Table des matières

Introduction

Nous sommes étudiantes à l'Université Pierre et Marie Curie, en première année de Master Sciences et Technologies, mention Informatique, spécialité Science et Technologie du Logiciel.

Notre projet de second semestre consiste à développer un logiciel d'enseignement pour le cours de Logique pour l'Informatique que nous avons suivi au premier semestre. Ce cours s'intéresse, dans sa seconde partie, au lien entre preuves et programmes. Il initie l'étudiant à la construction de preuves en déduction naturelle, au λ-calcul et au typage de λ-termes. Notre logiciel doit aider l'étudiant à comprendre ces notions par la pratique. Il offre pour cela trois fonctionnalités. Il permet, d'abord, de construire une preuve en utilisant les règles de déduction de la logique propositionnelle, ensuite, de typer un λ-terme pas à pas en indiquant quelle règle de typage est utilisée à chaque étape, enfin, de combiner les résultats obtenus grâce aux fonctionnalités précédentes pour illustrer l'isomorphisme de Curry-Howard. Notre logiciel est implanté en Ocaml.

Ce rapport contient, dans une première partie, les principales définitions et théorèmes nécessaires à la compréhension de notre programme. Dans une deuxième partie, nous présentons notre assistant à la preuve suivi, dans une troixième partie, de la description de notre outil de typage. Enfin, nous montrons comment l'étudiant peut mettre à profit nos outils d'aide à la preuve et de typage de λ-termes pour comprendre l'isomorphisme de Curry-Howard.

1.Outils théoriques

1.1.Les règles de déduction de la logique propositionnelle

A notre niveau, une preuve mathématique est une suite d'applications de règles de déduction à partir d'axiomes. Pour construire une preuve et nous assurer de sa correction, nous devons d'abord fixer ces règles et axiomes.

Nous travaillons dans le cadre de la **logique propositionnelle**. La syntaxe de la logique propositionnelle est fondée sur des variables de proposition, ici notées A...Z. Ces variables sont combinées au moyen de connecteurs logiques pour former des formules ou propositions.

Les **formules** de la logique minimale propositionnelle sont définies inductivement par :

- tout symbole propositionnel est une formule
- si φ_1 et φ_2 sont des formules, alors $\varphi_1 \Rightarrow \varphi_2$ est une formule

Nous ajoutons à l'implication, la négation (si A est une formule, alors ¬A est une formule), la conjonction (le connecteur Λ est défini par A Λ B = ¬ (A ⇒ ¬B)) et la disjonction (le connecteur V est défini par A V B = ¬A ⇒ B).

Un **séquent** est un couple ($\mathscr{F}$, F) où $\mathscr{F}$ est un ensemble fini de formules et F une formule. Un séquent formalise la notion de conséquence logique : si toutes les prémisses du séquent, c'est-à-dire les formules de $\mathscr{F}$, sont vraies, alors la conclusion du séquent, c'est-à-dire la formule F, est vraie. [biblio3]

Les **règles du calcul des séquents** que nous avons implantées dans notre assistant à la preuve sont alors :

$$\textbf{(Hyp)}\ \frac{}{A, \Gamma \vdash A} \qquad \textbf{(I ⇒)}\ \frac{A, \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \textbf{(E ⇒)}\ \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\textbf{(I Λ)}\ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \textbf{(E}^{\textbf{l}}\textbf{ Λ)}\ \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \textbf{(E}^{\textbf{r}}\textbf{ Λ)}\ \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

$$\textbf{(I}^{\textbf{l}}\textbf{ V)}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \textbf{(I}^{\textbf{r}}\textbf{ V)}\ \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\textbf{(E V)}\ \frac{\Gamma \vdash A \vee B \quad A, \Gamma \vdash C \quad B, \Gamma \vdash C}{\Gamma \vdash C} \qquad \textbf{(E ⊥)}\ \frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

[biblio1]

1.2.Le λ-calcul non typé

Le **λ-calcul** est un modèle établi pour réprésenter les fonctions. Il est muni de deux opérations : l'application et l'abstraction. L'application s'apparente intuitivement à l'application d'une fonction mathématique à un argument ; par exemple (t_1 t_2) représente l'application de t_1 à l'argument t_2. L'abstraction consiste intuitivement à remplacer une fonction mathématique par son nom ; par exemple l'abstraction $\lambda x.t$ est la fonction qui à x associe t. [biblio2]

Les termes du λ-calcul (les **λ-termes**) sont définis inductivement par :

- tout symbole de variable est un terme
- si t_1 et t_2 sont des termes, alors (t_1 t_2) est un terme
- si x est un symbole de variable et t un terme, alors $\lambda x.t$ est un terme [biblio1]

1.3.Le λ-calcul typé

Le **λ-calcul typé** permet de se rapprocher de la définition ensembliste d'une fonction. Typer un λ-terme permet de lui donner une sémantique, on peut alors distinguer dans l'ensemble des λ-termes ceux qui ont un sens : ce sont les λ-termes typables.

Le typage est une interprétation abstraite des λ-termes : un **type** est une valeur abstraite représentant tous les λ-termes de ce type ; par exemple la valeur abstraite int→int désigne toutes les fonctions des entiers dans les entiers.

Un **contexte de typage** est un ensemble fini de couples de la forme (x, τ) où x est une variable et τ est le type associé à cette variable.

Les **règles de typage** permettent alors d'associer un type à un λ-terme dans un contexte Γ donné. Voici celles que nous avons implantées :

$$\textbf{(Var)}\ \frac{}{(x,\tau), \Gamma \vdash x : \tau} \qquad \textbf{(App)}\ \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}$$

$$\textbf{(Abs)}\ \frac{(x,\tau_1), \Gamma \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \textbf{(Pair)}\ \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}$$

$$\textbf{(Fst)}\ \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{fst}\ (t) : \tau_1} \qquad \textbf{(Snd)}\ \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{snd}\ (t) : \tau_2}$$

$$\textbf{(Injl)}\ \frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathrm{inj}^{l}_{\tau1,\tau2}\ (t) : \tau_1 + \tau_2} \qquad \textbf{(Injr)}\ \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathrm{inj}^{r}_{\tau1,\tau2}\ (t) : \tau_1 + \tau_2}$$

$$\textbf{(Case)}\ \frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad (x_1 : \tau_1), \Gamma \vdash t_1 : \tau \qquad (x_2 : \tau_2), \Gamma \vdash t_2 : \tau}{\Gamma \vdash \mathrm{case}_{\tau1,\tau2}\ t\ \mathrm{of}\ \mathrm{inj}^{l}_{\tau1,\tau2}\ (x_1) \to t_1 : \tau \ \mid\ \mathrm{inj}^{r}_{\tau1,\tau2}\ (x_2) \to t_2 : \tau}$$

1.4.L'isomorphisme de Curry-Howard

L'**isomorphisme de Curry-Howard** relie la logique mathématique et l'informatique, par l'intermédiaire du λ-calcul. Il établit la correspondance entre preuves et programmes.

preuve	≡	λ-terme	≡	programme
proposition	≡	type	≡	spécification [biblio1]

A chaque preuve, on peut associer un programme informatique [biblio2]. En effet, en associant une instruction à chaque règle de déduction, on transforme une preuve en une suite d'instructions, c'est-à-dire en un programme. Par exemple, la règle du modus ponens permet à partir d'une démonstration D_A de la formule A et d'une démonstration $D_{A\to B}$ de $A\to B$ d'obtenir une démonstration D_B de la formule B ; c'est la formulation littéraire de notre règle (E⇒) du calcul des séquents. On peut lui associer l'instruction qui prend les programmes P_A et $P_{A\to B}$ et qui donne le programme $P_B = P_{A\to B}(P_A)$, soit l'équivalent de la règle de typage (App) ci-dessus. En faisant de même pour chacune des règles de déduction, on sait donc fabriquer un programme à partir d'une preuve quelconque.

L'isomorphisme de Curry-Howard nous permet aussi de dire que typer un programme revient à prouver un théorème. Pour un théorème T donné, il existe différentes démonstrations de T, il existe donc différents programmes de même type. Si on peut associer à une formule un λ-terme typable, alors on aura prouvé que cette formule est un théorème.

Le but de notre projet est de fournir à l'étudiant une interface simple qui lui permette de construire ses preuves en logique propositionnelle et de typer ses λ-termes. Or la syntaxe familière à l'étudiant, qui construit ses preuves sur le papier et veut simplement vérifier leur correction à l'aide de la machine, n'est pas directement reconnue par Ocaml.

Pour offrir une bonne ergonomie, notre logiciel doit donc transformer les chaînes de caractères entrées par l'étudiant - la syntaxe concrète, en une syntaxe abstraite, manipulable par le noyau de notre application.

Nous avons utilisé deux structures de données différentes pour réaliser nos analyses lexicale et syntaxique : le noyau de notre assistant à la preuve manipule des listes alors que le noyau de notre typeur de λ-termes manipule des flux. Il nous semble intéressant de vous présenter dans ce rapport ces deux approches. Nous avons commencé par implémenter l'assistant à la preuve manipulant des listes, ce qui nous a permis de cerner les problèmes liés aux analyseurs lexicaux et syntaxiques. La version manipulant des flux, développée ensuite, s'appuie davantage sur la documentation existant sur le langage Ocaml [biblio6]. Elle est mieux adaptée aux problèmes d'analyse lexico-syntaxique – nous expliquerons pourquoi par la suite, mais n'aurait pas pu être réalisée sans l'apport pédagogique de notre première approche.

2.Notre assistant à la preuve

2.1.Fonctionnement souhaité

L'étudiant saisit la formule à prouver, il choisit la règle de déduction à appliquer et saisit, si besoin, les formules qui n'apparaissent pas dans la conclusion de la règle choisie.

L'assistant à la preuve engendre les nouveaux sous-buts à prouver et l'étudiant poursuit sa preuve. Le programme détecte automatiquement la fin de la preuve.

Si une formule saisie n'est pas valide ou si la règle choisie n'est pas applicable aux formules saisies, l'assistant affiche une erreur.

2.2.Implantation

Le noyau

Le noyau de notre assistant à la preuve définit les types *formule* et *sequent* conformément à leur description mathématique [cf. 1.1] et implante les règles de déduction de la logique propositionnelle.

```
type formule =
    SINGLE of string
  | IMPLIC of formule * formule
  | AND of formule * formule
  | OR of formule * formule
  | FALSE ;;

type sequent = {hyp : formule list; f : formule };;
```

Nous présentons, en exemple, l'implantation de deux des règles de déduction de la logique propositionnelle. [voir annexe1 pour la liste exhaustive]

La fonction *introImplic* prend en argument un séquent *s* et retourne la liste des séquents

permettant de prouver *s* par l'application de la règle d'introduction du connecteur ⇒. *introImplic* lève l'exception *RegleNonApplicable* si le séquent *s* est syntaxiquement incorrect.

Règle d'introduction du connecteur ⇒

```
let introImplic s =
  match s.f with
    IMPLIC (a,b) -> [{hyp = a::s.hyp ; f = b}]
  | _ -> raise RegleNonApplicable;;
```

La fonction *elimImplic* est légèrement différente : elle implante la règle d'élimination du connecteur ⇒ qui nécessite, en entrée, un séquent *s* mais aussi une formule *a* qui n'apparaît pas dans la conclusion de la règle.

Règle d'élimination du connecteur ⇒

```
let elimImplic s a =
  [{hyp = s.hyp ; f = IMPLIC (a, s.f)} ;
   {hyp = s.hyp ; f = a}];;
```

L'interface

Nous avons choisi d'utiliser les bibliothèques du GTK (GIMP Toolkit) et **LablGTK2**, qui permet d'interfacer le GTK avec Objective Caml, pour créer notre interface graphique.

Pour LablGTK2, une interface est conçue comme un assemblage de conteneurs dans lesquels nous pouvons placer différents « widgets », c'est-à-dire des boutons, du texte ou des zones de saisie. Ces widgets sont en attente d'événements (clic de souris, déplacement du curseur ou saisie d'un texte, par exemple). Lorsque, par exemple, l'utilisateur clique sur un bouton, le widget envoie un signal à la fonction principale, dans laquelle nous avons défini l'interface. Des « signal handlers » nous permettent alors de préciser le traitement approprié au signal reçu. [biblio4]

Nous avons conçu, de cette façon, une interface permettant à l'étudiant d'entrer la formule à prouver et les hypothèses du contexte de preuve. Des boutons radio lui permettent de sélectionner pas à pas la règle à appliquer à sa formule. L'étudiant peut et doit aussi entrer les formules qui n'apparaissent pas dans la formule à prouver, dans les cas où il voudrait appliquer les règles d'élimination des connecteurs ⇒, Λ, et V, que nous avons clairement distingué dans l'interface. La preuve s'affiche pas à pas sous la forme d'un arbre d'inférence. L'étudiant peut revenir en arrière dans sa preuve et recommencer une nouvelle preuve à volonté. [voir capture d'écran plus bas]

Les fonctions de « liaison noyau-interface »

Les formules entrées par l'utilisateur sont récupérées par le programme sous la forme d'une chaîne de caractères. Cette chaîne de caractères doit être adaptée à la syntaxe définie dans notre noyau.

Les fonctions *transforme* et *transforme2* assurent le passage « interface vers noyau ». Elles prennent en argument la chaîne de caractère entrée par l'utilisateur et effectuent les adaptations nécessaires.

transforme traîte le cas où l'utilisateur a entré une seule formule. Elle commence par éliminer tous les espaces dans la chaîne *s* entrée, puis identifie la nature de la formule entrée (singleton, implication, conjonction, disjonction ou négation), enfin retourne l'objet de type *formule* correspondant. *transforme* lève l'exception *Erreur_syntaxe* si *s* est syntaxiquement incorrecte.

transforme2 traîte le cas où l'utilisateur a entré plusieurs formules. Elle sépare chaque formule entrée et la stocke dans une liste, puis appelle *transforme* sur chaque élément de cette liste.

Réciproquement, les fonctions *trans_to_string* et *trans_to_list* assurent le passage « noyau vers interface ». Elles prennent en argument l'objet de type *formule,* résultat de l'application de la règle choisie par l'étudiant, et la transforment en une chaîne de caractères pour l'afficher sur l'interface.

Les structures de données importantes

Cinq listes sont importantes dans notre programme :

- *list_txt* : permet de détecter la fin de la preuve. *list_txt* contient tous les séquents qui se trouvent dans l'arbre, sur lesquels l'étudiant doit appliquer une règle. *list_txt* fonctionne comme une pile : lors de l'application d'une nouvelle règle, la tête de liste est remplacée par le nouveau résultat.
- *txt_visite* : est le complémentaire de *list_txt*. *txt_visite* contient les séquents des étapes passées de la preuve, ce qui permet d'implémenter la fonction de retour en arrière dans la preuve.
- *dern_regle* : permet de mémoriser le nom des règles appliquées jusqu'à l'état courant. *dern_regle* est utilisée par la fonction de retour arrière ; elle nous indique jusqu'où nous devons dépiler les quatre autres piles.
- *branche_active* et *branche_visitee* : permettent de détecter dans quelle branche de la preuve se trouve l'étudiant. Ce sont des listes de conteneurs au sens de LablGTK. *branche_active* et *branche_*visitee permettent une vision « orientée interface » de la preuve, alors que *list_txt* et *txt_visite* travaillaient sur les objets du noyau.

Les fonctions de traitement des événements de l'interface

Les structures de données décrites ci-dessus nous ont permis d'enrichir le traitement des événements de l'interface. Le schéma général de traitement des différents événements est donc le suivant :

- réception du signal émis par le widget ;
- appel des fonctions *transforme* et *transforme2* et récupération du résultat sous-forme d'un objet de type *formule* ;
- application de la règle choisie à la formule ;
- mise à jour des listes concernées ;
- transformation du résultat de l'application de la règle en une chaîne de caractère destinée à l'affichage.

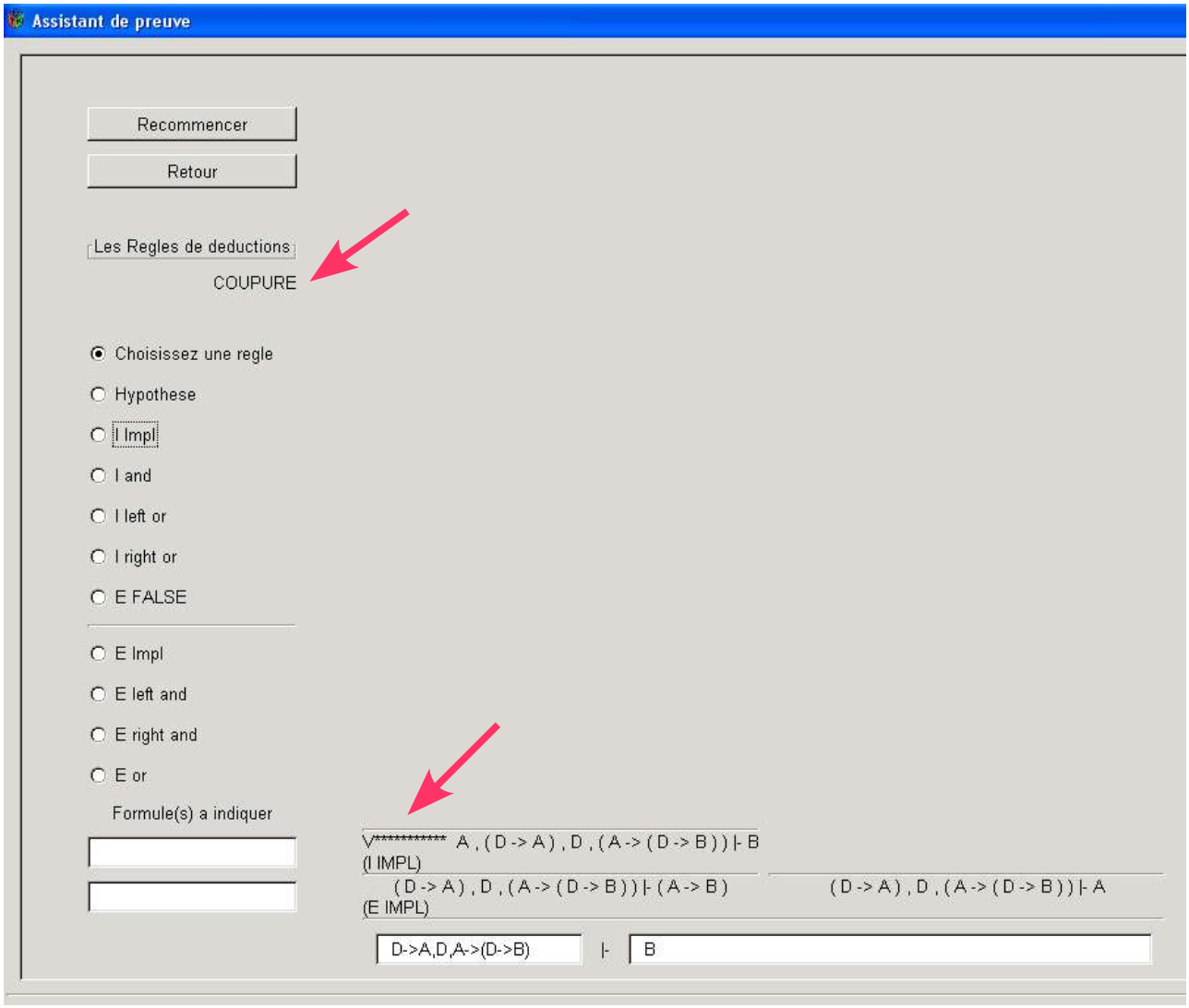


Illustration 1 : Notre assistant à la preuve, exemple d'un début de preuve avec coupure

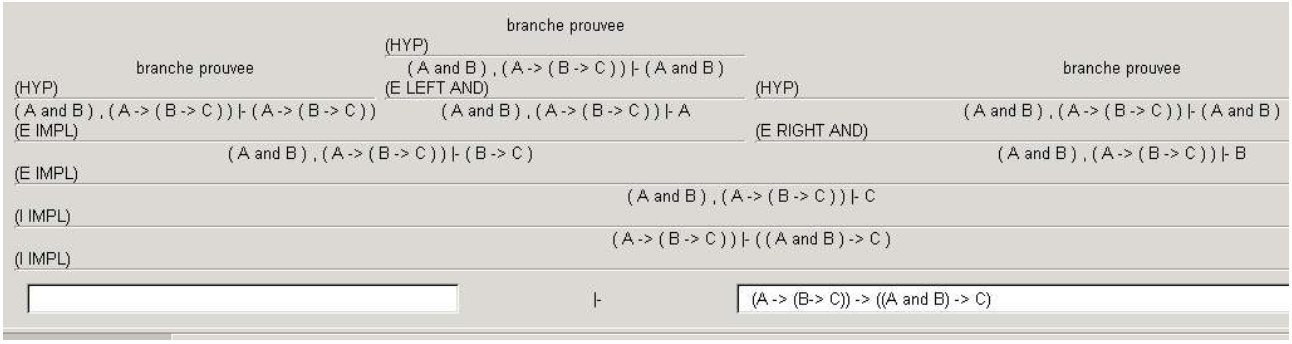


Illustration 2 : Agrandissement d'une seconde preuve

3.Notre outil de typage

3.1.Fonctionnement souhaité

L'étudiant entre le λ-terme à typer.

L'outil de typage génère les types associés au λ-terme entré par l'étudiant et affiche les nouveaux sous-termes à typer. L'outil affiche également le système de contraintes sur les types au fur et à mesure de la construction de l'arbre de typage.

Si la règle choisie n'est pas applicable au λ-terme saisi, l'assistant affiche une erreur. Si le λ-terme entré n'est pas typable, c'est-à-dire que le système de contraintes sur les types n'admet pas de solution, le programme affiche une erreur et s'arrête.

3.3.Implantation

Nous voulons, une nouvelle fois, permettre à l'étudiant d'écrire les λ-termes qu'il veut typer suivant la syntaxe utilisée dans le cours de Logique de M1 et que cette écriture, agréable pour lui, engendre une valeur Ocaml de type *terme* défini par :

```
type terme =
    V of var
  | App of terme * terme
  | Abs of var * terme
  | Pair of terme * terme
  | Fst of terme
  | Snd of terme
  | Injl of typ * typ * terme
  | Injr of typ * typ * terme
  | Case of typ * typ * var * var * terme * terme * terme ;;
```

Les flux

Nous utilisons, cette fois, les flux pour programmer l'analyse lexicale et syntaxique du langage. Tout comme les listes, les flux sont des suites de valeurs de même type. Le type d'un flux est *t stream* en Ocaml [biblio6], où *t* est le type des éléments du flux.

Les flux diffèrent des listes sur deux points importants, qui les rendent mieux adaptés aux problèmes d'analyse lexico-syntaxique. Première différence : l'accès dans un flux est destructif. Lorsqu'on consulte le premier élément d'un flux, cet élément est aussitôt retiré du flux et remplacé par l'élément suivant, ce qui convient à notre analyseur qui n'a pas besoin de conserver les caractères entrés par l'étudiant une fois qu'ils ont été lus. Deuxième différence : l'évaluation des éléments contenus dans un flux se fait pas à pas. Lorsqu'on construit un flux des caractères, on mémorise le caractère courant et on ne va chercher le caractère suivant sur la source que lorsqu'on en a besoin. Cette évaluation paresseuse est plus économique en mémoire.

Analyse lexicale

Le module *Genlex* de Ocaml fournit une primitive permettant d'analyser une suite de caractères selon plusieurs catégories d'unités lexicales prédéfinies (*Kwd* pour les identificateurs distingués ou les caractères spéciaux, *Ident* pour les identificateurs de variables ou de fonctions, *Int*, *Float*, *String*, *Char*) [biblio5]. Nous l'utilisons pour générer l'analyseur des caractères entrés par l'étudiant.

```
let keywords =
[ "lambda" ; "fst" ; "snd" ; "injl" ; "injr" ; "case" ; "of" ; "->" ; "." ; "(" ; ")" ;
"," ; "|" ;"+";"*"] ;;

let analyseur_lexical l = Genlex.make_lexer keywords l ;;
```

La primitive *make_lexer* produit un flux d'unités lexicales – les lexèmes – à partir du flux de caractères entrés. (Remarque : elle remplit les mêmes fonctions que *transforme*, précédemment dans notre outil d'aide à la preuve)

Analyse syntaxique

Pour assembler les lexèmes produits par *make_lexer* de façon à former des phrases grammaticalement correctes, lisibles par notre noyau fonctionnel, nous effectuons une analyse syntaxique descendante (de gauche à droite) facilitée par notre choix d'utiliser des flux Ocaml.

La fonction *analyseur_syntaxique* lit les lexèmes et les transforme en éléments de type *terme*. Nous vous présentons en exemple un extrait de la fonction *analyseur_syntaxique* qui traite les règles de typage de l'application (App), l'abstraction (Abs) et le (Case) [cf. 1.3].

```
let analyseur_syntaxique l =
  try (
    let rec termes_rec =  parser
      | [< 'Ident c >] -> V (Var c)
      | [< 'Kwd "(" ; t = lambda ; 'Kwd ")" >] -> t

    and app = parser
      | [< t = termes_rec ; lam = (other_applications t) >] -> lam

    and other_applications f = parser
      | [< arg = termes_rec ; stream >] ->
          other_applications (App (f, arg)) stream
      | [< 'Kwd "," ; arg = termes_rec >]  -> Pair(f,arg)
      | [<>] -> f

    and lambda = parser
      | [< 'Kwd "lambda" ;'Ident c; 'Kwd ".";t1 = lambda >] -> Abs(Var c,t1)
[...]
     | [< 'Kwd "case" ; 'Kwd "(" ;  a = types ; 'Kwd "," ; b = types ; 'Kwd ")" ; t = termes_rec ;
          'Kwd "of" ;
          'Kwd "injl" ; 'Kwd "(" ;  c = types ; 'Kwd "," ; d = types ; 'Kwd ")" ;
                        'Kwd "(" ; 'Ident v1 ; 'Kwd ")" ; 'Kwd "->" ; t1 = termes_rec ;
          'Kwd "|" ;
          'Kwd "injr" ; 'Kwd "(" ;  e = types ; 'Kwd "," ; f = types ; 'Kwd ")" ;
                        'Kwd "(" ; 'Ident v2 ; 'Kwd ")" ; 'Kwd "->" ; t2 = termes_rec >] ->
            if ((a == c) && ( a == e) && (b == d) && (b == f)) then
               Case(a,b,Var v1,Var v2,t,t1,t2)
            else raise Erreur_syntaxe
      | [<>] -> f
          in termes_rec l
      ) with Stream.Error ("") -> raise Erreur_syntaxe ;;
```

En composant *analyseur_lexical* et *analyseur_syntaxique*, nous obtenons une fonction qui transforme un flux de caractères (syntaxe concrète) en un arbre (syntaxe abstraite). [voir annexe2 pour la liste exhaustive des correspondances entre syntaxes concrètes et abstraites]

Typage

Nous voulons maintenant typer les termes produits par l'analyse lexicale et syntaxique. Nous disposons d'un enregistrement correpondant à notre environnement de typage pour chaque λ-terme entré par l'étudiant.

```
type typ =
  | SINGLE of string
  | IMPL of typ * typ
  | PRODUIT of typ * typ
  | SOMME of typ * typ ;;

type assoc = ASSOC of var * typ ;;
type sequent = {env: assoc list ; lambda_terme: terme ; typage: typ} ;;
```

Les équations entre types

La synthèse de types est analogue à la résolution d'équations mathématiques. Mais à la différence des mathématiques où nous devons résoudre un ensemble d'équations données par avance, le synthétiseur de types doit découvrir dans le terme qui lui est soumis l'ensemble des équations à résoudre.

De plus, l'algorithme de synthèse de types ne devra pas introduire de nouvelles inconnues inutilement. Par exemple, si nous avons λx.y de type $t_1 \rightarrow t_2$ et que nous voulons typer x et y, nous devrons utiliser les types t_1 et t_2 qui interviennent déjà dans le problème.

Enfin, l'algorithme n'attend pas d'avoir entièrement construit le système d'équations pour commencer à le résoudre : il effectue simultanément l'introduction des équations et leur résolution.

Méthode de résolution

Nous résolvons les équations de types par substitution. Par exemple, si $t_1=t_2*t_3$ et que nous savons que $t_3=t_4 \rightarrow t_5$, nous obtenons $t_1=t_2*(t_4 \rightarrow t_5)$. Le mécanisme de résolution est, en réalité, plus général : c'est une méthode de propagation de contraintes d'égalité connue sous le nom de mécanisme d'unification [biblio6].

L'unification consiste à résoudre un ensemble d'équations, en donnant aux variables de type des valeurs qui rendent toutes les équations vraies. Nous représentons les équations par des couples de types. L'unification revient à prendre deux types et à les rendre égaux s'ils ont été obtenus par le même constructeur principal.

Avant d'unifier deux types, nous effectuons un test d'occurence pour nous assurer qu'il n'y a pas de cycle. Par exemple, si on a $t_1=t_2*t_1$, t_1 a une occurrence dans t_2*t_1 et dans ce cas on lance une exception pour que le typeur s'arrête tout de suite : les types ne sont pas unifiables donc le λ-terme n'est pas typable.

L'interface

Nous contruisons une interface similaire à celle de l'assistant à la preuve. L'étudiant dispose d'une zone de saisie prévue pour introduire le λ-terme à typer et de boutons radio lui permettant de choisir la règle de typage à appliquer à son λ-terme. La zone d'affichage principale est réservée à l'arbre de preuve, le système de contraintes sur les types s'affiche sur la droite, il est résolu par notre outil qui affiche toutes les équations pas à pas.

Si la règle choisie par l'étudiant n'est pas applicable au λ-terme courant, l'outil le lui signale et l'étudiant peut directement choisir une autre règle. Le bouton « Recommencer » efface entièrement l'arbre de preuve, pour typer un nouveau λ-terme.

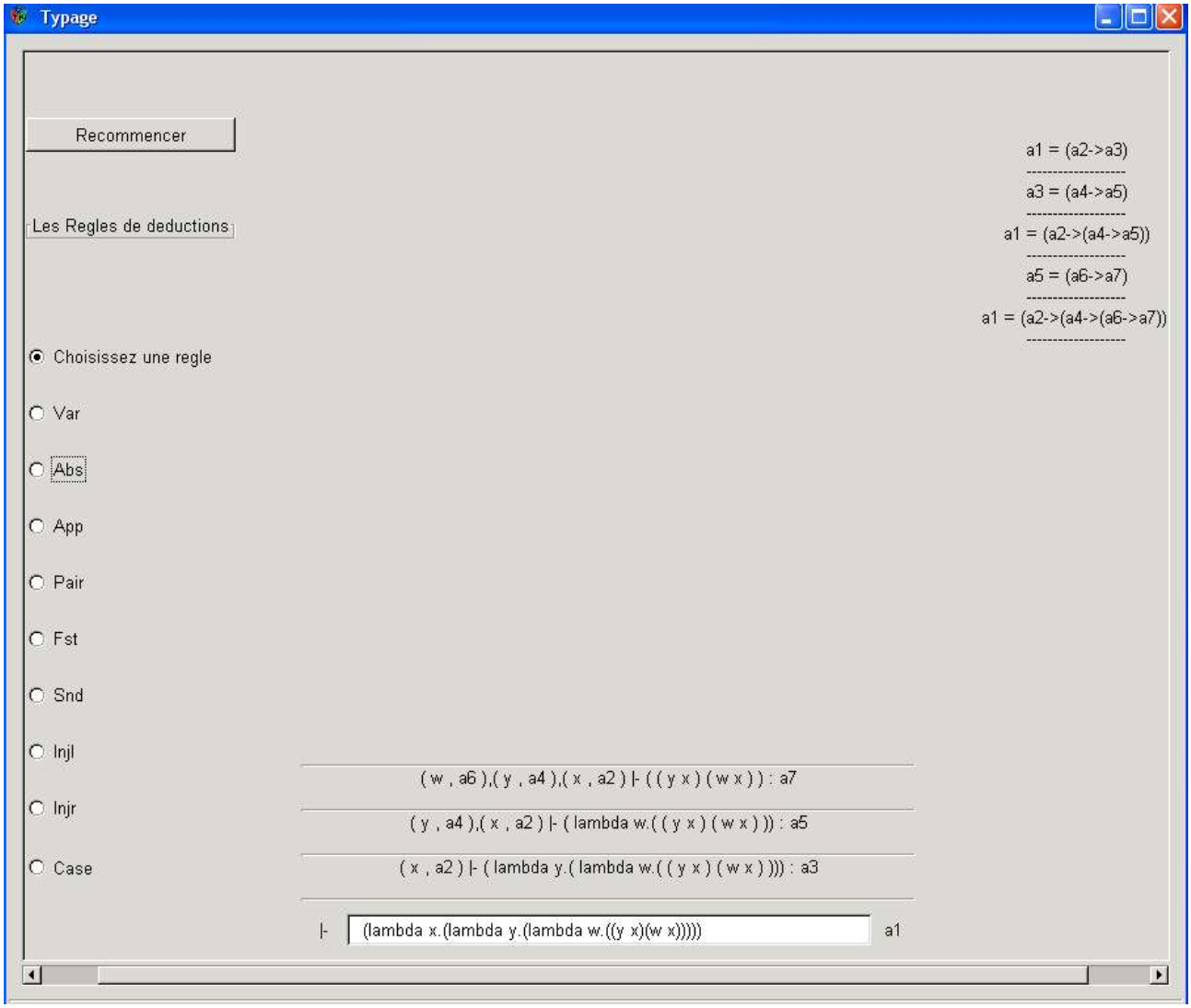


Illustration 3 : Notre outil de typage d'un lambda-terme

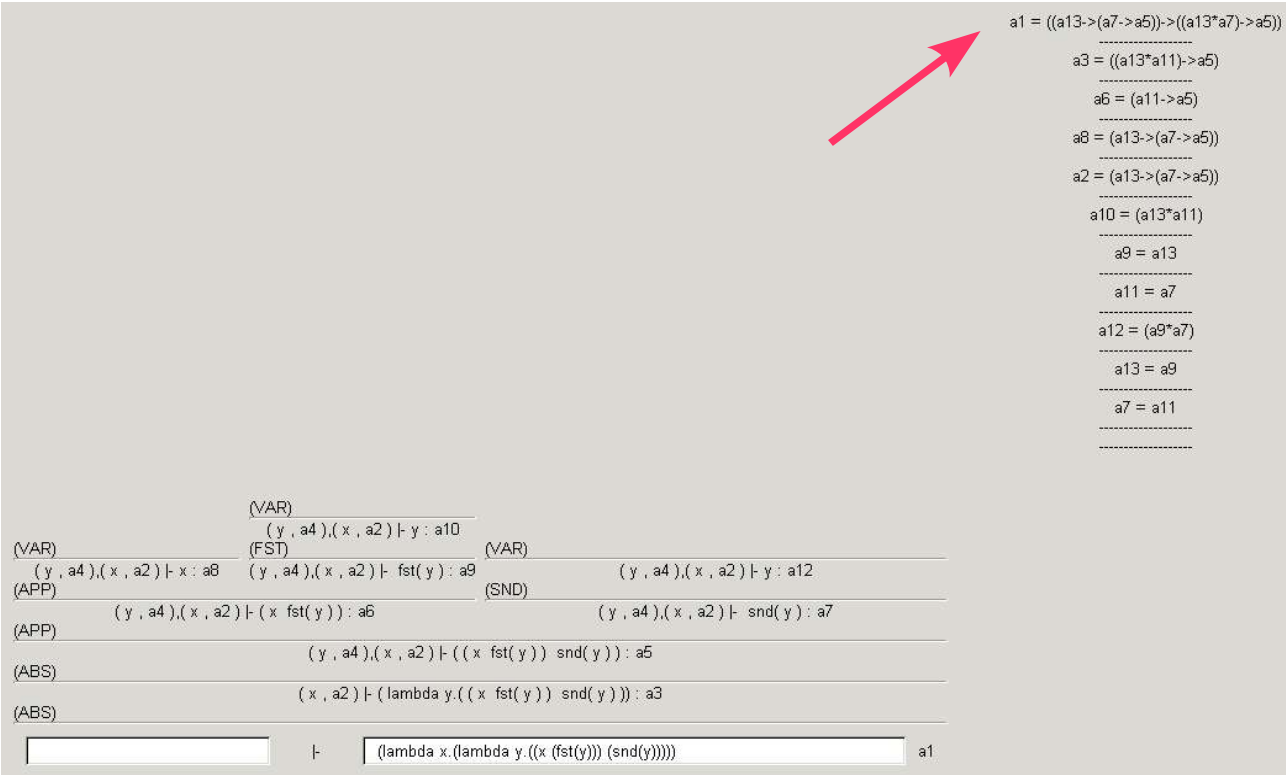


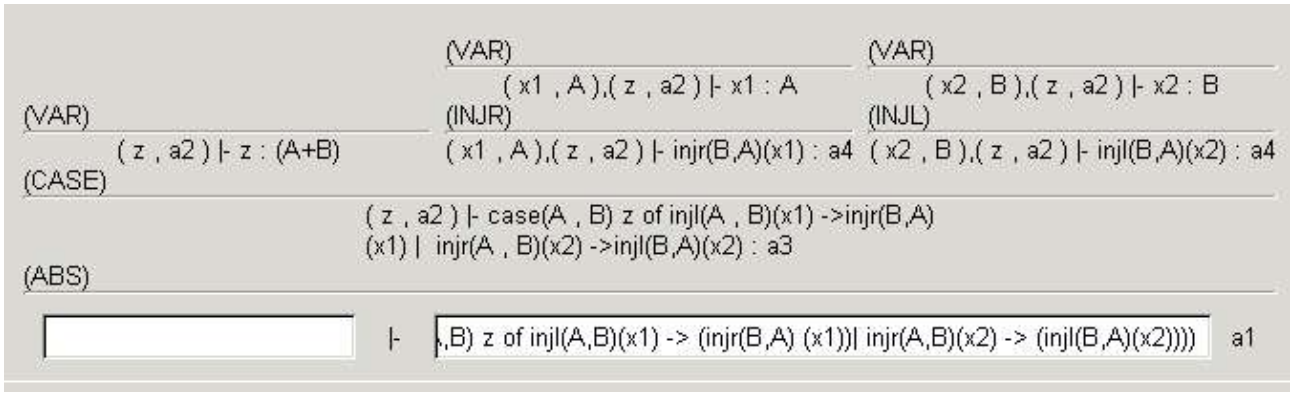
Illustration 4 : Exemple de typage, a1 est le type final

4.Comment illustrer l'isomorphisme de Curry-Howard

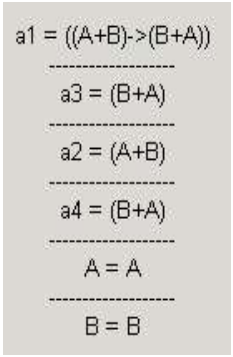
Etant donnés une formule de logique propositionnelle et un λ-terme, la preuve de la formule et le typage du λ-terme produisent, séparément, deux arbres de preuve. L'étudiant peut donc vérifier si les règles utilisées pour prouver la proposition logique trouvent, une à une, leur symétrique dans les règles de typage. Dans ce cas, il pourra conclure que le λ-terme donné correspond à la preuve de la proposition logique qu'il a entrée. Il aura donc obtenu deux arbres de preuve équivalents, donc un programme permettant de prouver sa formule logique.

Par exemple, le typage du λ-terme :

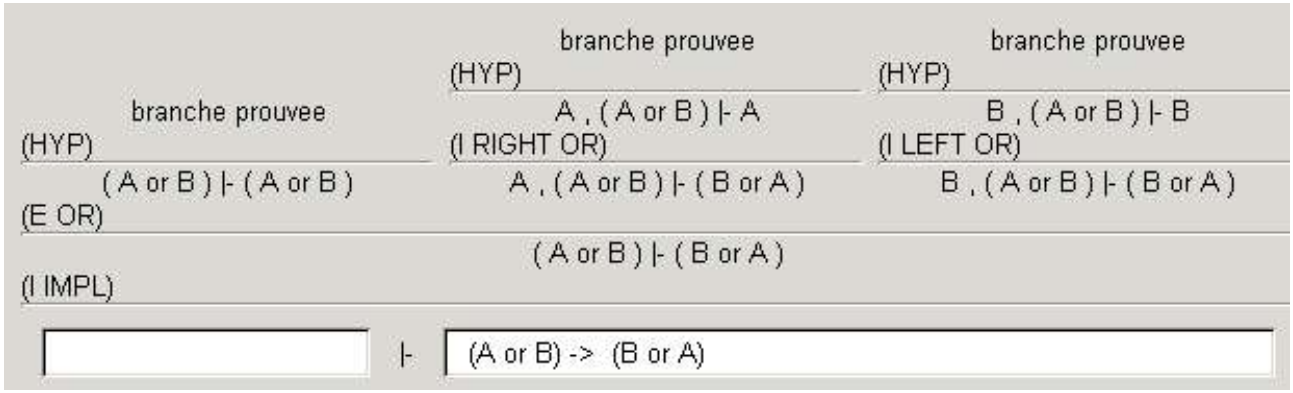
λz. case$_{A, B}$ z of inj$^{l}_{A, B}$ (x$_1$) → inj$^{r}_{B, A}$ (x$_1$) | inj$^{r}_{A, B}$ (x$_2$) → inj$^{l}_{B, A}$ (x$_2$) produit l'arbre suivant :



et le système de contraintes de type :



La preuve de la proposition : *A V B → B V A* produit quant à elle :



Les règles de déduction et de typage correspondent (I IMPL↔ABS, E OR↔CASE, VAR↔HYP, I RIGHT OR↔INJR, I LEFT OR↔INJL). *λz. case$_{A, B}$ z of inj$^{l}_{A, B}$ (x$_1$) → inj$^{r}_{B, A}$ (x$_1$) | inj$^{r}_{A, B}$ (x$_2$) → inj$^{l}_{B, A}$ (x$_2$)* est donc le λ-terme correspondant à la preuve de *A V B → B V A*, ce qui est cohérent avec le type *a1* trouvé par notre typeur.

Conclusion

Nous avons donc implanté deux systèmes interactifs. Notre assistant à la preuve permet à l'étudiant de choisir les règles de déduction de la logique propositionnelle étendue, vue en cours de Logique, pour prouver une formule. Notre outil de typage accompagne pas à pas l'étudiant qui veut typer un λ-terme en résolvant le système de contraintes sur les types engendré par son raisonnement.

L'élaboration de ces deux outils a mobilisé nos connaissances sur le principe des analyses lexicale et syntaxique et l'élaboration d'une grammaire efficace. Nous avons découvert la richesse du langage Ocaml pour l'analyse de programmes et les spécificités de ce langage de programmation, en particulier le fonctionnement du contrôleur de types Ocaml qui nous permet d'étudier plus facilement la cohérence d'un programme et son sens, ainsi que la grande proximité du langage Ocaml avec le langage mathématique.

Nous avons pu évaluer l'étendue des recherches parties du résultat de Curry-Howard établissant la correspondance entre preuves et programmes. Ce faisant, nous nous sommes interrogées sur les possibilités de mécaniser en partie le raisonnement mathématique et transformer ainsi les ordinateurs en outils d'aide à la démonstration de théorèmes.

Une extension à notre projet consisterait à élaborer un troisième outil permettant à l'étudiant de construire un arbre de preuve à l'aide des règles de la logique propositionnelle, puis de « décorer » cet arbre avec des λ-termes, afin de vérifier plus directement que le λ-terme finalement obtenu est typable. Ceci offrirait un point de vue différent permettant de montrer tout aussi bien l'isomorphisme de Curry-Howard.

Annexe 1

Syntaxe concrète entrée par l'étudiant	***Syntaxe abstraite lisible par le noyau de l'assistant***	***Règle de déduction applicable***
A...Z	SINGLE (s)	(Hyp), (E ⇒), (E^l Λ), (E^r Λ), (E V), (E ⊥)
A -> B	IMPLIC (f1, f2)	(I ⇒)
A and B	AND (f1, f2)	(I Λ)
A or B	OR (f1, f2)	(I^l V), (I^r V)
false	FALSE	(E ⊥)

Syntaxe et sémantique pour l'assistant à la preuve – Tableau récapitulatif

Annexe 2

Syntaxe concrète	***Syntaxe abstraite***	***Règle de typage applicable***
a...z, A...Z	V (Var v)	(Var)
(t1 t2)	App (t1, t2)	(App)
(lambda x . t)	Abs (Var v, t)	(Abs)
(t1, t2)	Pair (t1, t2)	(Pair)
(fst (t))	Fst t	(Fst)
(snd (t))	Snd t	(Snd)
(injl (to1, to2) (t)) où to1 et to2 sont de la forme : a, (a + b), (a * b) ou (a -> b)	Injl (to1, to2, t) où to1 et to2 sont de la forme : SINGLE a, IMPL (a,b), PRODUIT (a,b) ou SOMME (a,b)	(Injl)
(injr (to1, to2) (t)) où to1 et to2 sont de la forme : a, (a + b), (a * b) ou (a -> b)	Injr (to1, to2, t) où to1 et to2 sont de la forme : SINGLE a, IMPL (a,b), PRODUIT (a,b) ou SOMME (a,b)	(Injr)
(case (to1, to2) t of injl (to1, to2) (v1) -> t1 | injr (to1, to2) (v2) -> t2) où to1 et to2 sont de la forme : a, (a + b), (a * b) ou (a -> b)	Case (to1, to2, Var v1, Var v2, t, t1, t2) où to1 et to2 sont de la forme : SINGLE a, IMPL (a,b), PRODUIT (a,b) ou SOMME (a,b)	(Case)

Syntaxe et sémantique pour l'outil de typage – Tableau récapitulatif

Répartition du travail

Principales étapes de développement	***Echéance***
Documentation sur les outils existants d'aide à la preuve, collecte des résultats théoriques sur la logique propositionnelle et le typage de λ-termes	*13 février – 27 février*
Codage du noyau de l'assistant à la preuve et apprentissage approfondi du langage Ocaml	*28 février – 27 mars*
Codage de l'interface de l'assistant à la preuve et apprentissage de la syntaxe de LablGTK2	*28 mars – 4 mai*
Rédaction d'un pré-rapport et préparation de la présentation de mi-parcours	*5 mai – 9 mai*
Pré-soutenance	***10 mai 2006***
Documentation complémentaire sur le langage Caml et ses fondements théoriques, sur les conseils de nos encadrants	*15 mai – 20 mai*
Codage des analyseurs lexical et syntaxique pour l'outil de typage	*21 mai – 6 juin*
Documentation complémentaire sur la résolution de systèmes de contraintes et compréhension des algorithmes d'unification	*7 juin – 11 juin*
Codage de la résolution du système de contraintes sur les types pour l'outil de typage et son affichage	*12 juin – 21 juin*
Rédaction du rapport final et préparation à la soutenance	*22 juin – 25 juin*
Recherches supplémentaires et début de codage d'un outil permettant de mettre directement en évidence l'isomorphisme de Curry-Howard	*26 juin – 29 juin*
Soutenance finale	***29 – 30 juin 2006***

Bibliographie

[biblio0]
Manuel Ocaml en ligne : http://caml.inria.fr/pub/docs/manual-ocaml/index.html
nous a été indispensable pour comprendre les structures de données spécifiques à Ocaml, les outils d'analyse lexicale et syntaxique, le fonctionnement général du langage.

[biblio1]
Transparents du cours de Mathieu Jaume : http://www.liafa.jussieu.fr/~ig/M1/coursM.pdf

[biblio2]
Articles de Jean-Louis Krivine : http://www.pps.jussieu.fr/~krivine/
une partie de ces articles a été conseillée par Mathieu Jaume au premier semestre pour culture générale.

[biblio3]
Livre d'André Arnold et Irène Guessarian : Mathématiques pour l'Informatique, 3ème édition, Masson, 2000
nous convenait pour sa formulation simple et pédagogique des principaux résultats de la logique et de la théorie des ensembles.

[biblio4]
Page Web de LablGTK2 et manuel utilisateur : http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html

[biblio5]
Livre d'Emmanuel Chailloux, Pascal Manoury et Bruno Pagano : Développement d'Applications avec Objective Caml, O'Reilly, 2000
nous l'avons utilisé en complément du manuel Ocaml en ligne.

[biblio6]
Livre de Pierre Weis et Xavier Leroy : Le langage Caml, 2ème édition, Dunod, 1999
conseillé par nos encadrants comprendre les spécificités du langage Caml et leurs motivations.

[biblio7]
Articles de Gilles Dowek : *Le langage mathématique et les langages de programmation*, Voir, entendre, raisonner, calculer, Cité des sciences et de l'industrie, La Villette, Paris, 1997 http://www.lix.polytechnique.fr/~dowek/Vulg/langagelangages.pdf
Le sens du calcul, Exposé au séminaire « Qu'est-ce qu'une logique ? » à la Sorbonne, le 9 Avril 1996 http://www.lix.polytechnique.fr/~dowek/Vulg/calcul.pdf