Rénovation du sous-système Camlp4 d'Objective Caml

Nicolas Pouillard

Stage encadré et dirigé par Michel Mauny.
À l'INRIA Rocquencourt dans le projet Gallium.

6 juin 2007

Introduction, cadre du stage

L'INRIA Rocquencourt

- Institut National de Recherche en Informatique et Automatique.
- Acteur mondial au cœur de la société de l'information.
- Six unités de recherche dont celle de Rocquencourt.
- 3 600 personnes réparties dans 138 projets.

Le projet Gallium

- Anciennement le projet Cristal.
- Fondateur du langage Caml.
- Thèmes de recherche en compilation, typage, conception de langages et méthodes formelles.

Introduction, Caml et Objective Caml

Le langage Caml

- Langage multi-paradigme, mais principalement fonctionnel.
- Caml vise la sûreté et la fiabilité des programmes.
- Caml est fortement typé.

Le système Objective Caml

- Principale implémentation du langage Caml.
- Système de module puissant et une couche objet.
- Compilateur natif pour une haute performance.
- Compilateur code-octets pour la portabilité.
- Boucle interactive pour l'expérimentation.

Introduction, Camlp4

Camlp4 : un préprocesseur, *pretty-printeur* pour Caml

- Fournit un *front-end* programmable pour Objective Caml.
- Apporte un système de grammaires extensibles.
- Ainsi qu'un mécanisme de *quotations* et d'*antiquotations*.
- Propose aussi une syntaxe dite «révisée» de Caml.
- Reconnu pour les extensions de syntaxe et ses messages d'erreurs précis.

Cependant...

- Design inchangé depuis plusieurs années.
- Certaines techniques n'avaient pas suivi OCaml.

Plan

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de CAMLP4.

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de Camlp4.
- Les sources en deux exemplaires (syntaxe révisée et non-révisée).

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de Camlp4.
- Les sources en deux exemplaires (syntaxe révisée et non-révisée).
- Pas assez comprise, car pas assez expliquée.

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de CAMLP4.
- Les sources en deux exemplaires (syntaxe révisée et non-révisée).
- Pas assez comprise, car pas assez expliquée.

Objectif simplicité :

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de Camlp4.
- Les sources en deux exemplaires (syntaxe révisée et non-révisée).
- Pas assez comprise, car pas assez expliquée.

Objectif simplicité :

- Rapprochement du mécanisme d'OCaml.

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de CAMLP4.
- Les sources en deux exemplaires (syntaxe révisée et non-révisée).
- Pas assez comprise, car pas assez expliquée.

Objectif simplicité :

- Rapprochement du mécanisme d'OCaml.
- Meilleures explications.

Je bootstrappe, tu bootstrappes, nous bootstrappons

L'ancienne procédure était trop compliquée :

- Dû à la structure hautement bootstrappée de Camlp4.
- Les sources en deux exemplaires (syntaxe révisée et non-révisée).
- Pas assez comprise, car pas assez expliquée.

Objectif simplicité :

- Rapprochement du mécanisme d'OCaml.
- Meilleures explications.
- Bootstrapper Camlp4 ne doit plus être un frein.

`camlp4boot` : tout ce qu'il faut, pas plus, pas moins

- Un seul programme (bytecode) est nécessaire au *bootstrap*.
- Tous les modules sont préchargés.

Contenu de `camlp4boot` :

- La syntaxe révisée.
- La syntaxe des *parseurs* et des grammaires.
- Les *quotations*.
- La syntaxe des macros, et une extension pour déboguer.

Que changer, et à quel moment

Que changer, et à quel moment

Doivent être modifiés en premier :

- La syntaxe du langage d'entrée : les règles de grammaire.
- La syntaxe du langage de sortie : les chaînes de format du *pretty-printeur*.
- Le code produit par les extensions : les *quotations* (leur sémantique et non leur syntaxe).

Que changer, et à quel moment

Doivent être modifiés en premier :

- La syntaxe du langage d'entrée : les règles de grammaire.
- La syntaxe du langage de sortie : les chaînes de format du *pretty-printeur*.
- Le code produit par les extensions : les *quotations* (leur sémantique et non leur syntaxe).

Peuvent être modifiés dans un deuxième temps :

- Les types des structures de données présentes dans le code généré.
- Le nom des fonctions, modules et valeurs exportées.
- La syntaxe des sources, et des *quotations*.

Procédure de *bootstrap*

- D'abord changer les langages d'entrée et de sortie.
- Produire le traducteur (`camlp4boot`).
- Accorder le traducteur lui-même à ces changements.
- Promouvoir le nouveau traducteur.
- L'utiliser pour re-construire le traducteur.

La cible `make bootstrap` :

- Sauvegarde le `camlp4boot` courant (`make backup`).

Procédure de *bootstrap*

- D'abord changer les langages d'entrée et de sortie.
- Produire le traducteur (`camlp4boot`).
- Accorder le traducteur lui-même à ces changements.
- Promouvoir le nouveau traducteur.
- L'utiliser pour re-construire le traducteur.

La cible `make bootstrap` :

- Sauvegarde le `camlp4boot` courant (`make backup`).
- Fait la promotion du `camlp4boot` produit (`make promote`).

Procédure de *bootstrap*

- D'abord changer les langages d'entrée et de sortie.
- Produire le traducteur (`camlp4boot`).
- Accorder le traducteur lui-même à ces changements.
- Promouvoir le nouveau traducteur.
- L'utiliser pour re-construire le traducteur.

La cible `make bootstrap` :

- Sauvegarde le `camlp4boot` courant (`make backup`).
- Fait la promotion du `camlp4boot` produit (`make promote`).
- Nettoie l'environnement (`make clean`).

Procédure de *bootstrap*

- D'abord changer les langages d'entrée et de sortie.
- Produire le traducteur (`camlp4boot`).
- Accorder le traducteur lui-même à ces changements.
- Promouvoir le nouveau traducteur.
- L'utiliser pour re-construire le traducteur.

La cible `make bootstrap` :

- Sauvegarde le `camlp4boot` courant (`make backup`).
- Fait la promotion du `camlp4boot` produit (`make promote`).
- Nettoie l'environnement (`make clean`).
- Compile tout avec le nouveau `camlp4boot` (`make all`).

Procédure de *bootstrap*

- D'abord changer les langages d'entrée et de sortie.
- Produire le traducteur (`camlp4boot`).
- Accorder le traducteur lui-même à ces changements.
- Promouvoir le nouveau traducteur.
- L'utiliser pour re-construire le traducteur.

La cible `make bootstrap` :

- Sauvegarde le `camlp4boot` courant (`make backup`).
- Fait la promotion du `camlp4boot` produit (`make promote`).
- Nettoie l'environnement (`make clean`).
- Compile tout avec le nouveau `camlp4boot` (`make all`).
- Compare le résultat avec celui utilisé (`make compare`).

Procédure de *bootstrap*

- D'abord changer les langages d'entrée et de sortie.
- Produire le traducteur (`camlp4boot`).
- Accorder le traducteur lui-même à ces changements.
- Promouvoir le nouveau traducteur.
- L'utiliser pour re-construire le traducteur.

La cible `make bootstrap` :

- Sauvegarde le `camlp4boot` courant (`make backup`).
- Fait la promotion du `camlp4boot` produit (`make promote`).
- Nettoie l'environnement (`make clean`).
- Compile tout avec le nouveau `camlp4boot` (`make all`).
- Compare le résultat avec celui utilisé (`make compare`).
- Si le point fixe n'est pas atteint il faut relancer `make bootstrap`.

Plan

Historique des *locations* de Camlp4

- Couple d'entiers : positions depuis le début du fichier, et relecture pour obtenir plus d'informations comme les numéros de ligne.
- Couple de `Lexing.position` : contient déjà ces informations supplémentaires.
- `loc` et `_loc` : Changement dû à l'avertissement à propos des variables non utilisées.
- `Stream.count` et fonction de *location*.
- `value position: ref (ref int * ref int * ref string)`...

Abstraction

- Des opérations plus claires.
- Déboguage plus facile.

Choix d'implémentation envisageable

- Couple de `Lexing.position`, version actuelle.
- Couple d'entiers, version précédente.
- `Location.t`, comme dans OCaml.
- `unit` lorsque les *locations* sont inutiles.
- Ou une autre...

Passage à un style fonctionnel

- Suppression des références et autres structures mutables, tant que possible.
- Passage en argument de la *location* de départ (*parseur*, *lexeur*, etc.).
- L'analyseur lexical produit maintenant un flot de couples (lexème $\times$ *location*)
- La fonction de *location* est alors inutile.

Plan

Justification du renommage

- Introduction de nouveaux noms de modules «standards».
- Beaucoup de noms pas assez clairs.
- Obtenir un projet plus modulaire.
- Faciliter les futures transitions.

Fonctorisation

- Utilisation systématique de foncteurs.
- Organisation hiérarchique de modules.
- Séparation des signatures et des structures.
- Au final les seuls modules à la racine sont :
  - `Camlp4` : Le noyau du système.
  - `Camlp4Parsers` : Les extensions permettant de parser.
  - `Camlp4Printers` : Les modules d'affichage.
  - `Camlp4Filters` : Les filtres.
  - `Camlp4Top` : L'extension du toplevel.
  - `Camlp4Bin` : Le programme principal.

Module d'extension de syntaxe

```
open Camlp4
module Id =
struct let name = ... let version = ... end
(* An extension is just a functor: Syntax → Syntax *)
module Make (Syntax : Sig.Syntax.S) = struct
include Syntax ; ; open Camlp4.Sig.Camlp4Token
let foo = Gram.Entry.mk "foo"
EXTEND Gram
foo: [ [ ... ] ] ;
END ; ;
Gram.parse_string foo (Loc.mk "<string>") "foo x?"
end
(* Make it usable via the camlp4 binary. *)
module M = Register.SyntaxExtension(Id)(Make)
```

Module pré-construit : `Camlp4.PreCast`

- Toute cette fonctorisation est masquée à l'utilisateur débutant.
- Le module `PreCast` propose un assemblage par défaut.
- Il consiste en une série d'applications de foncteurs.
- Modules accessibles via `PreCast` : `Loc`, `Ast`, `Token`, `Lexer`, `Gram`, `Quotation`, `Syntax`, `AstFilters`.

Plan

Un nouvel analyseur lexical

- Utilise `ocamllex`, plutôt qu'un *parseur* de flots.
- Ré-entrant : plus d'état global.
- Style fonctionnel : dans la gestion des structures imbriquées.
- Ignorant et indépendant des mots-clés.
- Supporte les *quotations* imbriquées.
- Ne cache aucune information : commentaires, blancs...
- Un seul inconvénient : `[<<...>>` devient `[ <<...>>`

Plan

Camlp4 : un traducteur source-à-source

- Camlp4 parse et pretty-printe : Camlp4 traduit.
- Plusieurs *parseurs* et extensions : `OCaml`, `OCamlr`, `Grammar`...
- De *pretty-printeurs* : `OCaml`, `OCamlr`...
- Syntaxe originale vers révisée : `camlp4 -parser OCaml -printer OCamlr`.
- Désucrer la syntaxe : `camlp4boot -printer OCaml`.

Des filtres, et la transformation de programmes devient possible

- Un filtre se place entre le *parseur* et le *pretty-printeur*.
- Un filtre est une fonction qui transforme un Ast.
- Ces fonctions sont enregistrées auprès d'un gestionnaire.
- Utilisation commune :
-
  - Parcourir l'arbre en profondeur.
  - Filtrer un motif, le remplacer.
- `<< x + 0 >>` → `x`
- Pour cela des fonctions `map` sont nécessaires.

Map généré, filtrage aisé

- Filtrer toutes les catégories syntaxiques (17 catégories, 200 constructeurs).
- Solution retenue proche de celle d'Cαml :
  - Génère une classe à partir de la définition du type.
  - Propose une méthode par catégorie.
  - Il suffit d'en dériver.
  - De redéfinir une ou plusieurs méthodes.
  - De filtrer le motif voulu.
  - De déléguer les autres cas à la classe supérieure.

Filtres disponibles et exemple

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

- `ExceptionTracer`, `Tracer` : Décorent le corps des fonctions.
- `LiftCamlp4Ast` : Retourne le programme qui engendre cet arbre.

Plan

Réécriture des *pretty-printeurs*

Se rapprocher de la technologie standard

- Utilisation du module `Format`.
- Parcours récursif descendant.
- Parenthésage géré manuellement.

Les autres choix de design

- Une classe fonctionnelle (extensibilité, gestion de l'état).
- Filtrage à l'aide des *quotations*.

Priorités, et structures ambiguës

- Ambiguïté contextuelle du parenthésage :
  - Les structures comme `match`, `try`, et `function`.
  - L'utilisation des points-virgules.
- Résolution à l'aide de l'état que porte l'objet.

```
match x with
| A y →
let y = y * 2 in
(match y with
| ... → ...
| ... → ...)
| ... → ...
```

Plan

Les grammaires extensibles de Camlp4

```
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
```

Interface fonctorielle, ou pas

Deux modes étaient disponibles.

Interface fonctorielle, ou pas

Deux modes étaient disponibles.

Interface classique :

- Les entrées de grammaire sont du même type.
- Les grammaires sont des valeurs accessibles.

Interface fonctorielle :

- Paramétrée par un module de grammaire.
- Les entrées sont incompatibles entre grammaires.
- **La** grammaire du module n'est pas accessible.

Interface fonctorielle, ou pas

Deux modes étaient disponibles.

Interface classique :

- Les entrées de grammaire sont du même type.
- Les grammaires sont des valeurs accessibles.

Interface fonctorielle :

- Paramétrée par un module de grammaire.
- Les entrées sont incompatibles entre grammaires.
- **La** grammaire du module n'est pas accessible.

Maintenant :

- Statique : ancienne interface fonctorielle.
- Dynamique : interface classique paramétrée par un module.

Motifs de lexèmes

- Utilisation des motifs de lexèmes :
  - Les mots-clés : `"let"`, `"in"`, `"+"`, `":"`.
  - Les lexèmes paramétrés : `s = LIDENT`, `i = INT`.
  - Les autres lexèmes constants : `UIDENT "Foo"`.
- Limitations :
  - Résultat du type `string` seulement.
  - Pas d'alternative (*or pattern*).
  - Représentation forcée : `string * string`.
- Difficulté de changer de représentation.

Exemple de type algébrique des lexèmes

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

Exemple utilisant les nouveaux motifs de lexèmes

```
EXTEND Gram
expr:
[ [ "let"; `LIDENT i; "="; `INT (i, j);
`UIDENT "Foo"; "in"; `STRING _ → Let (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
```

Transformation d'une règle de grammaire

```
‘ANTIQUOT (("exp"|"" as n), a) → Ant (n, a)
(* becomes after transformation *)
(([ (* token pattern *)
Gram.Stoken
((function (* predicate in normal form *)
| ANTIQUOT (("exp" | ""), ) → true —→false),
(* the pretty-print string in normal form *)
"ANTIQUOT ((\"exp\" | \"\"), _)") ]),
 (Gram.Action.mk (* semantic action *)
    (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)))
```

Plan

Le système de *quotations*

Description

- Syntaxe des *quotations* : `<:expr<...>>`, `<:patt<...>>`.
- Syntaxe des *antiquotations* : `$expr$`, `$name:expr$`.

Utilité

- Inutile de connaître les 200 constructeurs d'Ast.
- Facilité d'écriture de larges Ast de programme.

Limitations du système précédent

- Antiquotation de liste in-intuitive : `<:expr< fun [ $list:pwel$ ] >>`.
- Syntaxe des *quotations* non-extensibles.

Parseurs standards et *parseurs* de *quotations*

- P0, P1 : *Parseur* simple, et expanseur de *quotations*.
- Peut-on obtenir un expanseur à partir d'un *parseur* simple (P1=P0+méta) ?
- Quelle serait cette «méta» opération ?
- Quelles sont les difficultés de mise en place ?
- Pourquoi a-t-il fallu changer l'Ast ?

Parseur P0, pour la règle de la flèche

Parseur standard (P0)

- Lit un flot de lexèmes et produit un AST.
- En simplifiant, `int → string` produit `TyArr (TyInt, TyStr)`.

```
EXTEND Gram
core_type:
[ [ t1 = SELF ; "→" ; t2 = SELF →
Ast.TyArr (_loc, t1, t2)
...
] ] ;
END
```

Parseur P1, pour la flèche

Parseur P1, expanseur de *quotations*

- Produit une expression représentant l'Ast.
- En simplifiant, `int → string` produit `ExApp (ExUid "TyArr",ExUid "TyInt",ExUid "TyStr")`

```
EXTEND Gram core_type:
[ [ meta_t1 = SELF ; "→" ; meta_t2 = SELF →
Ast.ExApp (_loc, Ast.ExApp (_loc, Ast.ExApp (_loc,
Ast.ExAcc (_loc, Ast.ExUid (_loc, "Ast"),
Ast.ExUid (_loc, "TyArr")),
Ast.ExLid (_loc, "_loc")),
meta_t1), meta_t2)
... ] ] ;
END
```

P0 et P1 avec les *quotations*

P0 et P1 peuvent se réécrire en utilisant P1.

P0 utilisant les *quotations*

```
t1 = SELF ; "→" ; t2 = SELF →
<:ctyp< $t1$ → $t2$ >>
```

P1 utilisant les *quotations*

```
t1 = SELF ; "→" ; t2 = SELF →
<:ctyp< Ast.TyArr _loc $t1$ $t2$ >>
```

Peut-on se contenter d'un seul *parseur*?

```
meta : <:ctyp< $t1$ -> $t2$ >> →
<:ctyp< Ast.TyArr _loc $meta t1$ $meta t2$ >>
```

Les *antiquotations*, le cauchemar des *quotations*

Pour se contenter d'un seul *parseur* :

- Inclure les règles pour les *antiquotations*.
- Utiliser des règles à la place des motifs de lexèmes.
- Placer les *antiquotations* dans l'AST.

Introduire les *antiquotations* dans l'AST :

- Sans tout dédoubler : `type` α `t = A of` α `| B of string`
- Introduction d'un échappement : "\\\$*name* :*code*"
- Ajout d'un cas aux catégories syntaxiques.
- Suppression des listes et des optionnels.
- Utilisation d'opérateurs binaires et de `Ast.XxNil`.

Exemple avec les règles de filtrage

Type des règles de filtrage

```
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
```

Grammaire abstraite des règles de filtrage

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

Facilité d'écriture

Avant il fallait :

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

Maintenant il suffit de :

```
<:expr< fun [ $mc1$ | $mc2$ | _ → error... ] >>
```

Quotations étendues, écriture détendue

On peut désormais écrire :

```
<:expr< arr.{e1, e2, e3, e4} >>
<:expr< parser [: ‘42 ; s :] → s >>
<:expr< EXTEND Gram expr: [[ ... ]] ; END >>
<:expr< fun [ <:expr< $x$ + $y$ >> →
<:expr< $y$ + $x$ >> ] >>
```

De plus en syntaxe originale :

```
<:expr< e1 ; e2 ; e3 >>
<:expr< arr.{e1, e2, e3, e4} >>
<:expr< parser [< ’42 ; s >] → s >>
<:expr< function <:expr< $x$ + $y$ >> →
<:expr< $y$ + $x$ >> >>
```

Factorisation des grammaires

Pour changer la syntaxe officielle.

Avant, trois *parseurs* :

- La syntaxe originale : `pa_o`.
- La syntaxe révisée : `pa_r`.
- La syntaxe des *quotations* : `q_MLast`.

Maintenant, un seul car :

- `Camlp4OCamlRevisedParser` : pose la plupart des règles.
- `Camlp4OCamlParser` : l'adapte à la syntaxe originale.
- `Camlp4Quotation{Common,Expander}` : utilisent les fonctions de méta-élévation.

Conclusion

- Les changements effectués ont porté sur : le *bootstrap*, les *locations*, l'organisation, le *lexeur*, les filtres, les *pretty-printeurs*, les motifs de lexèmes, l'interface fonctorielle, et les *quotations* extensibles.
- Cette version n'est ni rétro-compatible ni parfaite.
- Elle pose cependant une base solide à la maintenance du projet.

Questions