Stagiaire Nicolas Pouillard
Spécialisation CSI
Promotion 2006

Rénovation de Camlp4, un Préprocesseur Pretty-Printeur pour Caml

Ce stage de fin de spécialisation a été effectué sous la direction de Michel Mauny au sein de l'équipe Gallium de l'INRIA Rocquencourt du 3 Janvier au 30 Juin 2006.

Table des matières

Chapitre 1

Résumé du stage

Mon stage s'est déroulé de Janvier à Juin 2006, au sein de l'équipe Gallium à l'INRIA-Rocquencourt. Ce stage a été encadré par Michel Mauny (Directeur de Recherche INRIA, en détachement à l'ENSTA) sur le logiciel Camlp4. L'objectif de ce stage était de redonner un souffle nouveau à ce projet qui n'avait pas subi d'évolution depuis longtemps.

Camlp4 est un préprocesseur *pretty-printeur* pour Objective Caml. Ce logiciel propose des notations pour simplifier la manipulation de syntaxe abstraite (*quotations*) et des *pretty-printeurs*. Mais surtout Camlp4 est connu pour son mécanisme de grammaires extensibles, qui permet entre autres d'effectuer des extensions de syntaxe pour le langage Caml. Lors de mon stage, je devais rénover ce logiciel afin faciliter sa maintenance future.

J'ai commencé par travailler sur les *locations* et le *bootstrap*, deux éléments de Camlp4 qu'il était prévu que je rénove. J'ai ensuite réécrit l'analyseur lexical et un *pretty-printeur*. Puis a eu lieu une réorganisation totale des modules et un renommage. Enfin, j'ai ajouté de nouvelles fonctionnalités : extension du système des grammaires extensibles pour supporter des motifs de lexèmes arbitraires, et refonte du mécanisme de *quotations* pour rendre la syntaxe des *quotations* aussi extensible que la syntaxe du langage lui-même.

L'environnement dans lequel s'est déroulé mon stage m'a beaucoup plu. Gallium, l'équipe qui m'a accueilli, est très sympathique. J'apprécie d'être au contact avec des chercheurs, et de pouvoir avoir avec eux des conversations vivement intéressantes. De plus, mon maître de stage, Michel Mauny, m'a vraiment bien encadré et nous nous sommes très bien entendus. Nous allons continuer à travailler ensemble dans le cadre d'un contrat d'ingénierie sur le projet de compilateur reFLect [Intel, 2006].

Chapitre 2

Introduction

Ce rapport présente les travaux réalisés au cours de mon stage de fin de spécialisation. Ce stage a été effectué à l'INRIA-Rocquencourt, sous la direction de Michel Mauny. Il s'est déroulé au sein de l'équipe Gallium, qui est en charge du projet Caml. Durant six mois, j'ai travaillé à la rénovation du sous-système Camlp4. Camlp4 est un préprocesseur *pretty-printer* pour le langage Caml. Cette rénovation comportait différents axes : rendre le logiciel plus facile à maintenir, l'adapter aux nouvelles technologies de Caml, améliorer la documentation et les tests, clarifier les interfaces, et ajouter de nouvelles fonctionnalités. Ce travail aboutit à une nouvelle version de ce logiciel, qui sera intégrée prochainement à la distribution officielle d'Objective Caml.

Après avoir énoncé le sujet du stage, nous expliquerons ses finalités et son extension. Ensuite nous présenterons l'INRIA ses perspectives et son organisation. Par la suite nous introduirons le projet Gallium (anciennement Cristal), son rôle dans l'unité de recherche et son activité. Viendra ensuite le projet Caml qui est développé au sein du projet Gallium. On expliquera ensuite la place du projet Camlp4 dans Caml. Enfin le sujet sera replacé dans son contexte.

2.1 Sujet du stage

Voici le sujet de stage, tel qu'il a été validé par le responsable de ma spécialisation :

«Rénovation du sous-système Camlp4 d'Objective Caml. Camlp4 est un préprocesseur programmable pour le langage Objective Caml. Après des évolutions successives, Camlp4 a maintenant besoin d'être ré-architecturé. L'objectif premier du stage est donc de revoir cette architecture, et de simplifier et documenter l'usage et le développement de Camlp4. Si le temps

le permet, on pourra procéder à une refonte partielle de ses interfaces et à réimplémenter le code correspondant de sorte à ce que CamlP4 soit plus résistant à des extensions ultérieures.»

L'objectif premier du stage était de rendre CamlP4 plus maintenable, et de finaliser cette nouvelle version par son intégration au cœur d'Objective Caml. Dans ce but, nous devions chercher les points problématiques du logiciel nuisant à sa maintenance. Ainsi, plusieurs parties étaient à réécrire, d'autres seulement à simplifier. Il fallait aussi remanier l'architecture de CamlP4 pour mieux y intégrer certaines fonctionnalités, ajoutées superficiellement. Cette nouvelle cohérence permettrait de rendre CamlP4 plus résistant aux nouvelles évolutions.

Le sujet était assez libre face à la profondeur des changements effectués. Le début de stage, et les premiers travaux étant concluant, cela nous à permis d'envisager une refonte plus profonde. En effet au fur et à mesure du temps, la nouvelle mouture de CamlP4 à pris forme. C'est finalement un changement majeur pour le projet CamlP4, il reste maintenant à officialiser cette nouvelle version en l'intégrant à la version stable d'Objective Caml.

2.2 Contexte de l'entreprise

L'INRIA, institut national de recherche en informatique et en automatique, a pour but d'entreprendre des recherches fondamentales et appliquées dans les domaines des sciences et technologies de l'information et de la communication. Il vise aussi à développer le transfert technologique, cela passe par la formation par la recherche, par la diffusion de l'information scientifique et technique, mais aussi par la participation à des programmes internationaux.

L'INRIA se place donc sur le plan mondial, comme un institut de recherche au cœur de la société de l'information. Sa concurrence s'établit donc à l'échelle nationale et internationale.

L'INRIA dispose de six unités de recherche situées à Rocquencourt, Rennes, Sophia Antipolis, Grenoble, Nancy et Bordeaux, Lille, Saclay. Elle accueille en tout 3 600 personnes dont 2 800 scientifiques, issus du CNRS, des universités, et des grandes écoles. Ils travaillent dans plus de 138 projets de recherche communs. Un grand nombre de chercheurs de l'INRIA sont également enseignants et leurs étudiants (environ 1 000) préparent leur thèse dans le cadre des projets de recherche de l'INRIA.

À l'INRIA, les recherches s'organisent en «projets». Les projets sont regroupés en cinq grands thèmes : systèmes communicants, systèmes cognitifs, systèmes symboliques, systèmes numériques, et systèmes biologiques. Ces thèmes sont tous représentés à l'INRIA Rocquencourt. Dans cette unité de recherche, il y a quarante projets dont le projet Gallium.

Le projet Gallium (successeur du projet Cristal), dirigé par Xavier Leroy, a pour thèmes les langages de programmation, les types, la compilation et les preuves. Les travaux de recherche de cette équipe portent sur la conception, la formalisation et l'implémentation de langages et systèmes de programmation. Le langage Caml regroupe de nombreux résultats de leurs recherches. L'objectif est d'améliorer la fiabilité (sûreté de fonctionnement et sécurité) des logiciels en utilisant :

- des langages de programmation de plus haut niveau, plus sûrs et plus expressifs, basés sur le paradigme de la programmation fonctionnelle ;
- la détection automatique d'erreurs de programmation à l'aide de systèmes de types et d'autres analyses statiques ;
- une meilleure intégration de la programmation et des méthodes formelles, en particulier la preuve de programme, et surtout les preuves sur machine [Bertot and Castéran, 2004].

2.3 Etat des connaissances sur le sujet

2.3.1 Dans l'entreprise

Cela va sans dire que l'INRIA a des connaissances sur le sujet, surtout l'équipe Cristal qui a fondé le projet Caml puis le projet Camlp4. Ce projet a débuté avec mon maître de stage (Michel Mauny) et un ingénieur de recherche (Daniel de Rauglaudre). Cependant cet ingénieur ne fait maintenant plus partie de l'équipe. Le projet perdant son principal développeur, se voit maintenu par une seule personne. La connaissance théorique sur les concepts manipulés dans Camlp4 sont largement connus par l'équipe, par contre les connaissances sur certains points techniques du code et de sa spécification sont en déclin.

2.3.2 Chez le stagiaire

Mon apprentissage du Caml a débuté dès la première année d'ÉPITA (info-sup) avec les cours sur ce langage (Nathalie Bouquet et Christophe Boullay). Par la suite en deuxième année (info-spé), j'ai dirigé et réalisé avec mon équipe le projet *EpiCash*. J'ai décidé qu'il soit entièrement écrit en Caml. Ce projet montrait déjà mon intérêt pour la syntaxe, car j'utilisais déjà Camlp4 via la syntaxe révisée de Caml, et d'autres extensions de syntaxe dont une développée pour *EpiCash*. En première année d'ingénierie

Julien Roussel et moi-même avons donné un cours et des exercices corrigés de Caml aux ING1 avant le projet qu'ils devaient effectuer en Caml. Au LRDE j'ai pris place au sein du projet *Transformers* qui a pour but de fournir un environnement de transformation de programmes *C++*. Mon orientation est aussi due a certains cours du cycle ingénierie de l'ÉPITA comme les cours de théorie des langages, de compilation et de logique formelle dispensés par Akim Demaille. Ces projets et cours m'ont beaucoup intéressé et recoupent les thématiques du stage. Ils m'ont aidé pour sa réalisation.

2.4 Contexte précis de travail

Je suis très satisfait du contexte de travail dans lequel j'ai pu faire mon stage. L'unité de recherche de l'INRIA Rocquencourt est située dans un sympathique cadre de verdure à proximité de Paris et Versailles. Le bâtiment dans lequel s'est déroulé mon stage regroupe deux projets : Gallium et Aoste, et il est très calme et bien situé. Le seul défaut de ce cadre sympathique est sans doute le temps d'accès. Heureusement un service de navettes est organisé. D'un point de vue purement matériel, les moyens mis à ma disposition étaient suffisants. Le bureau dans lequel j'étais était très agréable et une machine m'a été prêtée, que j'ai pu installée comme bon me semblait. Dans le bureau que j'occupais, j'ai rencontré Alain Frisch, un jeune chercheur très sympathique avec qui j'ai eu des discussions passionnantes. Le repas était aussi un moment privilégié, où les discussions ne manquent pas et les repas sont de qualité.

Michel Mauny, mon maître de stage, travaille maintenant essentiellement à l'ENSTA. Dans l'équipe Gallium, il intervient en ce moment sur principalement deux projets dont Camlp4. Nous nous sommes vus régulièrement, en moyenne deux jours par semaine. À chaque fois nous faisions le point sur l'avancée du projet, discutions des tâches en cours. On abordait aussi d'autres points au fur et à mesure que des nouveaux changements semblaient nécessaires. Je suis très heureux de travailler avec Michel, avec lui j'ai beaucoup appris à propos du langage Caml, de son histoire, de ses détails, et aussi de la désérialisation [Henry et al., 2006], du typage, etc.

Ce cadre est très agréable et donne envie de travailler. C'est un environnement dans lequel j'aurai plaisir à travailler plus tard.

Chapitre 3

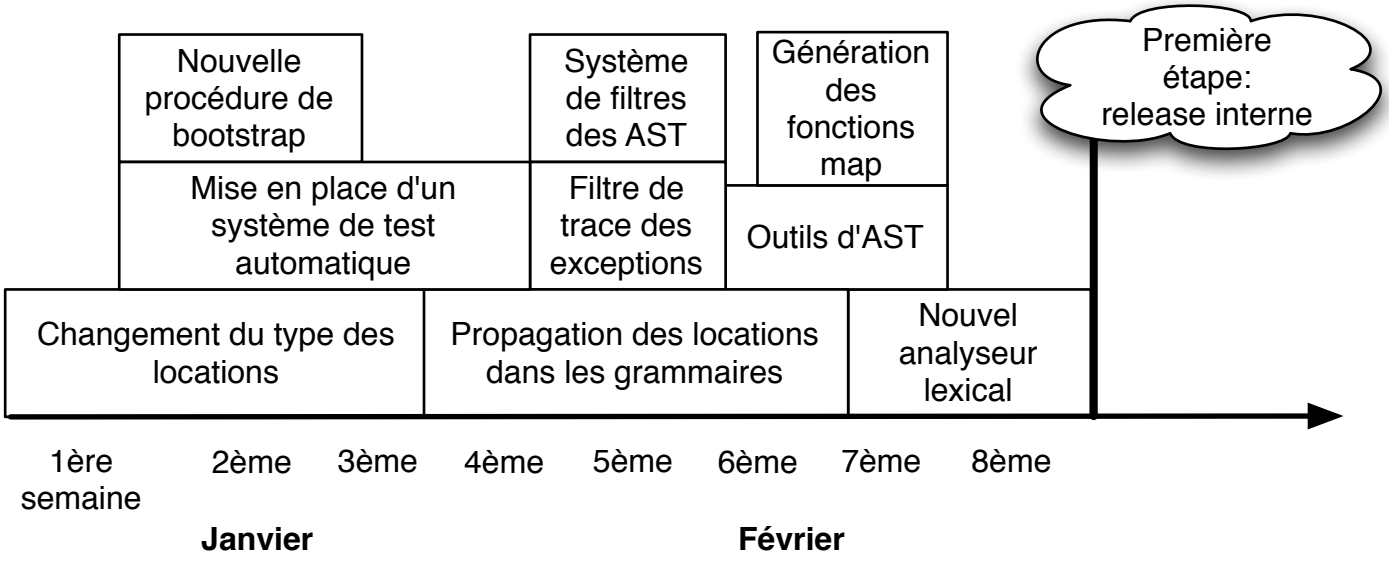
Aspects organisationnels

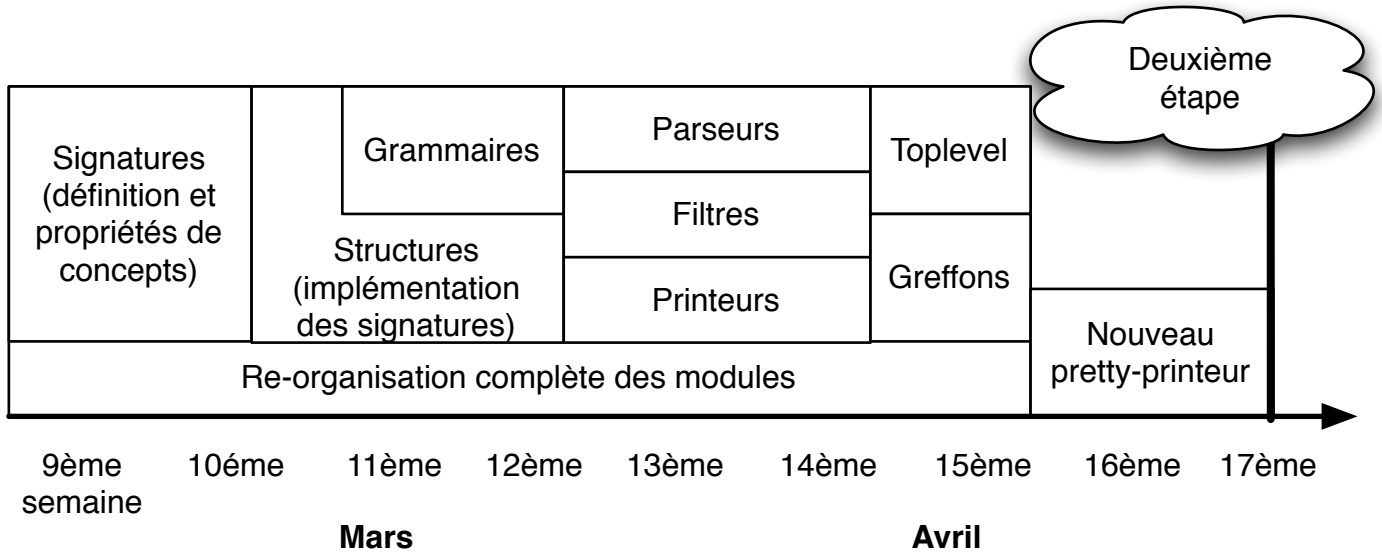
D'un point de vue organisationnel, le stage a été assez libre. Au départ j'ai commencé avec des tâches très précises comme le *bootstrap* et les *locations*. Par la suite, l'organisation s'est établie dans une forme interactive, c'est-à-dire que l'on discutait d'un changement commencé ou à venir pour choisir de l'intégrer ou non de manière officielle. Nous avons tout de même souhaité obtenir des «délivrables» intermédiaires, même s'ils sont restés très locaux au groupe Gallium.

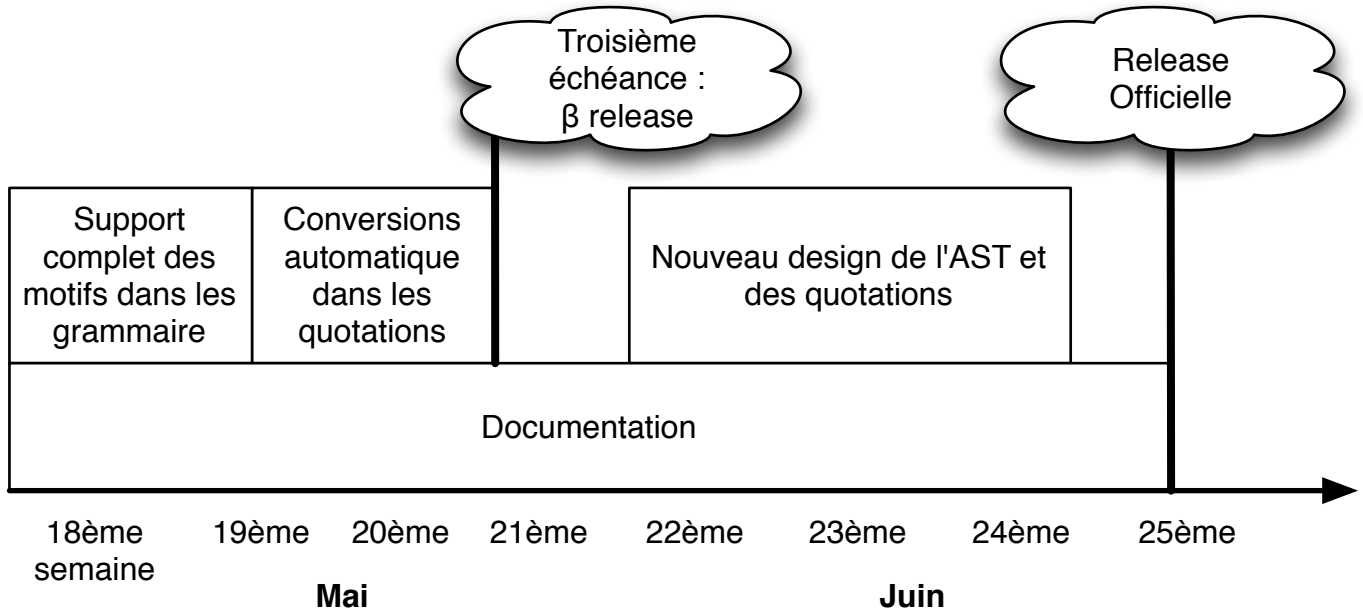
3.1 Découpage du projet

Comme indiqué précédemment, le projet a suivi un découpage assez interactif, non pas parce que l'on ne savait pas où aller mais plutôt parce qu'il fallait chercher quelles modifications apporteraient vraiment un intérêt à Camlp4. Cependant nous avons tenu à placer plusieurs étapes, jalonnant l'évolution du stage.

À titre d'illustration la page suivante contient un diagramme qui retrace les différentes étapes, il permet notamment d'observer les tâches en série et en parallèle, ainsi que les étapes intermédiaires.


Nouvelle procédure de bootstrap
Système de filtres des AST
Génération des fonctions map
Première étape: release interne
Mise en place d'un système de test automatique
Filtre de trace des exceptions
Outils d'AST
Changement du type des locations
Propagation des locations dans les grammaires
Nouvel analyseur lexical
1ère semaine
2ème
3ème
4ème
5ème
6ème
7ème
8ème
Janvier
Février


Deuxième étape
Signatures (définition et propriétés de concepts)
Grammaires
Parseurs
Toplevel
Filtres
Structures (implémentation des signatures)
Greffons
Printeurs
Nouveau pretty-printeur
Re-organisation complète des modules
9ème semaine
10éme
11ème
12ème
13ème
14ème
15ème
16ème
17ème
Mars
Avril


Troisième échéance : β release
Release Officielle
Support complet des motifs dans les grammaire
Conversions automatique dans les quotations
Nouveau design de l'AST et des quotations
Documentation
18ème semaine
19ème
20ème
21ème
22ème
23ème
24ème
25ème
Mai
Juin

La première étape contenait le travail sur les *locations* (section 4.3) et la procédure de *bootstrap* (section 4.2). Pour mener à bien cette étape, je suis passé par le développement d'outils pour les AST, la génération des fonctions map (section 4.5.3) et le mécanisme de filtres (section 4.5).

À cette étape, Michel a accepté que j'ajoute le travail sur l'analyseur lexical (section 4.4) puisque le changement était isolé et bien précis. Pour valider cette étape, l'objectif était de mettre à jour en conséquence COQ [Bertot and Castéran, 2004], qui utilise largement CAMLP4.

Une fois cette étape réalisée, des problèmes de conflits de noms sont apparus. Nous avons décidé de répondre à ce problème d'espaces de nommage (section 4.7). CAMLP4 a donc été placé dans un espace de noms qui lui est propre et par la même occasion, un grand renommage et une réorganisation ont été effectués. Cette réorganisation a été accompagnée par l'écriture d'un nouveau *pretty-printeur* (section 4.6). Cette partie a elle aussi été validée par la mise à jour de COQ (beaucoup plus longue pour cette partie).

Dans la troisième partie de mon stage, je me suis attaché à étendre réellement CAMLP4, en y ajoutant des fonctionnalités significatives comme le support complet des motifs de lexèmes (section 4.8), des conversions automatiques dans les *quotations*, et surtout en fournissant un gros travail sur les nouvelles *quotations* bootstrappées (section 4.9). Cette dernière partie se terminera par l'intégration de CAMLP4 dans la distribution officielle d'Objective CAML, lors de sa prochaine version.

3.2 Respect du découpage

Les délais et le découpage ont été respectés, même si le «respect du découpage» n'est pas trop adapté ici, car le découpage a été étudié par mon maître de stage et moi. Pour cette même raison je n'ai aucune remarque à faire sur la pertinence de ce découpage.

Les points de contrôle en interne étaient suffisants, même s'ils étaient effectués d'une manière un peu informelle. Je pense qu'il aurait été judicieux de placer plus de points de contrôle sur les modifications réalisées, cela aurait peut-être permis de gagner en efficacité. Mais cela représente du temps et je pense que Michel a déjà passé beaucoup de temps pour m'épauler dans ce projet.

Nous n'avons pas eu besoin de recourir à des procédures du type «emergency», car il n'y avait pas de réelle deadline autre que celle de la fin du stage et de la release officielle qui est en très bonne voie.

Chapitre 4

Aspects scientifiques et techniques

4.1 Introduction

4.1.1 Le langage Caml

Le langage Caml est un langage de programmation faisant partie de la famille ML. C'est un langage généraliste, qui a pour objectif de renforcer la sûreté des programmes, en procédant à plus de vérifications statiques qui permettent de trouver des erreurs dans le programme avant son exécution. Le langage Caml est un langage fortement typé, aux multiples paradigmes. C'est particulièrement vrai dans la distribution Objective Caml, qui inclut un système puissant de modules et de foncteurs permettant la réutilisabilité et la programmation générique. On peut aussi définir des signatures et forcer la vérification de certaines propriétés. OCaml propose ainsi un style fonctionnel en premier lieu, mais aussi des styles impératif et orienté objet. Objective Caml permet de s'abstraire de la machine, du système d'exploitation, et des erreurs dans cette optique il dispose d'un récupérateur automatique de mémoire, et d'un puissant système d'exceptions.

4.1.2 Camlp4

Camlp4 est un préprocesseur, *pretty-printeur* pour Objective Caml. Il permet d'écrire des parsers, des grammaires et de manipuler des programmes à l'aide des *quotations*. Camlp4 fournit en somme un environnement pour manipuler des programmes d'un point de vue syntaxique.

Ce qui a fait le succès de Camlp4, c'est son système de grammaire extensible. Il permet aux utilisateurs du langage Caml d'étendre la syntaxe de Caml et d'y ajouter des nouvelles constructions, des notations, des langages

spécifiques à un domaine, etc. Pour créer une extension de syntaxe, on utilise justement une extension de syntaxe ajoutant un langage de description de grammaire. Ainsi on ajoute ou supprime des règles et des entrées à une grammaire comme celle de Caml. Le rôle d'une extension de syntaxe devient donc de traduire ces nouvelles constructions à l'aide des constructions standards de Caml. Pour faciliter l'écriture d'arbres de syntaxe abstraite, un système de *quotations* permet d'écrire ces arbres simplement à l'aide d'une syntaxe concrète.

En effet, cette syntaxe concrète n'est pas la syntaxe originale de Caml, mais la syntaxe dite révisée. Cette syntaxe est une variante de Caml qui a plusieurs objectifs : fermer les constructions syntaxiques qui ne le sont pas suffisamment (`match`, `try`, etc.), supprimer les constructions redondantes (`begin ... end` est équivalent à `(...)`), et pallier à d'autres défauts dus à l'histoire de Caml. Dans la syntaxe révisée, les constructeurs, les types et les modules sont dits curryfiés, à la manière des fonctions. Par exemple, `Foo(x,y)` devient `Foo x y`, `Set.Make(String).t` devient `(Set.Make String).t` et `(`α`, `β`) Hashtbl.t` devient `Hashtbl.t` α β. Finalement la syntaxe révisée est plus régulière et moins ambiguë, ce qui apporte de bons fondements à la création d'un système de *quotations*, qui suscite l'utilisation de fragments de syntaxe.

Camlp4 fournit aussi plusieurs *pretty-printeurs*. Un *pretty-printeur* permet d'afficher un arbre de syntaxe abstraite sous forme de syntaxe concrète. Cet affichage est bien présenté : indentation, respect d'une largeur maximale, style horizontal et vertical, restauration des commentaires, etc. Ainsi Camlp4 peut servir de traducteur source-à-source. Par exemple, en combinant le *parseur* de la syntaxe originale et le *pretty-printeur* de la syntaxe révisée, on obtient un traducteur de syntaxe originale vers révisée. Ce traducteur est fort utile à l'apprentissage et à la transition vers la syntaxe révisée.

Tout au long de ce chapitre, des morceaux de code sont présentés pour clarifier les propos énoncés. Ces parties de code ne font pas partie du projet Camlp4. Ces figures représentent des entrées et des sorties du logiciel, qui l'illustrent, au même titre que des impressions d'écran illustrent une interface graphique.

4.2 Simplification du système de *bootstrap*

Camlp4 est un préprocesseur *bootstrapé*, c'est-à-dire qu'il est nécessaire à sa propre construction. En d'autres termes, Camlp4 peut être utilisé pour

traduire un programme écrit dans la syntaxe *révisée* de Caml vers un programme équivalent dans la syntaxe originale de Caml. Camlp4 est le seul outil à fournir le support de cette syntaxe *révisée*. Camlp4 étant lui même écrit en *révisé*, il est nécessaire, afin de construire Camlp4, de disposer d'une version déjà construite de Camlp4. Bien entendu la première itération de cette boucle de *bootstrap* a dû être initialisée différemment. En effet la première version de Camlp4 était écrite dans la syntaxe originale et fut traduite automatiquement une fois le premier traducteur fonctionnel.

Le mécanisme de *bootstrap* existant reposait sur un ensemble de sources en syntaxe originale qui est généré à partir des sources réelles. Ces sources sont écrites en syntaxe *révisée*. Le mécanisme de construction était le suivant : compilation des sources en syntaxe originale, utilisation du programme produit pour traduire et compiler les sources en syntaxe *révisée* et donc produire le programme final. En fait il existait quelques raccourcis pour ne pas avoir à gérer de telles reconstructions systématiques.

Le problème était que seules quelques personnes comprenaient ce mécanisme. Sans rentrer dans les détails, il était en effet trop complexe d'arriver à *bootstraper* Camlp4 une fois les changements nécessaires appliqués.

4.2.1 Choix techniques possibles

Une première sorte de solution était de tout simplement mieux expliquer ce système qui paraît obscur surtout parce qu'il n'est pas expliqué en détail dans un manuel réservé aux développeurs. Disposer de plus d'explications et d'exemples aiderait déjà grandement ceux qui tentent d'apporter une contribution à Camlp4 et sont confrontés au problème de *bootstrap*. D'autres formes de solutions étaient envisageables mais aucune n'avait été évoquée.

4.2.2 Solution retenue, justification

L'idée principale de la solution retenue est d'arriver à se passer des sources générées (en syntaxe originale) permettant de «démarrer» le *bootstrap*. Les inconvénients de ces sources générées sont :

- pollution introduite par le fait d'avoir une partie des sources en double, c'est particulièrement désagréable en ce qui concerne la recherche dans les sources, les correctifs soumis au gestionnaire de versions sont également en double ...
- dans certains cas le meilleur moyen de réussir certains *bootstraps* difficiles était de modifier temporairement ces sources. Ce type de manipulation impliquait de changer par la suite de la même manière les sources en syntaxe *révisée*.

Il a donc été décidé de créer un programme exécutable avec le minimum de fonctionnalités nécessaires à la compilation de toutes les sources ; et non pas un mélange complexe où chaque fichier a besoin de certains modules. Ici l'objectif était de fournir une plus grande facilité d'emploi sans y perdre l'essentiel. Ce nouvel exécutable, appelé `camlp4boot`, permet donc de compiler entièrement la partie distribuée de Camlp4, ce qui est un progrès en terme de clarté du système.

4.2.3 Problème rencontré

Le seul réel problème rencontré a été celui de la compréhension du système en lui-même. Cette compréhension était nécessaire pour pouvoir créer `camlp4boot` de façon à ce qu'il contienne toutes les extensions de syntaxe nécessaires à l'ensemble des sources de Camlp4. Bien que tous les fichiers n'aient pas besoin des mêmes extensions, la création de `camlp4boot` n'a pas été si difficile car il n'y a pas eu d'incompatibilités à utiliser le même préprocesseur.

4.3 Abstraire et corriger les *locations*

Dans un préprocesseur comme dans un compilateur il est agréable pour l'utilisateur d'avoir une indication précise d'où se situe une erreur. C'est l'objectif du système de *locations* qui permet d'étiqueter les représentations internes du fichier source par une indication de position. Ainsi lorsqu'une erreur est détectée il suffit d'utiliser en général la *location* la plus accessible pour afficher un message clair.

4.3.1 Historique des *locations* dans Camlp4

La représentation des *locations* a subi des changements majeurs en Caml en 2004. C'est pour ce type de raison que le stage a pris forme. Le concept de *location* peut généralement être vu comme la zone placée entre deux *positions*. Une *position* est alors une indication plus ou moins complète d'un endroit dans le fichier source.

C'est ainsi que la première représentation d'une *location* était un couple d'entiers. Chaque entier représentait la position dans le flot de caractères. Le nom du fichier était stocké dans une référence globale. De son côté le compilateur OCaml utilisait une structure de données plus fine permettant de conserver le nom du fichier, le numéro de la ligne, la position dans le flot de caractères et aussi la position du début de la ligne. Toutes ces informations permettent un affichage comme celui-ci :

Pour obtenir ces informations l'ancienne version de Camlp4 avait recours à une deuxième lecture du fichier pour compter les lignes comme le

```
File "foo", line 10, characters 14-25:
Syntax error
```

faisait OCaml avant. En quelque sorte CAMLP4 avait une technologie de retard par rapport à OCaml. Ce couple d'entiers est devenu un couple de `Lexing.position` depuis la version 3.08, ce qui permet de se passer de la relecture du fichier source en cas d'erreur, ainsi que des références globales. Cependant les références sont restées, ce qui n'a donc pas simplifié le système.

4.3.2 Abstraction

Pour éviter de continuer dans cette direction j'ai décidé d'abstraire ces *locations*, d'identifier les opérations sur ces *locations*, et d'en définir une signature de module permettant une grande souplesse d'utilisation et d'implémentation.

Ainsi la signature comporte des fonctions de construction, d'accès pour chaque information disponible, de conversion vers d'autres représentations, de fusion de *locations*, et de déplacements.

Les *locations* n'étaient pas bien véhiculées d'étape en étape ce qui impliquait le déplacement des *locations* de tout un arbre. Ces opérations sont particulièrement lourdes à mettre en place. De plus les erreurs dans le calcul des décalages à effectuer sont fréquentes. Prenons par exemple l'expression *print_expr* <:*expr*< $42 * 2$ >>, elle contient une *quotation* (la partie délimitée par <:...< et >>). Une analyse lexicale était faite pour chacune des *quotations*. Cette analyse attribuait au contenu de la *quotation* des *locations* relatives au début de celle-ci. Il était donc nécessaire de décaler ces *locations* de manière à ce qu'elles soient relatives au fichier. Dans notre exemple tout doit être décalé de seize caractères.

Pour résoudre ce genre de problèmes j'ai simplement véhiculé toutes les *locations*. Ainsi la fonction débutant l'analyse d'une *quotation* est paramétrée par la *location* initiale. Le travail a donc été de mettre à jour toutes les parties du code qui manipulaient des *locations* pour qu'elles reçoivent une *location* correcte plutôt que d'aller consulter des références, les modifier, les protéger, et les restaurer.

Une autre modification à propos des *locations* concerne le flot de lexèmes. L'analyseur lexical est traditionnellement conçu pour renvoyer le prochain lexème du flot lorsque sa fonction principale est appelée. Si l'on consulte la position dans le fichier avant et après chaque lexème on peut étiqueter chacun des lexèmes du flot par sa *location*. Cependant CAMLP4 n'étant pas

prévu dès le départ pour transporter des *locations*, pour des raisons de simplicité les *locations* associées aux lexèmes ne furent pas transportées partout où l'information était nécessaire. En revanche un système de **fonctions de** *locations* a été introduit. L'analyseur lexical renvoyait un flot de lexèmes et une fonction de *location* qui était capable d'associer une position à chaque lexème du flot. Il y a deux problèmes avec cette solution. Premièrement le maintien de cette fonction n'est pas évident car cette fonction est liée à un seul flot et vu que l'objectif était de ne pas transporter systématiquement cette information, la table était donc plus ou moins globale. Le deuxième problème concerne la gestion mémoire. En effet la table qui sert de valeur interne pour la fonction de *location* n'est pas collectée par le GC au fur et à mesure que les lexèmes sont inutilisés et donc collectés par le GC. Cela ne pose pas trop de problèmes pour une utilisation classique où seul un fichier est analysé. Par contre cela devient réellement gênant pour des applications avec une longue durée de vie analysant des flots potentiellement infinis. Pour résoudre ce problème il fallait tout simplement garder un flot de couples, c'est-à-dire de lexèmes avec *location*. J'ai donc mis à jour l'ensemble des sources de CAMLP4 pour y intégrer ce fonctionnement.

4.4 Réécriture de l'analyseur lexical

L'analyseur lexical reconnaît des mots dans un flot de caractères, et les identifies comme lexèmes. Le rôle de l'analyseur lexical ne s'arrête pourtant pas réellement aux mots. Il s'attache aussi à reconnaître certaines constructions comme les commentaires imbriqués, les citations imbriquées et d'autres encore.

L'analyseur lexical de CAMLP4 était écrit «à la main» en utilisant la syntaxe dite des «*parseurs* de flots» (figure 4.1). Cette syntaxe permet de définir des fonctions de filtrage destructif sur des flots. Cela permet donc de faciliter l'écriture d'analyseurs définis comme des fonctions mutuellement récursives. Généralement le calcul est dirigé par le premier élément du flot que l'on «regarde» pour choisir une branche dans laquelle va se poursuivre le calcul.

```
let rec integer accu = parser
| [< ' (('0'..'9') as c); stream >] ->
    integer (10 * accu + (Char.code c - Char.code '0')) stream
| [< >] -> accu
in integer 0 [< ' '4'; ' '2' >]
```

FIG. 4.1 – Exemple de «*parseur* de flots» reconnaissant [0-9]*

4.4.1 Les défauts de la solution actuelle

L'analyseur lexical de Camlp4 comportait plusieurs défauts. Premièrement il n'était pas ré-entrant, car il manipulait un état partagé entre toutes les instances d'un analyseur. Deuxièmement les buffers («tampons mémoire») utilisés pour accumuler le contenu du lexème courant étaient dans un style pseudo-fonctionel. Ces buffers avaient un inconvénient du style fonctionnel (transmission de l'état en cours : la position dans le buffer) et un inconvénient du style impératif puisqu'il y a tout de même un effet de bord dans la chaîne. Troisièmement, la syntaxe des «*parseurs* de flots» n'est pas très adaptée à l'analyse lexicale où les langages reconnus sont souvent rationnels. Dans le cadre des langages rationnels, les expressions rationnelles sont un moyen bien plus concis pour exprimer le langage à reconnaître et par conséquent l'analyseur lexical. Quatrièmement d'un point de vue génie logiciel, l'ancien analyseur s'occupait de deux tâches simultanément ce qui réduisait sa réutilisabilité et compliquait le code. En effet l'ancien analyseur était paramétré par la table des mots-clés valides. Cette table permettait de distinguer un mot-clé d'un identifiant classique, et donc de produire le lexème correspondant. Cette table et sa modification impliquaient finalement un lien étrange entre l'analyseur lexical et son utilisateur.

4.4.2 Solution retenue, justification

C'est pour cet ensemble de raisons que la décision de réécrire l'analyseur lexical de Camlp4 a été prise. Le nouvel analyseur utilise `ocamllex`, il est écrit d'une manière plus fonctionnelle, il est ré-entrant, il utilise le module `Buffer` de la bibliothèque standard d'Objective Caml et utilise des buffers locaux, il ne s'occupe plus de gérer les mots-clés directement, déléguant cette tâche aux filtres de lexèmes. Le résultat obtenu est significativement plus rapide car les automates produits par `ocamllex` sont plus efficaces. Il est aussi deux fois plus compact et beaucoup plus compréhensible car plus déclaratif.

4.5 Ajout d'un système de filtrage

4.5.1 Camlp4 : un traducteur source-à-source

Camlp4 avait deux sortes de composants : les *parseurs* (analyseurs syntaxiques) et les *pretty-printeurs*. Cela permet de lire des programmes écrits dans différents langages, d'en obtenir une forme d'arbre interne appelée Ast, puis d'afficher cet arbre avec un *pretty-printeur*. C'est grâce à ce mécanisme que Camlp4 peut être utilisé comme un traducteur source-à-source. Il peut être utile de traduire du code OCaml vers la syntaxe révisée d'OCaml pour commencer à utiliser cette syntaxe sans pour autant modifier le code à la main. Cela peut être aussi utile dans l'autre sens lorsque l'on ne comprend

pas la syntaxe révisée. Et puis en dehors d'OCaml d'autres langages sont utilisables. En effet CAMLP4 a disposé de *parseurs* pour SML et SCHEME. Les *parseurs* et les printeurs sont chargés dynamiquement, ils permettent ainsi de changer le comportement du programme CAMLP4 qui peut donc traiter des fichiers écrits dans différents langages, sans pour autant réécrire de fonction principale (main).

4.5.2 Des filtres, et la transformation de programmes devient possible

La transformation de programme n'était pas aisée, car si l'on peut lire puis écrire, rien ne permettait de transformer la représentation sous forme d'arbre entre la lecture et l'écriture. C'est pour cette raison que le système de filtres a vu le jour. Ce système est assez simple. Les modules qui veulent filtrer disposent d'une fonction ou de plusieurs fonctions de filtrage. Ils les enregistrent auprès du système via des fonctions d'ordre supérieur. Lors de l'exécution du programme, lorsque l'analyse syntaxique a produit l'AST, les fonctions enregistrées préalablement sont appelées successivement à la manière de *fold* (`fn(... f3(f2(f1(ast))))`).

4.5.3 *Map* généré, filtrage aisé

Le fait de pouvoir insérer des filtres ne suffit pas à pouvoir transformer aisément l'arbre. En effet une simple transformation comme `x + 0 ==> x` (figures 4.2 et 4.5.4) se révèles complexe. Le manque de fonctions permettant de traverser l'arbre se fait très vite sentir. Le stage ne concernant pas du tout cet aspect-là je me suis contenté de *map*. La fonction *map* est régulièrement présente dans les langages de programmation. Lorsqu'elle s'applique à une structure de données, cette fonction permet généralement d'obtenir une autre structure où chacun des éléments a été transformé par une même fonction (par exemple la fonction `List.map` a pour type `(`α `->` β`)` `->` α `list ->` β `list` et se comporte ainsi : `[x0 ; x1 ; ... ; xn] ==> [f(x0) ; f(x1) ; ... ; f(xn)]`). Cependant, ce schéma ne s'applique pas toujours aussi bien, car les structures qui nous intéressent sont beaucoup plus hétérogènes. On dispose de plusieurs types d'arbre (les expressions, les motifs de filtrage, les types ...) qui sont assez entremêlés. On appellera par la suite ces différents types d'arbre les catégories syntaxiques du langage.

Il y a 17 catégories, réunissant 207 constructeurs de données. Envisager une telle fonction *map* devient alors délicat, de plus une seule fonction ne suffirait pas. Il en faudrait une par catégorie, chacune de ces fonctions serait paramétrée par autant de fonctions qu'il y a de catégories. L'utilisation de telles fonctions se serait révélé trop complexe. Nous pourrions donc considérer des fonctions plus simples paramétrées par seulement une

fonction à la fois. Ces fonctions seraient regroupées dans un module, par exemple `Expr` regrouperait les fonctions permettant de transformer les expressions contenues dans les expressions, les motifs, les types, etc. Il faut un module comme celui-ci par catégorie syntaxique. On sent donc tout de suite que la taille du module *Map* considéré serait conséquente (17 * 207 = 3519). De plus lorsque l'on veut modifier plusieurs catégories on doit faire plusieurs parcours ce qui est assez coûteux.

Une autre solution serait d'utiliser une classe proposant un parcours par défaut se traduisant par une recopie de l'arbre à l'identique. Ce comportement par défaut pourrait être modifié à l'aide d'héritage et de redéfinition de méthodes. L'avantage d'une telle approche est de permettre en même temps d'écrire des fonctions *map* très précises, changeant une ou plusieurs catégories à la fois, tout en conservant une réutilisation de code intéressante sur de telles structures. Ce mécanisme est semblable à celui d'Cαml de François Pottier (prononcé alphaCaml) [Pottier, 2006, Pottier, 2005] où de telles classes sont générées à partir de la signature donnée à Cαml. La différence réside dans le fait qu'il n'y a pas une méthode par catégorie et constructeur mais seulement par catégorie. En effet avoir une méthode par constructeur de données serait inadapté, car la pratique de CAMLP4 veut que l'on n'ait pas à connaître ces 207 constructeurs mais seulement la syntaxe des 17 catégories.

Plutôt que d'écrire 17 modules de chacun 207 cas, le choix s'est porté sur la génération du code de la deuxième solution (en effet le code produit pour la première solution faisait environ 5000 lignes). Le code est donc produit à partir de la définition du type, c'est donc un filtre qui emmagasine les définitions de types puis crée l'arbre de la classe qui implémente la fonction *map*. La fonction *map* générée suit un parcours de bas en haut (bottom-up). Le résultat est plus concis (environ 400 lignes), plus puissant dans son utilisation. Un inconvénient pourrait être la difficulté d'utilisation, car dériver d'une classe pour y redéfinir une méthode est plus contraignant que l'utilisation classique à base de fonctions passées en argument. Cependant ce mécanisme est très facile à créer par dessus ce système et l'on peut donc proposer ce type de fonctions pour les utilisations les plus classiques.

Cependant j'aurais aimé poursuivre ce travail et proposer des moyens plus modernes de traverser et transformer ce type de structure de données. Par exemple j'aurais pu utiliser des Zippers [Huet, 1997], ou des réitérateurs [Filliâtre, 2006b, Filliâtre, 2006a], en générant ces itérateurs à partir de la définition du type.

4.5.4 Filtres disponibles et exemple

Différents filtres ont été mis au point, pour générer du code comme pour le *map*. Un filtre très simple a été développé afin de faciliter le déboguage, ce filtre rajoute une construction `try ... with` autour de chaque corps de fonction. Ce gestionnaire d'exception capture n'importe quelle exception, l'affiche ainsi que la position dans le code source et la relance. Ce type de petit filtre s'est révélé très pratique et assez efficace.

```
open Camlp4.PreCast
let simplify =
  object (self)
    inherit Ast.map as super
    method expr e =
      match e with
      | <:expr< $x$ + 0 >> | <:expr< 0 + $x$ >> -> self#expr x
      | e -> super#expr e
  end
in AstFilters.register_str_item simplify#str_item
```

FIG. 4.2 – Exemple de définition de filtre

```
$ ocamlc -c Camlp4.cma -pp camlp4of simplify0.ml
$ camlp4of simplify0.cmo -printer OCaml \
            -str 'fun x -> x * (0 + 3 + 0 + 0)'
fun x -> x * 3
```

4.6 Réécriture d'un *pretty-printeur*

Les *pretty-printeurs* de CAMLP4 disposaient d'un mécanisme d'extensibilité. Cependant ce mécanisme n'a pas été autant développé que pour les grammaires extensibles. Aujourd'hui on peut dire que les *pretty-printeurs* de CAMLP4 n'ont pas de succès. Finalement alors que le nombre d'extensions de syntaxe ou de front-ends pour OCaml ne cesse de croître, le nombre de *pretty-printeurs* développés à l'extérieur est quasi nul.

Par contre les deux *pretty-printeurs* principaux (syntaxe OCaml, syntaxe révisée) sont indispensables. Il a donc été décidé de simplifier le système par le vide. Le mécanisme de *pretty-printeur* extensible disparaîtra progressivement, les deux *pretty-printeurs* principaux resteront mais utiliseront des techniques d'affichage plus standards. C'est dans cette optique que j'ai réécrit le *pretty-printeur* d'OCaml.

L'ancien système de *pretty-printeur* comportait plusieurs défauts. Le principal était la complexité du code. Il était vraiment difficile de se faire une idée précise de l'exécution d'un *pretty-printeur*. De plus la gestion des commentaires n'était pas idéale. J'ai donc décidé de refaire un *pretty-printeur* utilisant une évaluation plus classique et le module `Format`.

4.6.1 Le module `Format`

Dans la bibliothèque standard d'Objective Caml, le module `Format` représente le moyen le plus évolué de faire de l'affichage structuré. Il comporte un système de boîtes et d'informations de coupure. Les boîtes permettent d'indiquer le formatage à suivre. Dans les boîtes horizontales le système tente de tout afficher sans revenir à la ligne. Dans les boîtes verticales chaque information de coupure est interprétée comme un changement de ligne. Les boîtes horizontales-verticales se comportent comme des boîtes horizontales tant qu'une seule ligne suffit, sinon une boîte complètement verticale est utilisée. Finalement les boîtes horizontales-ou-verticales se remplissent horizontalement tant que possible et utilisent des coupures verticales lorsqu'une ligne est pleine.

Le système repose sur un mécanisme de flot, il reçoit des informations d'affichage : des ouvertures et fermetures de boîtes, des coupures, des chaînes, etc. ; il affiche en sortie les chaînes à l'identique, il enregistre les informations de boîte et transforme les informations de coupure en fonction du contexte. Il ne garantit pas de choisir les meilleures coupures ou de ne jamais dépasser la marge imposée. Par contre pour s'approcher de la meilleure solution le module `Format` essaie de retarder ses décisions de coupure. Pour conserver un algorithme efficace le module `Format` ne revient pas sur ses décisions de coupure.

4.6.2 Mesure de la qualité d'un *pretty-printeur*

Les caractéristiques qui permettent de juger de la qualité d'un *pretty-printeur* sont les suivantes : premièrement le *pretty-printeur* se doit d'être correct, c'est-à-dire que pour un arbre correct donné, il doit produire un texte qui appartient au langage ciblé et qui a la même sémantique. Deuxièmement, il ne doit pas être déviant, c'est-à-dire que lorsque l'on dispose d'un code source `A`, cette égalité doit être vraie modulo les *locations* `parse(pp(parse(A))) = parse(A)` (`pp` pour pretty-print). Troisièmement il doit produire un texte agréable à lire, si possible dans un style proche de ce qu'écrivent les développeurs eux-mêmes. Ainsi le *pretty-printeur* produit un texte indenté, respectant au mieux une largeur de ligne maximale (80 colonnes par exemple). Il doit aussi minimiser le nombre de parenthèses qu'il introduit, en tenant compte par exemple de la priorité des opérateurs, sans bien entendu produire un code

ambigu ou obtenir une sémantique différente. Quatrièmement si l'arbre à afficher provient d'un fichier source, il peut essayer de restaurer les commentaires et le style présent dans le fichier original.

4.6.3 Correction et non-déviance

Ces deux propriétés du *pretty-printeur* peuvent être testées mécaniquement. Comme l'indique la relation `parse(pp(parse(A))) = parse(A)`, tester la correction n'est pas difficile. Pour être indépendant des *locations*, on les normalise à l'aide d'un filtre, ce qui donne `strip_loc(parse(pp(parse(A))))` `= strip_loc(parse(A))`. Cependant lorsque ce test échoue, le développeur n'est pas aidé quant à la nature précise du problème. Une solution serait de pouvoir calculer la différence de deux arbres. Pour des raisons de simplicité on préfère calculer la différence textuelle entre le pretty-print de chaque arbre. Pour *pretty-printer* des arbres (figure 4.4), on n'écrit pas un autre *pretty-printeur* (qui ne servirait qu'à tester le premier). Dans ce but j'ai utilisé la fonction `meta_lift` (décrite dans la section 4.9.2) et utilisé l'ancien *pretty-printeur* sur le résultat. La formule devient la suivante :

```
let pp_tree x = old_pp (meta_lift x) in
let diff_tree a b = diff_text (pp_tree a) (pp_tree b) in
let ref_tree = parse ref in
let my = pp ref_tree in
diff_tree (strip_loc (parse my)) (strip_loc ref_tree)
```

FIG. 4.3 – Test du *pretty-printeur*

```
let y = f x in y
(* becomes in a simpler representation *)
Let (Id "y")
    (App
      (Id "f")
      (Id "x"))
    (Id "y")
```

FIG. 4.4 – Exemple du *pretty-printeur* d'arbre

4.6.4 Priorités, et structures ambiguës

La gestion des priorités des opérateurs de sorte à ce qu'un nombre minimal de parenthèses soit affiché est un problème assez classique. Lorsque le *pretty-printeur* est écrit manuellement à l'aide de fonctions mutuellement récursives, une solution courante est de recréer des niveaux de priorité en

découpant une fonction en plusieurs niveaux. Atteindre le dernier niveau provoque l'ajout de parenthèses. Diminuer le nombre de parenthèses n'est pas non plus une priorité absolue. De temps en temps il vaut mieux forcer un certain parenthésage s'il augmente la lisibilité.

Un problème plus délicat concerne les structures ambiguës comme les `match` imbriqués qu'il faut parenthéser (figure 4.5). Pour traiter ce genre de problèmes, où l'affichage dépend réellement du contexte, on peut par exemple véhiculer une structure représentant le contexte. Cette structure porte par exemple le fait de devoir parenthéser le prochain `match`. Ce booléen est activé dans chaque branche d'un `match`, il est désactivé lorsqu'un parenthésage est ajouté. De manière à implanter correctement un tel algorithme il est préférable d'utiliser une structure purement applicative (purement fonctionnelle, non modifiable ...) pour porter cette information.

```
match x with
| A1 y ->
    (match y with
     | B1 -> ...
     | B2 -> ...)
| A2 -> ...
```

FIG. 4.5 – Sans les parenthèses A2 serai au niveau de B1 et B2

Certains détails dans l'impression de structures comme le `match` sont eux aussi délicats. Prenons par exemple le premier `|` optionnel. Souvent lorsqu'il est utilisé, il l'est dans les constructions verticales, et non pas dans celles tenant sur une seule ligne (figure 4.6). Pour réaliser cet affichage le module `Format` propose une fonction appropriée qui ne fait l'action suivante que si la ligne est vide. Il suffit donc d'afficher `"| "` seulement si la ligne est vide. Cependant un bogue dans l'implémentation actuelle du module `Format` rend son utilisation quasiment impossible, la résolution du problème avec l'auteur a été engagée.

```
match x with
| Blue -> 2
| Red -> 21
match x with Blue -> 2 | Red -> 21
```

FIG. 4.6 – Avec et sans l'effet de `pp_if_newline`

4.6.5 Solution proposée

Le nouveau *pretty-printeur* pour la syntaxe originale d'OCaml est implémenté à l'aide d'une classe fonctionnelle permettant d'encapsuler le contexte d'une manière applicative. Les méthodes utilisent la syntaxe concrète apportée par les *quotations* pour filtrer les arbres en entrée, et le module `Format` pour afficher le résultat. Une utilisation quasi-systématique des boîtes et des informations de coupures ont permis d'obtenir un affichage agréable proche de celui utilisé manuellement. Certaines priorités ont été encodées en stratifiant certaines méthodes. Les tests ont montré que le *pretty-printeur* était correct et non déviant en accord avec les tests proposés plus haut. Les commentaires sont restaurés de la manière suivante : lorsque l'on affiche un nœud à la *location* `L`, on affiche tous les commentaires non encore affichés dont la *location* est strictement inférieure à `L`. Cette technique semble suffisante car la plupart des commentaires se trouvent devant le nœud qu'ils décrivent.

4.7 Renommage et fonctorisation

Les problèmes de conventions de nommage et d'espaces de noms ne sont pas les problématiques qui passionnent le plus les utilisateurs et les développeurs d'Objective Caml.

4.7.1 Justification du renommage

Dans Camlp4 lorsqu'un nom de module était déjà réservé par un autre module (dans le compilateur par exemple), un nom légèrement différent a été choisi. Ainsi sont apparus des noms comme `Plexer`, `Pcaml`, `MLast`, etc. De même lorsqu'au fil du temps il a fallu mettre à jour quelques prototypes de fonctions, d'autres fonctions ont été créées pour éviter les changements qui ne garantissent pas la compatibilité arrière. Ainsi les fonctions `Grammar.gcreate` et `Plexer.gmake` viennent remplacer `Grammar.create` et `Plexer.make_lexer`.

Comme la nouvelle version que je développais ajoutait de nouveaux modules, le nom de ces modules risquait d'entrer en conflit avec des modules existants développés par d'autres utilisateurs. Afin d'éviter ces problèmes, il a été décidé de protéger les modules de Camlp4 par un module Camlp4. Ce module englobe tout le noyau de Camlp4. Les extensions sont regroupées dans quatre modules : `Camlp4Parsers`, `Camlp4Printers`, `Camlp4Filters` et `Camlp4Top`. Ces modules contiennent respectivement les analyseurs syntaxiques, les modules d'affichage, les transformations sur les arbres, et les extensions de la boucle d'interaction de Caml.

4.7.2 Fonctorisation

Alors que le renommage était déjà commencé, il a été décidé d'uniformiser le nom des modules et fonctions de CAMLP4. Ce renommage a permis d'introduire deux caractéristiques qui m'étaient chères : premièrement la séparation réelle des signatures et des implémentations, deuxièmement le paramétrage des implémentations par les implémentations nécessaires. Avec cette méthode les utilisations directes des autres composants sont abstraites à l'aide d'un mécanisme permettant le paramétrage. Ce découplage permet une plus grande réutilisabilité des composants logiciels et une meilleure visibilité des dépendances. Cependant il est vrai que l'utilisation de foncteurs pour réaliser ce découplage ne facilite pas l'abord du projet. En effet un système de `mixin modules` [Hirschowitz and Leroy, 2005] améliorerait cet aspect.

4.7.3 Module pré-construit : `Camlp4.PreCast`

Les sources de CAMLP4 étant dorénavant entièrement paramétrables, il n'en est pas moins indispensable de définir un assemblage, une instanciation par défaut convenant aux usages classiques. C'est le rôle du module `Camlp4.PreCast` qui instancie toutes les implémentations nécessaires à une utilisation standard de CAMLP4, ainsi qu'à la création de l'exécutable final. Ce module constitue un bon départ pour les développeurs qui veulent utiliser CAMLP4.

Pour les utilisateurs ayant des besoins spécifiques, le remplacement d'un des composants de CAMLP4 est devenu plus aisé. En effet, après création d'un nouveau composant permettant de remplacer un de ceux existants, il suffit de recréer un module pré-construit, afin qu'il prenne en compte ce changement. Cette dernière étape est facile : il faut recopier le module `Camlp4.PreCast` (50 lignes) et faire référence aux nouveaux composants.

4.8 Motifs de lexèmes arbitraires dans les grammaires extensibles

4.8.1 Les grammaires extensibles de CAMLP4

Une des forces de CAMLP4 est le dynamisme de sa technologie d'analyse syntaxique. Quoiqu'un peu *ad-hoc* et très algorithmiques, les grammaires de CAMLP4 offrent un outil assez puissant et fournissent au programmeur un système à mi-chemin entre la spécification formelle d'une grammaire et le *parseur* écrit à la main. Une grammaire (figure 4.7) se décompose en un ensemble d'entrées, chaque entrée est décomposée en un nom et une liste de niveaux, chaque niveau en une liste de règles, et chaque règle en une

liste de symboles et une action. Les niveaux sont potentiellement étiquetés par un nom et une associativité. Les symboles sont soit des terminaux, soit des non-terminaux. Les symboles sont potentiellement nommés par un nom de variable. Les non-terminaux sont des noms d'entrées (potentiellement étiquetés par un niveau), ou des règles anonymes introduites en-place. Les terminaux sont soit des chaînes de caractères alors interprétés comme des mots-clés, soit des motifs de lexèmes.

```
EXTEND Gram
  expr:
    [ [ x = expr; "+"; y = expr -> x + y
      | x = expr; "-"; y = expr -> x - y ]
    | [ x = expr; "*"; y = expr -> x * y
      | x = expr; "/"; y = expr -> x / y ]
    | [ x = INT -> int_of_string x
      | "("; e = expr; ")" -> e ] ];
END
```

FIG. 4.7 – Grammaire d'expressions arithmétiques simples

Ces grammaires sont évaluées lors de l'analyse lexicale un peu comme le ferait un *parseur* récursif descendant. La différence principale est la suivante : la représentation interne des grammaires est «factorisée à gauche», c'est-à-dire que lorsque deux règles d'un même niveau ont un préfixe commun, alors cette partie est commune dans la structure interne.

Voici comment fonctionne l'analyse lexicale : le *parseur* consomme un flot de lexèmes, et essaye généralement d'entrer dans une règle. Il essaye donc successivement les règles d'une entrée en commençant par le premier niveau. Dès que le premier non-terminal ou les premiers terminaux correspondent à la tête du flot (les premiers terminaux d'une règle sont considérés d'un bloc pour effectuer une factorisation supplémentaire), la règle est irrémédiablement choisie et la tête du flot consommée. La technique d'analyse syntaxique est donc plutôt LL(k) une fois que la grammaire est factorisée à gauche (k étant une borne maximale égale au nombre maximal de terminaux en tête de règle).

4.8.2 Motifs de lexèmes

Les symboles terminaux d'une grammaire CAMLP4 permettent d'écrire une certaine forme de filtrage. Dans cette grammaire (figure 4.8) on peut distinguer quatre formes de motifs. Les motifs comme `"let"`, `"="` et `"in"` sont des motifs de mots-clés. Les motifs comme `s = LIDENT` et `i = INT`

sont des motifs nommés : `s` et `i` lient le contenu du lexème (LIDENT signifie lowercase identifier, UIDENT uppercase identifier). Un motif comme `UIDENT "Foo"` ne correspond qu'au lexème correspondant à l'identifiant `Foo`. Un motif comme `STRING` peut correspondre à n'importe quel lexème de chaîne de caractères (`"abc"`).

```
EXTEND Gram
  expr:
  [ [ "let"; s = LIDENT; "="; i = INT; UIDENT "Foo";
      "in"; STRING -> LetFoo (s, int_of_string i, i)
  ] ];
END
```

FIG. 4.8 – Grammaire montrant l'utilisation de motifs de lexèmes

4.8.3 Représentation des lexèmes et des motifs de lexèmes

Originellement dans CAMLP4 les lexèmes sont représentés à l'aide d'un couple de chaînes de caractères. La première partie représente la classe de lexèmes (`UIDENT`, `LIDENT`, `INT`, `STRING`...). Si cette première partie est vide, le lexème est un mot-clé. La deuxième partie représente le contenu. Les motifs de lexèmes sont aussi représentés par un couple de chaînes. La première partie est identique à celle des lexèmes, et la deuxième partie représente le contenu attendu sauf si la chaîne est vide. Dans ce cas, tous les lexèmes de cette classe correspondent à ce motif.

Cette représentation est très souple, car on peut tout encoder dans une chaîne de caractères. Néanmoins ce n'est pas très bon du point de vue de la sûreté du code. Généralement les lexèmes sont représentés par un type algébrique (dit aussi type somme) où chaque classe de lexèmes est représentée par un constructeur de données. Ce type (figure 4.9) permet de spécifier plus finement le contenu de chaque lexème. On peut par exemple choisir que le constructeur `INT` porte une valeur entière plutôt que sa représentation sous forme de chaîne de caractères, ou les deux. On peut aussi facilement disposer de lexèmes contenant deux valeurs comme les *quotations* (<:`expr`< `foo` >>) qui ont au moins deux valeurs : le nom (`expr`) et le code ('foo').

Ce nouveau type de lexèmes apporte plus de sûreté et évite certaines conversions ou certains encodages dus à la restriction aux chaînes de caractères. Si l'on devait convertir les terminaux de cette grammaire (figure 4.8) en motifs CAML on obtiendrait : `KEYWORD "let"`, `LIDENT i`, `KEYWORD "="`, `UIDENT "Foo"`, et `STRING _`. On voit donc ici que le comportement actuel peut être traduit vers une utilisation de motifs CAML standards. L'ob-

```
type my_token_type =
  | LIDENT of string
  | UIDENT of string
  | KEYWORD of string
  | STRING of string
  | INT of int * string
```

FIG. 4.9 – Exemple de type algébrique des lexèmes

jectif de cette extension est de pouvoir utiliser n'importe quel motif CAML, et donc d'obtenir une plus grande puissance d'expression. L'extension s'utilise simplement, il suffit d'ajouter un ‘ pour annoncer le début d'un motif de lexèmes (figure 4.10).

```
EXTEND Gram
  expr:
  [ [ "let"; `LIDENT i; "="; `INT (i, j);
      `UIDENT "Foo"; "in"; `STRING _ -> LetFoo (s, i, j)
    | `LIDENT ("foo"|"bar" as s) ->
        FooOrBar s (* or patterns and alias *)
    | `ANTIQUOT (("exp"|"" as n), a) -> Ant (n, a)
    | `INT (42, s) -> Int s
        (* s can be 42, 000042, 0x2a, 0b0101010 ... *)
  ] ];
END
```

FIG. 4.10 – Exemple utilisant les nouveau motifs de lexèmes

4.8.4 Les difficultés rencontrées

Les grammaires extensibles de CAMLP4 sont des structures de données. La syntaxe des grammaires détaillée précédemment (**EXTEND** ... **END**) est traduite (lors de l'analyse lexicale du code source de la grammaire) en la construction d'une valeur CAML qui est par la suite évaluée lors de l'exécution. Dans cette valeur, les anciens motifs de lexèmes étaient eux aussi des valeurs (couple de chaînes de caractères) ce qui ne pose pas de problème. Maintenant le seul moyen d'inclure les motifs introduits est de créer des abstractions procédant à un filtrage d'un lexème par ce motif (figure 4.11).

La différence majeure entre ces deux représentations des motifs de lexèmes est le nombre d'opérations que l'on peut appliquer à ces valeurs. Pour que

```
function
| LIDENT ("foo"|"bar" as s) -> s
| _ -> raise Stream.Failure
```

FIG. 4.11 – Fonction d'extraction d'un motif

l'évaluation des grammaires de CAMLP4 soit réalisable, il est nécessaire que les motifs de lexèmes supportent les opérations suivantes :

- L'égalité symbolique de deux motifs est nécessaire pour que la factorisation soit réalisable (une relation d'ordre serait préférable).
- Une représentation sous forme de chaîne de caractères est nécessaire à la gestion d'erreur.
- Un prédicat indiquant si un lexème est reconnu par le motif est nécessaire à l'analyse lexicale.
- Une méthode d'extraction est nécessaire à l'appel de l'action utilisateur.

Pour satisfaire toutes ces exigences, la solution retenue utilise une fonction de prédicat (`Token.t -> bool`) et une chaîne de caractères servant pour le test d'égalité et l'affichage. L'extraction se fait à l'aide d'un `match` en prologue de l'action utilisateur.

La chaîne de caractères est le résultat du pretty-print de l'arbre de syntaxe du motif après une étape de normalisation. Cette normalisation remplace toutes les variables par le motif universel (_), et supprime tous les alias (`motif as variable` devient `motif`). La figure 4.12 montre la traduction d'une règle de grammaire en la valeur CAML correspondante.

4.9 Quotations, Méta-élévation, et réflexion

Dans un premier temps, je présente le système de *quotations* avec ses avantages et ses limites. Puis, je montre les changements effectués dans CAMLP4 pour dépasser ces limites.

4.9.1 Le système de *quotations*

Description et utilité

CAMLP4 dispose d'un système de *quotations* qui permet de manipuler des arbres de syntaxe directement avec leur syntaxe concrète. Ces *quotations* sont des sortes de guillemets référants à l'arbre de syntaxe de ce qui est contenu entre ces guillemets. Dans CAMLP4, les *quotations* sont nommées (<:*expr*<...>>, <:`patt`<...>>, etc.). Il existe cependant une *quotation* par

```
`ANTIQUOT (("exp"|"" as n), a) -> Ant (n, a)

(* becomes after transformation *)

((* This rule has only one symbol *)
 ([ (* token pattern *)
    Gram.Stoken
     ((function (* predicate in normal form *)
       | ANTIQUOT (("exp" | ""), _) -> true
       | _ -> false),
      (* the pretty-print string in normal form *)
      "ANTIQUOT␣((\"exp\"␣|␣\"\"),␣_)") ]),
 (* semantic action *)
 (Gram.Action.mk
    (fun (__camlp4_0 : Gram.Token.t) (_loc : Loc.t) ->
       (* this match extracts "n" and "a" *)
       match __camlp4_0 with
       | ANTIQUOT ((("exp" | "" as n)), a) -> Ant (n, a)
       | _ -> assert false)))
```

FIG. 4.12 – Transformation d'une règle de grammaire

défaut : <<...>>. Par exemple, lorsque la *quotation* <:*expr*< 42 >> est placée en position d'expression, elle est équivalente à `Ast.ExInt (_loc, "42")`. Les *quotations* peuvent donc être des expressions, mais aussi des motifs. Cela permet de filtrer les arbres. L'avantage de cette notation est qu'il suffit de connaître la syntaxe concrète du langage considéré pour être capable de manipuler des arbres de syntaxe abstraite. Il n'est donc pas nécessaire de connaître les 207 constructeurs de données de la syntaxe d'OCaml. Cependant, sans les antiquotations on ne pourrait traiter que des arbres constants.

Les antiquotations offrent des trous dans les *quotations*, que l'on peut remplir par des expressions (respectivement des motifs), si l'on est en position d'expression (respectivement de motif). Par exemple <:*expr*< `$str:"Hello,␣"^"world!"$` >> est égal à <:*expr*< `"Hello,␣world!"` >> après évaluation. Les antiquotations sont potentiellement nommées. Dans l'exemple précédent, le nom de l'antiquotation est `str`. Il existe une *quotation* par catégorie syntaxique et suffisamment d'antiquotations pour pouvoir trouer une *quotation* où on le souhaite. En résumé, pour manipuler des arbres de syntaxe, il suffit de connaître le nom des *quotations*, la syntaxe de chacune et les antiquotations. Cela reste raisonnable.

Ce mécanisme de *quotations* permet de traiter des programmes dans une forme structurée qui apporte une certaine sûreté. Pour être utilisable, le

système doit tout de même rester souple.

Limitations du système actuel

Il existe cependant certaines limitations aux *quotations* fournies par CamlP4. Premièrement, la syntaxe utilisée pour le contenu des *quotations* est la syntaxe révisée de Caml. Cette syntaxe est étudiée pour être moins ambiguë et donc plus utilisable sur de petits fragments de code. Cette syntaxe étant beaucoup moins utilisée que la syntaxe originale, elle constitue une barrière supplémentaire à l'utilisation des *quotations*. Deuxièmement, de nombreux fragments de code n'ont pas de *quotation* associée. Bien souvent, on a recours à l'antiquotation `list`. Par exemple, <:*expr*< **fun** [*$list*:*pwel$*] >> construit une fonction dont le contenu est `pwel`, qui est une liste de triplets, chaque triplet étant constitué d'un motif, d'une garde optionnelle et d'une expression. Certaines antiquotations ne sont pas intuitives, comme `list` et `opt`. Ce manque de syntaxe concrète force l'utilisateur à connaître précisément le type des arbres, ce qui est contraire au principe initial.

Ces questions d'apprentissage et d'utilisation intuitive sont importantes mais sont surmontables. En effet les utilisateurs convaincus de l'utilité des *quotations* dans Caml prennent le temps nécessaire à l'apprentissage des *quotations* actuelles. Un problème beaucoup moins surmontable est celui de l'extensibilité de la syntaxe des *quotations*. D'un côté CamlP4 revendique l'extensibilité de la syntaxe, et d'un autre côté la syntaxe de ses *quotations* n'est pas extensible.

4.9.2 Parseurs standards et *parseurs* de *quotations*

Pour mieux comprendre les limitations énoncées, il est nécessaire de comprendre les différences qui séparent les *parseurs* standards et les *parseurs* de *quotations* dans CamlP4. Plaçons-nous dans un contexte où le résultat de cette analyse syntaxique va par la suite être transmis au compilateur. Celui-ci traite l'arbre de syntaxe qui lui est donné en entrée, pour produire par exemple un programme pour la machine virtuelle. Prenons maintenant un exemple : le programme `print_endline "Hello, world!"` effectue l'appel de la fonction `print_endline` avec comme argument `"Hello, world!"`. Son arbre de syntaxe est représenté dans la figure 4.13, dans laquelle `loc1`, `loc2` et `loc3` sont les *locations* indiquant l'endroit dans le code source où ce code apparaît. Etudions maintenant <:*expr*< *print_endline* `"Hello,␣world!"`>>. Comme vu précédemment, cette *quotation* permet de manipuler le programme `print_endline "Hello, world!"`. Cette *quotation* est étendue au moment de l'analyse syntaxique et remplacée par son arbre de syntaxe. Cet arbre de syntaxe est lui aussi sous forme d'un arbre de syntaxe (figure 4.14). Ce dernier arbre semble beaucoup plus complexe, mais ce n'est que celui qui

est produit si l'on voit le premier arbre (figure 4.13) comme un programme source CAML.

```
Ast.ExApp (loc1,
  Ast.ExLid (loc2, "print_endline"),
  Ast.ExStr (loc3, "Hello, world!"))
```

FIG. 4.13 – L'arbre de syntaxe de `print_endline "Hello, world!"`

```
Ast.ExApp (loc1,
  Ast.ExApp (loc2,
    Ast.ExApp (loc2,
      Ast.ExAcc (loc3, Ast.ExUid (loc4, "Ast"),
        Ast.ExUid (loc5, "ExApp")),
      Ast.ExLid (loc6, "_loc")),
    Ast.ExApp (loc7,
      Ast.ExApp (loc8,
        Ast.ExAcc (loc9, Ast.ExUid (loc10, "Ast"),
          Ast.ExUid (loc11, "ExLid")),
        Ast.ExLid (loc12, "_loc")),
      Ast.ExStr (loc13, "print_endline"))),
  Ast.ExApp (loc14,
    Ast.ExApp (loc15,
      Ast.ExAcc (loc16, Ast.ExUid (loc17, "Ast"),
        Ast.ExUid (loc18, "ExStr")),
      Ast.ExLid (loc19, "_loc")),
    Ast.ExStr (loc20, "Hello, world!"))))
```

FIG. 4.14 – Méta-arbre de syntaxe pour `print_endline "Hello, world!"`

Les *parseurs* de *quotations* produisent donc des arbres de syntaxes d'arbres de syntaxes. Si l'on devait étager les *parseurs* en fonction de ce qu'ils produisent, on placerait le *parseur* classique au niveau zéro et le *parseur* de *quotations* au niveau un. On appellera par la suite ces *parseurs* P0 et P1. Dans CAMLP4 il y a donc deux *parseurs* de la syntaxe révisée (un P0 et un P1). Les grammaires sont essentiellement identiques et cependant cette «duplication» n'est pas anodine. Ce sont les actions sémantiques qui les différencient (figures 4.15 et 4.16). L'un produit un nœud alors que l'autre produit le sous -arbre représentant ce nœud. En utilisant les *quotations* on peut simplifier l'écriture de ces grammaires (figures 4.17 et 4.18).

On peut remarquer que la nouvelle version de P1 ressemble à la première version de P0 avec une *quotation* (modulo la syntaxe révisée). Dans cette

```
EXTEND Gram
  expr:
  [ [ e1 = SELF; e2 = SELF -> Ast.ExApp (_loc, e1, e2)
    ...
  ] ]
END
```

FIG. 4.15 – Parseur P0, pour la règle de l'application

```
EXTEND Gram
  expr:
  [ [ e1 = SELF; e2 = SELF ->
      Ast.ExApp (_loc,
        Ast.ExApp (_loc,
          Ast.ExApp (_loc,
            Ast.ExAcc (_loc,
              Ast.ExUid (_loc, "Ast"), Ast.ExUid (_loc, "ExApp")),
            Ast.ExLid (_loc, "_loc")),
          e1),
        e2)
      ...
  ] ]
END
```

FIG. 4.16 – Parseur P1, pour l'application

optique, pourquoi ne pas réécrire P1 (figure 4.19) en utilisant deux niveaux de *quotations*? Ceci n'est pas possible car le langage reconnu par P1 n'inclut pas les *quotations*. Avoir les *quotations* dans P1 nécessiterait soit un P2, soit une fonction permettant d'élever un arbre en un méta-arbre (c'est-à-dire un arbre d'arbre). Cette fonction pourrait s'écrire d'une manière assez systématique en utilisant par exemple les *quotations* d'un côté pour le filtrage et de l'autre pour produire les méta-arbres (figure 4.20). Avec de telles fonctions (une par type : `expr`, `patt`, `loc`...), il semble aisé d'utiliser `P0+méta` à la place de P1. Cependant un détail va tout compliquer : les antiquotations.

4.9.3 Les antiquotations, le cauchemar des *quotations*

Pour suivre un modèle où l'on n'utilise qu'un seul *parseur* P0, il faut pouvoir traiter la partie que seul P1 traite : les antiquotations. Les antiquotations interviennent un peu partout dans la grammaire puisque l'on peut les placer quasiment n'importe où. Le problème n'est pas seulement d'intégrer les antiquotations dans la grammaire de P0 mais plutôt de placer l'information dans l'arbre produit par P0. Prenons un exemple : <:*expr*< *f* *e* >>.

```
...
e1 = SELF; e2 = SELF -> <:expr< $e1$ $e2$ >>
...
```

FIG. 4.17 – P0 avec les *quotations*

```
...
e1 = SELF; e2 = SELF -> <:expr< Ast.ExApp _loc $e1$ $e2$ >>
...
```

FIG. 4.18 – P1 avec les *quotations*

Ici `e` est une expression si la *quotation* est en position d'expression et un motif si la *quotation* est placée comme telle. Il est donc plus simple de conserver l'antiquotation `$e$` sous forme de chaîne de caractères.

Cette chaîne ne convient pourtant pas directement, car ici le nœud d'application attend deux expressions. On ajoute donc un constructeur pour les antiquotations d'expressions (`Ast.ExAnt`), et on fait de même pour toutes les autres catégories syntaxiques. Les antiquotations peuvent aussi remplacer directement une chaîne de caractères. Une solution consiste à distinguer une antiquotation d'une chaîne classique à l'aide d'une séquence d'échappement interdite dans les chaînes CAML (`\$` par exemple). Plus difficile maintenant, cette *quotation* <:*expr*< **fun** [$*list*:*pwel*$] >> contient une antiquotation `list` qui a pour type (*Ast.patt* $*$ *Ast.expr option* $*$ *Ast.expr*) *list*. Définir une exception permettant de placer la chaîne de l'antiquotation serait absurde, ajouter une indirection (`type` α `or_anti = Any of` α `| Ant of string`) serait aussi trop lourd.

```
... (* ... this code does not work ... *)
e1 = SELF; e2 = SELF -> <:expr< <:expr< $e1$ $e2$ >> >>
...
```

FIG. 4.19 – Essai raté de simplification de P1

```
let rec meta_expr e =
  match e with
  | <:expr@loc< $e1$ $e2$ >> ->
      <:expr@loc< Ast.ExApp $meta_loc loc$
                  $meta_expr e1$ $meta_expr e2$ >>
  | <:expr@loc< $int:s$ >> ->
      <:expr@loc< Ast.ExInt $meta_loc loc$ $str:s$ >>
  | ...
and meta_patt p =
  ...
and meta_loc l =
  ...
```

FIG. 4.20 – Ebauche de la fonction de méta-élévation

4.9.4 Solution retenue et justification

La solution retenue vise à résoudre deux problèmes à la fois : le côté intuitif des antiquotations au type complexe comme les listes, le manque de syntaxe concrète pour certains fragments du langage, et finalement le placement des antiquotations dans l'arbre de syntaxe abstraite.

La solution consiste en la suppression des listes et des options dans l'arbre. Pour cela des constructeurs sont ajoutés pour traiter toutes les constructions où les listes étaient utilisées. On choisit des constructeurs binaires, et un élément terminal (comparable à nil ou None) pour représenter ces structures. À titre d'exemple, continuons avec <:*expr*< **fun** [*$list:pwel$*] >> qui devient <:*expr*< **fun** [*mc*] >> où `mc` est une branche de filtrage (*match case* figures 4.21 et 4.22).

```
type match_case =
  | McNil of Loc.t
  | McOr of Loc.t * match_case * match_case
  | McArrow of Loc.t * patt * expr * expr
  | McAnt of Loc.t * string
```

FIG. 4.21 – Type des règles de filtrage

```
mc ::= (* empty *)
     | mc "|" mc
     | patt "->" expr
     | patt "when" expr "->" expr
     | $s$
```

FIG. 4.22 – Grammaire abstraite des règles de filtrage

On peut donc non seulement construire de nouveaux fragments comme <:*match_case*< *Foo i* −> *i* >> mais aussi les combiner aisément. Par exemple si `mc1` et `mc2` sont deux ensembles de cas de filtrage, alors cette *quotation* <:*expr*< **fun** [*$mc1$* | *$mc2$* | _ −> *error* ...] >> crée une fonction qui rassemble `mc1` et `mc2` et fournit un cas par défaut de gestion d'erreurs. Dans le formalisme précédent, l'on aurait écrit d'une manière bien moins intuitive (figure 4.9.4).

```
<:expr< fun [ $list:mc1 @ mc2 @
                  [(<:patt< _ >>, None, <:expr< error ... >>)]$ ] >>
```

Avec cette nouvelle représentation, les antiquotations ne posent pas d'autres problèmes. On peut donc définir P1 comme étant la combinaison de P0, de la fonction méta, et d'une expansion des *quotations* sur le méta-arbre obtenu. P1 étant maintenant défini dynamiquement à partir de P0, il bénéficie des extensions de syntaxe de P0. Des *quotations* comme <:*expr*< **parser** [< '42; *s* >] −> *s* >> ou encore <:*expr*< **EXTEND** ... **END** >>, et même <:*expr*< <:*patt*< *Foo x* >> >> sont dorénavant utilisables.

De manière à pouvoir utiliser la syntaxe originale d'OCaml dans les *quotations*, la grammaire a elle aussi été adaptée. Cette grammaire est maintenant sous la forme d'une extension de la syntaxe révisée, ce qui permet un partage plus fort entre les deux grammaires.

Chapitre 5

Bilan

5.1 Intérêt du stage pour l'entreprise

Le stage est issu d'un besoin réel de l'équipe Gallium. Par manque de temps, l'équipe n'a pas fait évoluer le projet Camlp4 durant les dernières années. Ainsi, le besoin d'un Camlp4 plus stable et maintenable devenait urgent.

À long terme, le travail effectué pendant le stage permettra de maintenir plus facilement Camlp4. Le traitement des bogues soumis par les utilisateurs sera plus rapide et facile. De plus, le développeur principal de Camlp4 ne faisant plus partie du projet, la connaissance et la maîtrise d'œuvre du projet a tendance à disparaître. Le résultat de ce stage, y compris le présent rapport aide à la diffusion des connaissances que j'ai acquises durant les six derniers mois. D'autre part, le fait d'avoir refondu en profondeur Camlp4 a permis d'ajouter des fonctionnalités avancées, qui étaient notamment attendues par de nombreux utilisateurs. Ceci améliore donc la popularité de Camlp4. Enfin, d'ici quelques temps, Caml passera en mode maintenance seulement, et par conséquent Camlp4 aussi. L'équipe a donc grand intérêt à ce que Camlp4 soit à jour, et facilement maintenable. Cela lui permet de consacrer plus de temps aux activités de recherche qui déboucheront peut-être sur un nouveau langage.

5.2 Intérêt personnel

Je souhaitais depuis longtemps travailler à l'INRIA, participer au développement d'un langage aussi important que Caml. J'apprécie ce langage car il allie sûreté et efficacité avec réussite, et j'étais donc heureux de pouvoir y contribuer. Le sujet était idéal, en effet j'ai toujours apprécié le domaine des langages de programmation, et la sûreté des programmes. Au travers de ma

formation à l'ÉPITA et surtout au LRDE, j'ai pris part au domaine du traitement de la syntaxe des langages de programmation, à la transformation de programmes, et plus particulièrement à la réflexivité. Tous ces thèmes se retrouvent dans Caml, et surtout dans Camlp4. De plus l'environnement de recherche dans lequel s'est passé le stage est passionnant, et m'a appris beaucoup.

5.3 Conclusion

Pour conclure je remercierais Michel de m'avoir permis de participer à ce projet en tant que stagiaire. L'objectif était de rendre Camlp4 plus maintenable, pour cela une rénovation profonde a été nécessaire. Ce travail apporte à l'équipe Gallium un nouvel élan pour ce composant de la distribution Objective Caml, il va permettre une meilleure diffusion de la connaissance technique du logiciel. Ma contribution apporte aussi son lot de nouvelles fonctionnalités essentiellement dues à la rénovation profonde apportée. Il y a cependant des points perfectibles. La nouvelle mouture de Camlp4 n'est pas pour autant parfaite, cependant elle constitue une base nouvelle sur laquelle on pourra construire plus aisément. De plus pour prendre un exemple précis, j'aurais aimé poursuivre le travail sur les parcours d'arbre (à la fin de la section 4.5.3).

J'ai beaucoup apprécié l'encadrement apporté par Michel, car il a été disponible et nos rendez-vous étaient réguliers (deux jours par semaine), bien qu'il ait beaucoup de responsabilités en ce moment. Finalement je remercie toute l'équipe Gallium avec qui j'ai pu partager des idées et des remarques sur nos travaux respectifs. Tout cela n'aurait pas non plus été possible sans ma formation à l'ÉPITA, et plus particulièrement pendant ma période de formation par la recherche au LRDE.

Bibliographie

[Bertot and Castéran, 2004] Bertot, Y. and Castéran, P. (2004). *Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series.

[Filliâtre, 2006a] Filliâtre, J.-C. (2006a). Backtracking iterators. Research Report 1428, LRI, Université Paris Sud. English translation of [Filliâtre, 2006b].

[Filliâtre, 2006b] Filliâtre, J.-C. (2006b). Itérer avec persistance. In *Dix-septièmes Journées Francophones des Langages Applicatifs*, Pauillac, France. INRIA. À paraître.

[Henry et al., 2006] Henry, G., Mauny, M., and Chailloux, E. (2006). Typer la désérialisation sans sérialiser les types. In *Journées Francophones des Langages Applicatifs*.

[Hirschowitz and Leroy, 2005] Hirschowitz, T. and Leroy, X. (2005). Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems*, 27(5) :857–881.

[Huet, 1997] Huet, G. (1997). The zipper. *J. Funct. Program.*, 7(5) :549–554.

[Intel, 2006] Intel (2006). reflect.

[Pottier, 2005] Pottier, F. (2005). Cαml.

[Pottier, 2006] Pottier, F. (2006). An overview of Cαml. In *ACM Workshop on ML*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 27–52.

Glossaire

Ast (Abstract Syntax Tree) :

arbre de syntaxe abstraite

Bootstrap :

en compilation : un compilateur bootstrapé est un compilateur écrit dans son propre langage.

Gc (Garbage Collector) :

récupérateur automatique de mémoire

Lexeur (lexer) :

analyseur lexical

Lexème (token) :

entité lexicale

Location :

zone d'un fichier source

Parseur (parser) :

analyseur syntaxique

Pretty-printeur (pretty-printer) :

enjoliveur de sources

Quotation :

notation pour exprimer des AST en syntaxe concrète