définition étaient fournis dans le même fichier et non dans un fichier importé à l'aide du mot-clé LP "require open". Ces commandes ne fonctionnaient pas toujours et l'éditeur affichait un "Loading..." au-dessus d'un mot-clé dès que la souris le survolait et que le serveur ne pouvait en fournir le type.

Une méthode getGoals (inchangé durant mon stage) était implémentée pour récupérer les buts d'une preuve en cours. Elle était appelée dès que l'utilisateur positionnait son curseur sur une ligne du texte.

Le texte faisait l'objet d'une coloration syntaxique binaire : les mots-clés LP étaient d'une couleur, le reste du texte n'était pas coloré.

Il n'existait pas de manière intuitive de taper des caractères unicode, qui font pourtant partie des incitatifs à utiliser LP et de sa syntaxe de base. En effet, LP se veut plus proche de l'écriture mathématique que les autres langages (de preuve ou de programmation), et les lettres grecques, par exemple, peuvent être écrites telles quelles et comprises par le parser.

3.4 Features souhaitées

* Dissocier la position du curseur et l'affichage des buts
* Fournir différents keybindings pour naviguer dans une preuve
* Colorer les éléments de preuve déjà parcourus

* Améliorer la coloration syntaxique
* Réparer les hover et definition
* Permettre de taper facilement des caractères unicode

3.5 Features additionelles

J'ai décidé de conserver la navigation de la preuve liée à la position du curseur, mais de la désactiver par défaut. Pour plus de confort visuel, la fenêtre VSCode se positionne automatiquement pour centrer l'état actuel de la preuve.

4. Extension Visual Studio Code

4.1 Utilisation du VSCode Extension API

4.1.1 Fonctionnement général

Les extensions VSCode fonctionnent à l'aide du langage TypeScript et des objets JSON.

Le fichier central dans une extension VSCode est le "extension manifest", un fichier intitulé "package.json". Il s'agit d'un objet JSON dont les attributs déterminent le nom, les paramètres et le fonctionnement de l'extension.

Par exemple,

* l'attribut "name" contient le nom de l'extension, "vscode-lp"
* l'attribut "contribute" est un tableau qui contient la plupart des éléments importants de l'extension, par exemple un attribut "main" qui est le nom du fichier exécuté au démarrage de l'extension. Ici, ce fichier est "./out/src/client.js", où le dossier courant "." est la racine de l'extension, où se trouve en fait le manifest.

Ce fichier est traité par le VSCode extension API pour créer et exécuter l'extension. Le fichier main, appelé client.js, est également central puisqu'il contient le code de toutes les commandes de l'extension. J'écris ce main en langage Typescript dans src/client.ts, qui est ensuite compilé en fichier JavaScript. Étant donnée la multitude de types et de classes utilisés par l'API VS-Code, c'est presque une nécessité d'utiliser TypeScript pour comprendre le code et éviter les erreurs.

L'ajout des différentes features se fait donc par la méthode suivante :

1. Référencer une feature dans le bon attribut de l'extension manifest, souvent le nom d'un fichier annexe
2. Écrire le fichier/les lignes de codes correspondant à cette feature

Un exemple sera donné en annexe pour chaque feature.

4.1.2 Syntax highlighting

Il y a plusieurs manières de définir une coloration syntaxique avec le VSCode extension API :

* à l'aide d'une requête au language server

* à l'aide de l'API Decorations et d'un thème
* à l'aide de grammaires TextMate et d'un thème.

Ces méthodes ne sont pas incompatibles. Leur principe est de :

* demander au serveur comment colorer une portion de texte, et le serveur envoie une couleur en réponse.
* déterminer au sein du "main" de l'extension des portions de texte à décorer.
* déterminer des "scopes", c'est à dire des portions de texte définies par une expression régulière, les ranger dans une certaine catégorie (fonctions, importations, warnings...), puis laisser le thème de VSCode les colorer.

Les deux premières méthodes sont assez lourdes, puisqu'il faut écrire des programmes parcourant le texte et récupérant l'intervalle de positions occupés par les mots-clés que l'on souhaite colorer, travail effectué automatiquement par VSCode dans la troisième méthode.

La première méthode ne s'adapte pas au thème VSCode de l'utilisateur, puisque la couleur renvoyée par le serveur n'en dépend pas, ce qui peut créer des incohérences graphiques dans l'éditeur de texte de l'utilisateur. La deuxième méthode permet de paramétrer les couleurs affichées en fonction du thème. La troisième ne permet pas de choisir la couleur du texte et laisse ce travail au thème de l'utilisateur, ce qui est généralement mieux pour assurer une cohérence visuelle.

C'est la troisième méthode que je retiens pour la coloration syntaxique, puisque plus adaptative et simple.

Reprenons les étapes décrites dans la section "Fonctionnement général" pour ajouter une feature :

1. Dans le tableau "contribute" de l'extension manifest est référencé, dans "grammars", le fichier "./syntaxes/lp.tmLanguage.json".
2. Dans le fichier "syntaxes/lp.tmLanguage.json", on met en place la coloration syntaxique à l'aide de la syntaxe issue du logiciel TextMate, en écrivant une "grammaire TextMate".

Cette syntaxe prend la forme d'un objet JSON dont les attributs sont :

* scopeName : un nom pour la grammaire, peu important
* patterns : la liste des scopes
* repository : un objet dont les attributs sont des scopes qui ne seront pas nécessairement référencés.

Une bonne pratique est de définir effectivement les scopes dans repository, puis de les inclure dans patterns avec le mot-clé include, ce qui permet de désactiver un scope facilement (étant donné que les commentaires n'existent pas en JSON ordinaire).

Un scope est un objet essentiellement constitué des attributs :

* match : une expression régulière (syntaxe d'Oniguruma) qui identifie la portion de texte à colorer

* name : la catégorie du texte à colorer, de préférence un nom usuel dans les thèmes de couleur VSCode. Par exemple "comment.line.double-slash" pour les //commentaire, auquel on peut ajouter le suffixe ".lp" qui ne rentrera pas en compte dans la coloration, à moins de définir un thème dédié à lp qui colore spécifiquement "comment.line.double-slash.lp". En d'autres termes, le thème colore les scopes selon le plus long préfixe reconnu dans l'attribut name.

On peut y ajouter un attribut "captures" qui repère les portions parenthésées du match et permet de leur assigner une catégorie différente (par exemple pour colorer différemment le mot-clé de déclaration et la variable qui suit).
B

4.1.3 Snippets

Toujours suivant la même méthode :

1. Dans le tableau "contribute" de l'extension manifest est référencé, dans "snippets", le fichier "./snippets/unicode.json".
2. Dans le fichier "./snippets/unicode.json", on met en place les snippets.

Syntaxe d'un snippet : objet JSON avec les attributs

* prefix : string, qui lorsqu'elle est tapée par l'utilisateur, peut être complétée/remplacée par VSCode
* body : la complétion susmentionnée, suggérée par VSCode dans un rectangle sous la ligne de texte tapée
* description : une description affichée dans la liste des suggestions

Cette syntaxe étant très répétitive sur 1313 lignes de snippets, j'utilise un code Python pour générer différentes parties du fichier. J'y ai inclus quelques polices d'écriture utiles à la notation mathématique déjà incluses dans Emacs, comme les lettres gothiques ou avec double barre.

Mon maître de stage souhaite pouvoir taper des indices et des exposants à la manière de latex, _1 devenant $_1$ par exemple, cependant cette feature n'est pas possible avec les snippets VSCode, et faisait doublon avec une autre extension VSCode, que je référence dans la documentation de l'extension VSCode disponible sur le git de LambdaPi.
C

4.1.4 Commandes

Dans le manifest package.json, je référence les noms des commandes que je souhaite implémenter, sous forme d'un tableau d'objets avec pour attributs le nom de la commande et son affichage pour l'utilisateur, accessible dans la liste des commandes VSCode. Ces commandes sont, entendu que se déplacer dans une preuve signifie afficher les buts :

* avancer/reculer d'une étape dans une preuve LP
* aller à l'étape de la preuve où se trouve le curseur
* aller à la preuve suivante/précédente
* activer/désactiver "aller systématiquement à l'étape où se trouve le curseur" ou "suivi du curseur"
* centrer l'étape courante dans la fenêtre (replacement)

* activer/désactiver le replacement automatique

Par ailleurs, je définis également des raccourcis clavier, toujours dans le manifest, associés à chaque commande.

J'implémente ensuite les commandes dans le main, c'est l'objet de la partie suivante.

4.2 Implémentation des commandes et du client

4.2.1 En place à mon arrivée

On écrit le main en langage TypeScript, qui est ensuite compilé par VSCode en code JavaScript utilisé effectivement comme main.

Microsoft fournit quatre librairies implémentant ensemble le LSP et son traitement côté client pour VSCode :

* vscode-jsonrpc : implémentation de JSON-RPC
* vscode-languageclient : envoi et traitement des messages, affichage côté client
* vscode-languageserver-protocol : implémentation du LSP
* vs-code-languageserver-types : librairie de types utilisés par le LSP.

Cet ensemble de librairies couvre l'entièreté des possibilités du protocole, contrairement à la librairie LSP pour Atom, moins complète, ce qui a justifié l'abandon par Deducteam de l'extension Atom au profit de VSCode.

En pratique, je n'ai besoin que de deux classes issues de ces librairies pour mon code :

* LanguageClient, qui permet de paraméter un client, et d'utiliser les méthodes start pour lancer le serveur et initialiser le protocole, ainsi que sendRequest pour envoyer des requêtes au serveur. Les hover et definition sont traités automatiquement.
* RequestType, qui permet de définir une requête envoyée avec la méthode send d'une instance de LanguageClient.

À mon arrivée sur le stage, les fonctions utilisant ces objets sont déjà définies :

* activate : fonction lancée au démarrage de l'extension, où l'on définit une variable client et l'on appelle client.start()
* sendGoalsRequest : fonction d'envoi de requête pour récupérer les buts de la preuve en cours.

Pour plus de précisions, ce que fait effectivement client.start() est de lancer la commande bash associée au language server de LP, passée en paramètre à la construction du client, puis d'envoyer des messages JSON-RPC sur le flux standard d'entrée avant de récupérer les réponses sur le flux standard de sortie, sur un terminal interne à VSCode.

4.2.2 Mes ajouts

J'implémente les commandes mentionnées dans la partie précédente. Celles-ci nécessitent de définir des variables "globales" donnant :

* l'étape en cours de la preuve, une position (il existe un objet dans la librairie 'vscode' pour cela)
* le mode "suivi du curseur",

* le mode "fenêtre centrée sur l'étape en cours"
* le surlignage vert de la preuve déjà parcourue, qui est un objet du Decorations API mentionné plus haut, et que je paramètre pour qu'il donne un vert différent selon si le thème est "dark" ou "light"
* le panneau HTML d'affichage des buts (de type WebView panel).

Pour éviter les problèmes liés aux variables globales, ne serait-ce que la difficulté à contrôler quelles fonctions accèdent à de telles variables, j'utilise la classe ExtensionContext de l'API VSCode qui contient notamment l'état de mémoire de l'extension, dont je passe une instance comme argument des fonctions qui doivent modifier l'état de mémoire de l'extension.

La fonction d'initialisation de l'extension, activate, qui est obligatoirement appelée dans le main, prend pour argument une instance "context" fournie par VSCode de cette classe, que je n'ai plus qu'à passer en argument des fonctions qui ont besoin des variables susmentionnées, appelées workspace variables. Je peux accéder à ces variables en lecture et en écriture.

Ensuite, je n'ai plus qu'à utiliser les divers outils de l'API pour :

* associer les commandes du manifest à des fonctions dans le main
* récupérer et mettre à jour les workspace variables
* décorer le texte avec le Decoration API
* rafraîchir l'affichage des buts avec une requête au serveur
* centrer une étape de la preuve dans la fenêtre avec la built-in command 'revealLine'
* rechercher dans le texte le mot-clé "proof" pour aller à la prochaine preuve...

D

5. Serveur LambdaPi

5.1 Implémentation existante du protocole dans le serveur LP

La manipulation des objets JSON, envoyés par le client, par le serveur écrit en OCaml requiert une librairie, Yojson dans le cas de LambdaPi.

La lecture et l'écriture sur les flux standards est gérée à l'aide de la librairie standard d'OCaml, les libraires Scanf et Format permettent de formatter le texte avant ou après conversion en objet Yojson, ainsi que de formatter un fichier de logs très utile pour le debug.

Un fichier lp_lsp.ml contient une fonction main avec une boucle (arrêtée uniquement en cas d'erreur ou arrêt du programme) d'exécution du serveur, qui fait appel à un handler différent pour chaque méthode différente de requête.

Par ailleurs, un fichier lp_doc fournit un type Lp_doc.t qui représente un document, dans lequel on peut récupérer les symboles identifiés par le parser, avec leur type et la position de leur définition.

La récupération des informations du parser est gérée par le module Pure, qui est une interface entre le parser de LP et le language server.

5.2 Traitements des requêtes "hover" et "definition"

On rappelle que

* les requêtes "hover" permettent à un éditeur de texte de donner le type (ou autre information à propos) d'un mot-clé, dans un petit cadre au-dessus de la ligne courante, lorsque la souris le survole
* les requêtes "definition" permettent à l'éditeur d'aller à la position où a été défini un mot-clé.

Lors de ces deux requêtes, le client envoie au serveur la position du curseur de l'utilisateur, que le serveur doit parvenir à situer dans le document pour renvoyer les informations souhaitées.

5.2.1 Précision des problèmes rencontrés

* lorsque le serveur ne peut fournir de type pour une portion de texte (ou "token"), il renvoie une erreur, ce qui se traduit par l'affichage d'un "Loading..." désagréable côté VSCode

* on ne peut pas voir la définition ou le type d'un mot-clé défini dans un fichier .lp inclus dans le fichier actuel
* on souhaite qu'un mot-clé comportant en préfixe le nom de son module d'origine (par exemple NaturalNumbers.plus pour l'addition importée d'un module "NaturalNumbers") soit surligné en entier lors d'un hover
* la gestion des requêtes est lourde et inefficace : à chaque requête, une fonction de lp_lsp reparcoure tout le texte du fichier .lp à la recherche d'un mot à une position donnée, alors même que ce travail est déjà effectué par le parser.

Le premier problème vient simplement d'une mauvaise gestion du cas où aucun type n'est trouvé (puisque tous les tokens n'ont pas de type) : il ne faut pas renvoyer une erreur mais un Null, selon la spécification de LSP.
Le deuxième vient d'une petite erreur dans le module Pure, où une fonction ne renvoie qu'une partie des mots-clés rencontrés.

Résoudre les autres problèmes demande en revanche plus de réflexion.

5.2.2 Récupération des informations issues du parsing

Dans le module Pure, je définis une fonction qui permet, à partir d'une variable contenant les commandes du fichier .lp identifiées par le parser, de récupérer une liste des tokens du document ainsi que leur position.

Puisque les commandes de LP sont déclinées en un grand nombre de types qui s'appellent récursivement les uns les autres, cette fonction est un pattern-matching fastidieux effectué par des fonctions mutuellement récursives. J'ai abondamment utilisé l'extension ReasonML de VS-Code, qui permet un pattern matching en partie automatisé.

L'idée est que, si on considère une commande comme un arbre dont les feuilles sont des tokens munis d'un intervalle des positions qu'il occupe, alors on souhaite récupérer ces feuilles.

5.2.3 Range map pour les tokens

Pour exploiter ces feuilles, l'objectif est de construire une range map.

Cette structure est un dictionnaire particulier, où ce n'est pas une unique clé qui permet d'accéder à une valeur, mais un intervalle de clés. Cela fait sens d'utiliser cette structure ici, puisque le client n'envoie au serveur qu'une position du curseur mais qu'un token occupe un intervalle de positions. On souhaite en effet que le fait de survoler n'importe quel caractère d'un token donne le type du token en question, donc l'intervalle occupé par le mot est l'ensemble des clés permettant d'accéder aux informations sur ce token.

J'ajoute une telle map à la classe Lp_doc.t, c'est à dire la représentation des documents .lp dans le code. Le fonctionnement d'une requête hover, par exemple, est alors très simple :

* le client envoie une position
* le serveur trouve le token correspondant à cette position avec la range map
* le serveur trouve le type correspondant à ce token (stocké dans une autre map générée par le parser)
* le serveur envoie le type au client.

J'ai fait une première implémentation des range maps à l'aide de listes pour tester le fonctionnement du programme, puis une seconde à l'aide du foncteur standard OCaml Map.Make qui est bien plus optimisée : la recherche d'un token passe au pire des cas de $O(n)$ à $O(log_2(n))$ où n est le nombre de tokens du document.

Le principe de l'implémentation est le suivant :

* fournir un type pour les positions et fournir un type pour les intervalles de positions dans un module Range
* utiliser Map.Make(Range) pour créer un dictionnaire d'intervalles
* dans ce dictionnaire, redéfinir la fonction de recherche d'un token : celle-ci ne prend pas en argument un intervalle mais une position du curseur.

Pour être plus précis, avant d'utiliser Map.Make, on doit munir les intervalles d'un ordre ; en supposant que les intervalles ne se chevauchent pas, vrai car les tokens ne se chevauchent pas, on considère qu'un intervalle est avant un autre dans l'ordre s'il est situé plus tôt dans le texte du fichier .lp.
Comme les clés du module créé par Map.Make sont des intervalles, pour chercher un token correspondant à une position il suffit de transformer cette position en intervalle ponctuel. Seulement, cette position n'est égale a priori à aucune clé disponible dans la map. Pour que la structure fonctionne, il faut considérer qu'un intervalle est égal à un autre si l'un des deux est inclus dans l'autre (puisque dès qu'une position est incluse dans un intervalle, c'est la bonne clé pour le token).
Par exemple si je cherche la position clé_pt dans la map, contenue dans un certain intervalle de positions clé_int, la fonction de recherche dans la map compare succesivement clé_pt à ses clés jusqu'à tomber sur l'intervalle clé_int ; on a alors clé_pt = clé_int d'après l'ordre qu'on a défini, donc on peut renvoyer le token associé à clé_int.
E

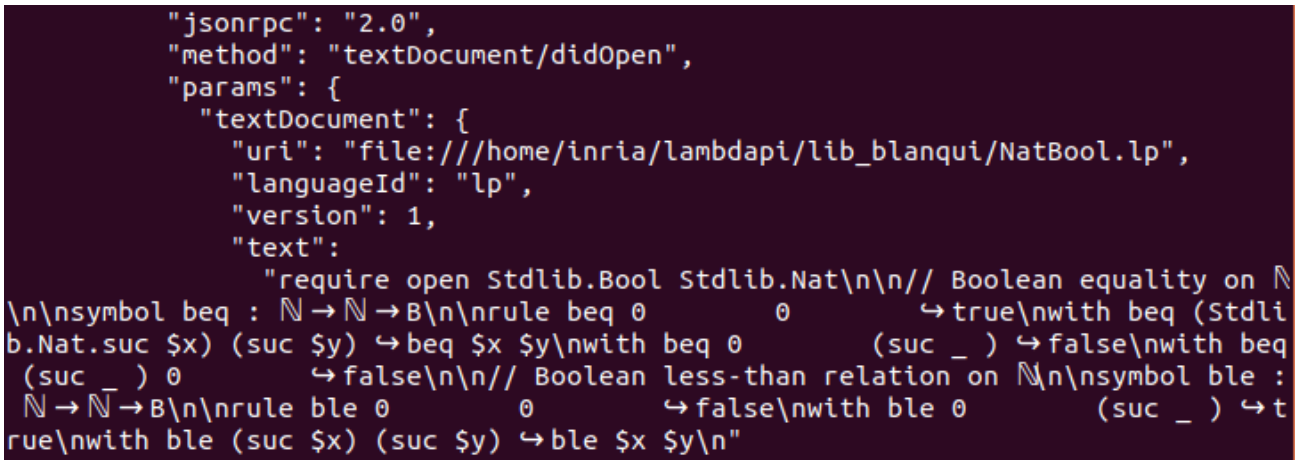
6. Conclusion

Au final, l'interface VSCode s'avère utile pour les utilisateurs de LambdaPi. Elle fonctionne et a été intégrée au code source sur la branche principale du git LambdaPi.
Perspectives d'amélioration :

* s'assurer que les modifications de fichiers inclus se répercutent au niveau du serveur
* afficher le panneau des buts à la demande, et le colorer avec une feuille de style CSS
* adapter l'extension à VSCodium qui est peu éloigné de VSCode
* parser le moins de texte possible à chaque modification d'un fichier .lp
* inférer le type des arguments d'un symbole
* gérer les symboles définis dans des fichiers .lpi (fichiers interface)
* donner le résultat des queries LP, faire des queries avec l'interface
* afficher plus d'informations sur les symboles
* permettre l'auto-complétion des noms de fonctions, notamment importées

Appendices

A. Exemple : message JSON-RPC envoyé du client au serveur



Objet JSON envoyé du client au serveur

La méthode "didOpen" est celle de l'ouverture d'un document : ici, uri est le chemin du document .lp, le langage et sa version sont également indiqués, suivis du texte du document en entier.

B. Exemple : coloration syntaxique

```
"external": {
  "match": "(open|require)\\s+(([^\\s+]*\\s+)*$)",
  "captures": {
    "1": {"name": "storage.type.lp"},
    "2": {"name": "entity.name.function.theorem.lp"}
  }
},
```

Syntaxe TextMate

"external" est un scope défini dans "repository", qui englobe les mots-clés d'importation de modules en LP.
Dans le pattern matching ("match"), la première capture correspond au mot-clé open ou require, et la deuxième au nom des modules importés. L'attribut "captures" permet d'assigner une certaine coloration à chaque capture.

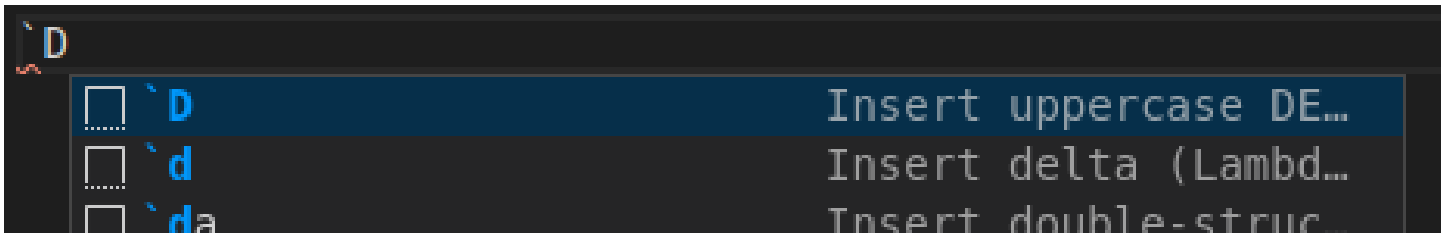
```
require open Stdlib.List
require open Stdlib.Bool Stdlib.Nat
```

Coloration d'un fichier .lp

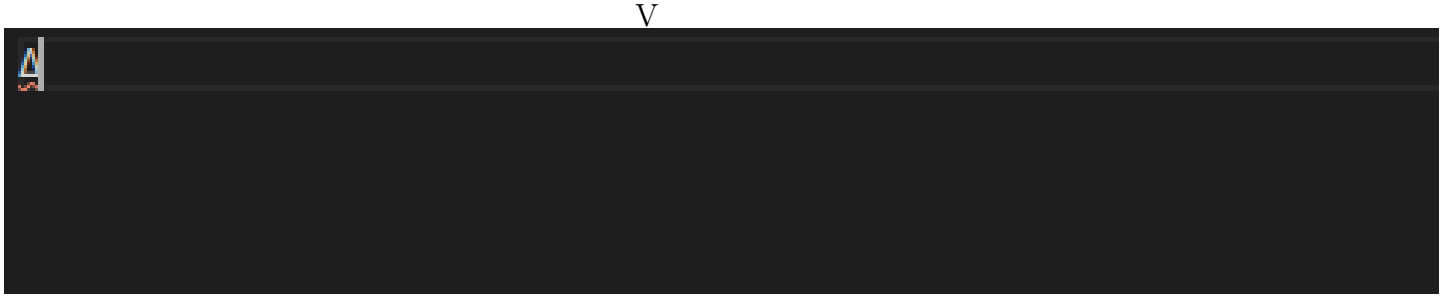
C. Exemple : snippet

```
"DELTA": {
    "prefix": "`D",
    "body": "Δ",
    "description": "Insert uppercase DELTA"
},
```

Syntaxe de déclaration d'un snippet

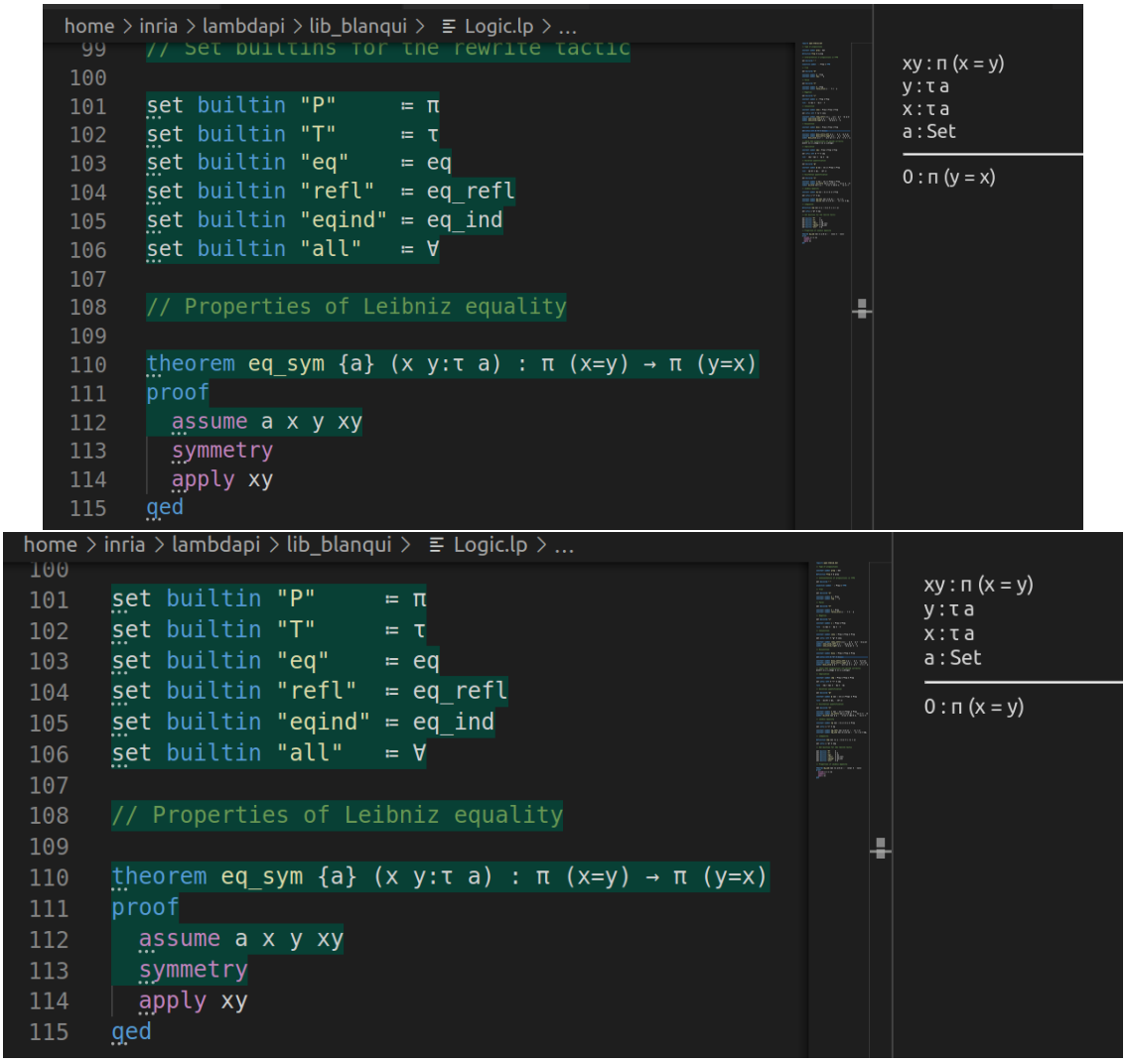


V



Complétion avec Tab dans un fichier .lp

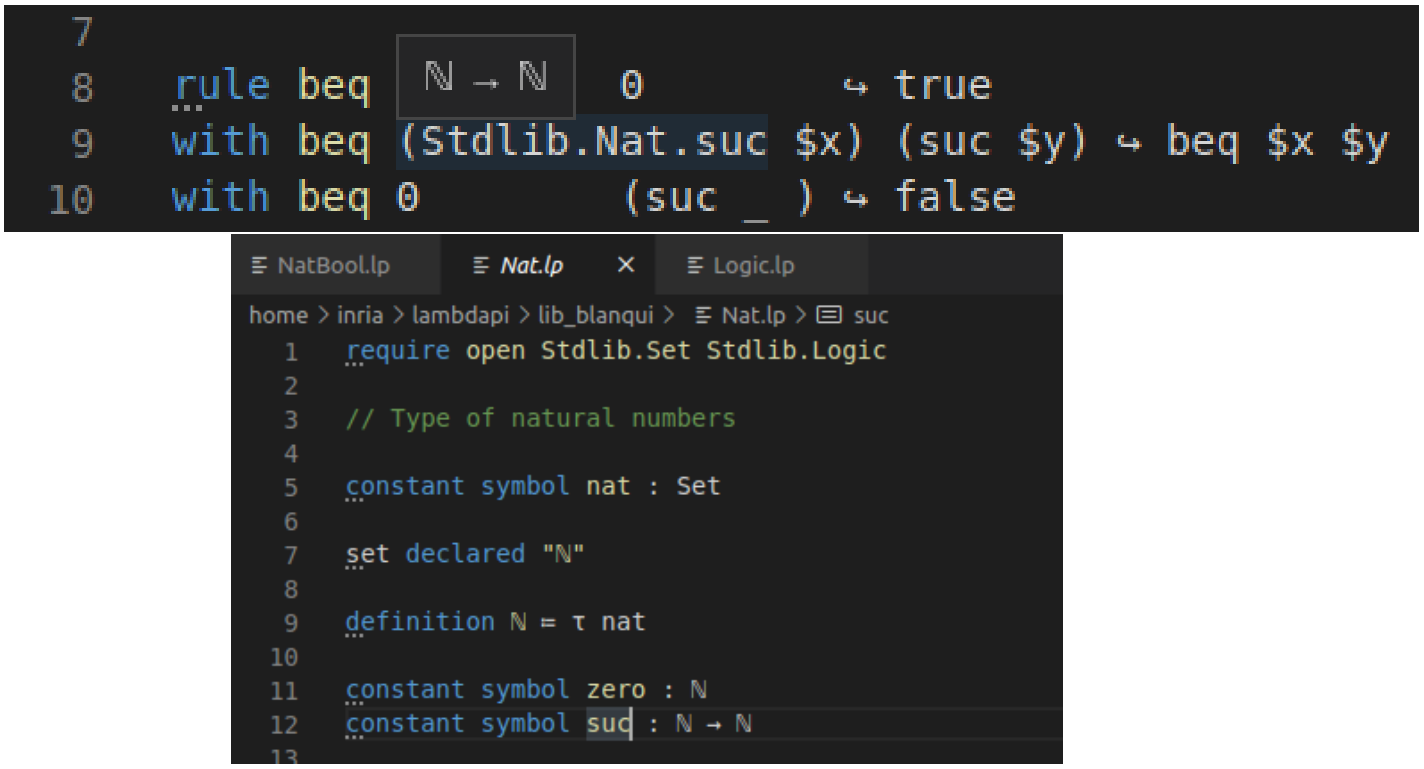
D. Exemple de navigation de preuve



Déplacement d'une étape dans la preuve avec Ctrl+shift+Bas

Application de la stratégie symmetry, les rôles d'x et y sont échangés dans le panneau "Goals" à droite de l'écran.

E. Exemple : hover et definition



hover, definition

Le survol du symbole suc (qui donne le successeur d'un entier naturel) donne son type ; appuyer sur F12 affiche alors sa définition.

F. Annexe : Développement durable et responsabilité sociétale

Une journée de stage nécessite l'utilisation d'un ordinateur entre 9h et 17h30, avec une heure de pause déjeuner. Afin de diminuer ma consommation en électricité, le laboratoire m'a fourni un ordinateur moins consommateur que celui dont je dispose.

Par ailleurs, une connexion internet est presque tout le temps nécessaire pour communiquer avec l'équipe, utiliser git, consulter la documentation VSCode ou OCaml.

J'ai effectué le déplacement à l'ENS pour les soutenances blanches en covoiturage. Les transports en commun ne permettaient pas de faire le trajet sans bus de remplacement ce jour-là.

Il est difficile de parler de l'impact sur la société qu'aura le travail réalisé durant le stage, étant donné que cette thématique appartient à la recherche fondamentale. À l'échelle de la vérification en général, cependant, l'enjeu de ces travaux est important dans tous les domaines où la sécurité informatique est cruciale : en particulier l'énergie, le transport, le médical ; on peut aussi mentionner la vérification formelle des protocoles de sécurité. L'objectif à long terme est de garantir la sûreté des programmes. Dans le cas de Dedukti, toutefois, cet enjeu est lointain, puisque les thèmes de recherche sont plutôt abstraits et fondamentaux.

Une mesure plus concrète a cependant été prise cette année à l'Inria : les équipes de recherche doivent désormais faire des rapports dans leur conseil d'administration à propos de leurs actions pour diminuer l'emprunte carbone de leurs travaux.

Pour ce qui est de la responsabilité sociétale et de l'inclusivité, l'équipe comprend, durant mon stage, une étudiante en doctorat et deux femmes stagiaires, soit 3 femmes sur 25.

Le responsable de l'équipe, Gilles Dowek, a présidé l'ARDHIS (Association pour la reconnaissance des droits des personnes homosexuelles et transsexuelles à l'immigration et au séjour).

G. Références

VSCode Extension API : https ://code.visualstudio.com/api

LSP : https ://microsoft.github.io/language-server-protocol/

Résumé

Durant ce stage, j'ai largement complété une interface graphique d'assistant de preuve pour LambdaPi, et ce dans une extension de l'éditeur VSCode. Le but est de permettre l'utilisation du langage Dedukti 3 comme d'un véritable assitant de preuve, en plus d'un logical framework. J'ai pour cela mis à profit le Language Server Protocol, qui permet la communication entre une interface utilisateur et le parser de Dedukti 3.

Mots-clés : Logique, Assistant de preuve, Dedukti, Language Server Protocol