Présentation d'alphaCaml

François Pottier

15 juillet 2005

Constat

Les langages de programmation dont nous disposons ne sont adaptés à la manipulation d'objets symboliques qu'en l'absence de notion de *liaison* de noms.

Le respect de la portée lexicale doit être *codé* à la main (opérations de renommage, de substitution sans capture, etc.), ce qui est lourd et fragile.

Il nous faudrait une approche plus *déclarative*, plus *robuste*, plus *automatisée*.

Trois facettes

On peut distinguer trois facettes du problème: définir

- un *langage de spécification*,
- une *technique d'implantation*,
- une *traduction automatique* du premier vers la seconde.

L'existant

On trouve dans la littérature un petit nombre de propositions d'ajouter aux langages de spécification algébriques une notion de *nom* et une construction, dite *abstraction*, permettant la liaison.

L'abstraction prend en général la forme $\langle a \rangle e$, ou $\langle a_1, \ldots, a_n \rangle e$, ou bien, comme en Fresh Objective Caml, $\langle e_1 \rangle e_2$.

Dans tous les cas, l'abstraction est binaire: les noms (ou *atomes*) a qui apparaissent à gauche sont liés et leur portée est l'expression e qui apparaît à droite.

Exemple: le λ-calcul pur

Le λ-calcul pur:

$$M := a \mid M\,M \mid \lambda a.M$$

est modélisé ainsi en Fresh Objective Caml:

```
type brand and var = brand name

type term =
  | EVar of var
  | EApp of term * term
  | ELam of <var>term
```

Un exemple plus délicat

Ajoutons des définitions *simultanées*:

$$M ::= \ldots \mid \mathsf{let}\ a_1 = M_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ a_n = M_n\ \mathsf{in}\ M$$

Les atomes a_i sont liés, donc, en Fresh Objective Caml, doivent apparaître *dans le membre gauche* de l'abstraction. Les termes M_i sont situés hors de la portée lexicale de ces atomes, donc doivent apparaître *hors* de l'abstraction:

```
type term =
  | ...
  | ELet of term list * ⟨var list⟩term
```

Un second exemple délicat

Ajoutons des définitions *récursives* simultanées:

$$M ::= \ldots \mid \textsf{letrec } a_1 = M_1 \textsf{ and } \ldots \textsf{ and } a_n = M_n \textsf{ in } M$$

Les atomes a_i sont liés, donc, en Fresh Objective Caml, doivent apparaître *dans le membre gauche* de l'abstraction. Les termes M_i sont situés dans la portée lexicale de ces atomes, donc doivent apparaître *dans le membre droit* de l'abstraction:

```
type term =
  | ...
  | ELetRec of ⟨var list⟩(term list * term)
```

Une solution

La racine du problème est l'idée que *structure lexicale* et *structure physique* doivent coïncider.

Un sous-terme doit pouvoir être physiquement rattaché à une abstraction sans pour autant être interprété comme contribuant des noms liants ou comme appartenant à la portée lexicale de l'abstraction.

Une solution (suite)

Au sein d'une abstraction, alphaCaml distingue trois types de composants élémentaires: des *noms liants*, des expressions situées *dans* la portée lexicale de l'abstraction, et des expressions situées *hors* de cette portée.

Ces composants sont assemblés à l'aide de sommes et de produits, ce qui donne lieu à une nouvelle catégorie syntaxique, les *motifs*. L'abstraction devient *unaire*; son contenu est un motif.

Définition simplifiée du langage

$$s ::= \mathsf{inner} \mid \mathsf{outer} \qquad \textit{Spécificateurs de portée}$$

$$t ::= \mathsf{unit} \mid t \times t \mid t + t \mid \mathsf{atom} \mid \langle u \rangle \qquad \textit{Types d'expression}$$
$$u ::= \mathsf{unit} \mid u \times u \mid u + u \mid \mathsf{atom} \mid s\ t \qquad \textit{Types de motif}$$

$$e ::= () \mid (e, e) \mid \mathsf{inj}_i\ e \mid a \mid \langle p \rangle \qquad \textit{Expressions}$$
$$p ::= () \mid (p, p) \mid \mathsf{inj}_i\ p \mid a \mid s\ e \qquad \textit{Motifs}$$

Une spécification alphaCaml est essentiellement la donnée d'une famille de types d'expression et de motif.

Retour au λ-calcul pur

Le λ-calcul pur est modélisé ainsi en alphaCaml:

```
sort var

type term =
  | EVar of atom var
  | EApp of term * term
  | ELam of ⟨lamp⟩

type lamp binds var =
    atom var * inner term
```

La liaison simultanée, revue et corrigée

Les définitions simultanées se modélisent maintenant sans difficulté:

```
type term =
  | ...
  | ELet of ⟨letp⟩

type letp binds var =
    binding list * inner term

type binding binds var =
    atom var * outer term
```

Exemples plus avancés

La syntaxe abstraite des motifs d'un langage de programmation proche d'Objective Caml pourrait s'écrire

```
type pattern binds var =
  | PWildcard
  | PVar of atom var
  | PRecord of pattern StringMap.t
  | PInjection of [ constructor ] * pattern list
  | PAnd of pattern * pattern
  | POr of pattern * pattern
```

Lorsqu'une valeur de type *pattern* apparaît au sein d'une abstraction, tous les noms situés aux feuilles *PVar* sont considérés comme liants.

Exemples plus avancés (suite)

Le type *pattern* peut être utilisé par exemple pour définir une construction "case":

```
type term =
  | ...
  | ECase of term * branch list

type branch =
    ⟨clause⟩

type clause binds var =
    pattern * inner term
```

Exemples plus avancés (suite)

Le langage permet la déclaration de multiples *sortes* d'atomes:

```
sort typevar

type typ =
  | TVar of atom typevar
  | ...

type term =
  | ...
  | ETypeAnnotation of term * typ

type pattern binds var =
  | ...
  | PTypeAnnotation of pattern * neutral typ
```

La technique de de Bruijn

Les noms sont représentés par des *entiers* interprétés de façon relative.

Avantages: l'alpha-équivalence coïncide avec l'égalité. Aucun générateur de noms frais n'est nécessaire.

Inconvénients: l'interprétation des termes ouverts est relative à un contexte, ce qui exige de "décaler" lorsque le contexte change. Décalages incorrects et oublis de décaler sont possibles, ce qui rend l'approche *fragile*.

La technique standard

Les noms sont représentés par des *atomes* disposant en principe uniquement d'une notion d'égalité. Ceux-ci peuvent être représentés par des entiers, interprétés alors de façon absolue.

Avantage: l'interprétation des termes ouverts n'est plus relative à un contexte, donc on ne "décale" plus.

Inconvénient: les abstractions doivent être "rafraîchies" à l'ouverture, ce qui exige un générateur de noms frais.

Mon choix

alphaCaml adopte la technique standard.

Les atomes sont représentés par des paires d'un entier et d'un nom, lequel ne définit pas l'identité de l'atome mais sert de suggestion pour l'affichage. Les ensembles d'atomes et les renommages sont représentés par des arbres de Patricia.

Les renommages sont *suspendus* et *composés* aux abstractions, ce qui permet des parcours d'arbre en temps linéaire.

Bien que le générateur ait un état modifiable, on peut sans risque sérialiser et désérialiser des termes *clos*.

Un traducteur indépendant

alphaCaml se présente comme la combinaison d'un *outil* indépendant et d'une *librairie*.

À titre de comparaison, Fresh Objective Caml est une extension d'Objective Caml, tandis que FreshLib est une simple librairie Haskell.

Que produit alphaCaml?

alphaCaml produit *deux versions* des définitions de types, ainsi qu'un ensemble d'opérations.

La version *brute* des définitions de types sert en principe lors de l'analyse syntaxique et lors de l'affichage.

La version *interne* sert à toutes les opérations exigeant le respect de la liaison lexicale.

Types bruts

La spécification du λ-calcul pur est traduite ainsi en types bruts:

```
type var = Identifier.t

type term =
  | EVar of var
  | EApp of term * term
  | ELam of lamp

and lamp =
    var * term
```

Noms et abstractions sont *concrets*.

Types internes

La spécification du λ-calcul pur est traduite ainsi en types internes:

```
type var = Var.Atom.t

type term =
  | EVar of var
  | EApp of term * term
  | ELam of opaque_lamp

and lamp =
    var * term

and opaque_lamp
```

Noms et abstractions sont *abstraits*.

Import/export

Le code de conversion entre formes brute et interne est engendré *automatiquement*:

```
val import_term : var Identifier.Map.t → Raw.term → term
val export_term : Var.AtomIdMap.t → term → Raw.term
```

Un environnement associant identificateurs et atomes doit être fourni. Il est vide si le terme est clos.

Emploi des abstractions

L'ouverture d'une abstraction implique son rafraîchissement:

val *open_lamp* : *opaque_lamp* → *lamp*
val *create_lamp* : *lamp* → *opaque_lamp*

Ceci impose le respect de la "*convention de Barendregt*".

Et d'autres plaques de bouilloire...

alphaCaml engendre des fonctions permettant de

- calculer l'ensemble des noms libres d'une expression,
- calculer l'ensemble des noms libres ou liants d'un motif,
- appliquer une substitution à un motif ou expression.

Il engendre également des *classes* Objective Caml permettant de définir aisément des *transformations* ou *parcours* de termes. Grâce à quoi, un client peut ensuite définir la substitution sans capture en moins de dix lignes...

Substitution sans capture

Voici comment un client définit la substitution sans capture:

```
class typeSubst (subst : type_expr Type_var.AtomMap.t) = object
  inherit map
  method tvar tyX =
    try
      Type_var.AtomMap.lookup tyX subst
    with Not_found →
      TVar tyX
end

let typeSubst subst tyT =
  (new typeSubst subst)#type_expr tyT
```

Situation actuelle

alphaCaml est *disponible*. Très peu d'utilisateurs se sont manifestés jusqu'ici.

La distribution comprend *deux démos*:

- un typeur et évaluateur naïf pour $F_{\leq}$
- un évaluateur naïf pour un calcul de mixins (Hirschowitz *et al.*)

Cette expérience limitée est encourageante.

Limitations

La nécessité de passer par les fonctions **open** pour examiner le contenu des abstractions *empêche le filtrage en profondeur*.

Comme en Fresh Objective Caml, le client peut écrire du code *dénué de sens*, par exemple une fonction qui renvoie l'ensemble des variables liées d'une expression.

Vers alpha-(ici-votre-prouveur-favori)?

Pitts, Urban et Tasson, et d'autres s'intéressent à la traduction d'un langage de spécification de haut niveau, similaire à celui d'alphaCaml, en *théorèmes et preuves*. L'idée est de produire et prouver automatiquement les principes de récursion et d'induction modulo alpha-équivalence correspondant à la spécification.

Le langage cible étant plus expressif, on *peut* être plus ambitieux et espérer interdire la fuite de noms frais. On *doit* probablement même le faire pour obtenir des principes cohérents et pour éviter le recours à un générateur de noms frais.