Analyse sémantique élémentaire
22 septembre 2011

Introduction

Rôle de l'analyse sémantique:

Entrée : arbre de syntaxe abstraite issu de l'analyse syntaxique
Sortie :

- un arbre de syntaxe abstraite décoré: résultat de la compréhension du programme
- une erreur si le programme est incorrect

Outils théoriques : descriptions sémantiques.
Outils logiciels : programmes récursifs sur les arbres de syntaxe abstraite.

Sommaire

Analyse de portée: principes

- Repérer les utilisations d'objets non déclarés.
- Relier les déclarations d'objets (variables, types, procédures, fonctions . . .) et leur utilisation.
- Une fois ce lien réalisé, le sauvegarder pour les autres phases.

Table des symboles

- Des informations doivent être collectées pour chaque identificateur introduit:
  - Nature de l'objet: variable, procédure, type; variable globale, locale, paramètre ...
  - Type d'une variable; signature d'une fonction.
  - Allocation en mémoire (emplacement, taille ...)
- Ces informations serviront à chaque utilisation de l'objet.
- Création d'un lien direct vers les informations pour chaque utilisation.

Arbres de syntaxe annotés

Plusieurs représentations possibles pour lier les utilisations et les déclarations :

- Chaque utilisation est un pointeur sur la déclaration.
- Chaque utilisation est un nom unique ou bien un numéro faisant référence à une table annexe qui contient les informations sur les déclarations.
  - Possibilité d'utiliser différentes tables suivant l'étape de compilation.

L'analyse de portée va permettre de construire cette nouvelle représentation de l'arbre.

Vérifier la portée

- **Entrée** : l'arbre de syntaxe abstraite issu de l'analyse syntaxique
- **Sortie** : un nouvel arbre de syntaxe abstraite avec des identificateurs uniques et une table qui stocke des informations sur les variables (globale ou locale).
- **Erreur** : si une variable a été utilisée sans être déclarée ou bien en dehors de son scope (portée).
- **Technique** : utiliser une liste d'associations intermédiaire entre les noms visibles et les déclarations de l'environnement identifiées par un nom unique.

Sommaire

Langage ARITH étendu avec des fonctions

```
type ident = string
type binop = Sum | Diff | Prod | Quot
type expr =
    Cst of int
  | Var of ident
  | Op of expr * binop * expr
  | Letin of ident * expr * expr
  | App of ident * expr list
type instr = Set of ident * expr | Print of expr
           | Fun of ident * ident list * expr
type prg = instr list
```

- L'analyse syntaxique construit un objet de type `prg`.
- L'analyse de portée produira un nouvel objet de type `prg` + un tableau identifiant chaque déclaration.

Exemple de programme

```
set x = 4
fun f (x,y) = 2 * x + y
set xx = let y = f(x,5) in - (y * y)
```

Arbre issu de l'analyse syntaxique:

```
[Set("x",Cst 4);
 Fun("f",["x","y"],Op(Cst 2,Mult,Var "x"),Plus,Var "y"),
 Set("xx",Letin("y",App("f",[Var "x",Cst 5])),
                  Op(Cst 0,Diff,Op(Var "y",Mult, Var "y")))))]
```

Après analyse de portée:

- Des noms uniques pour les objets.
- Une table des symboles qui conserve les informations sur les déclarations:
  - le nom original (impression),
  - la nature de la variable, sa “position”,
  - le nombre de paramètres des fonctions . . .

```
[Set("x_0",Cst 4);
 Fun("f_3",["x_1","y_2"],
           Op(Cst 2,Mult,Var "x_1"),Plus,Var "y_2"),
 Set("xx_5",
     Letin("y_4",App("f",[Var "x_0",Cst 5])),
           Op(Cst 0,Diff,Op(Var "y_4",Mult, Var "y_4")))))]
```

x_0	Global(1)	x
x_1	Param(1)	x
y_2	Param(2)	y
f_3	Fun(2)	f
y_4	Local(1)	y
xx_5	Global(2)	xx

Sommaire

Règles de portées

$\rho \vdash e$ **ok** dans l'environnement où les variables de ρ sont visibles, l'expression e respecte les règles de portées.

$$\frac{\rho \vdash e_1 \textbf{ ok} \quad \rho \vdash e_2 \textbf{ ok}}{\rho \vdash \texttt{Op}(e_1, op, e_2) \textbf{ ok}} \qquad \frac{x \in \rho}{\rho \vdash \texttt{Var}(x) \textbf{ ok}}$$

$$\frac{\rho \vdash e_1 \textbf{ ok} \quad \rho + x \vdash e_2 \textbf{ ok}}{\rho \vdash \texttt{Letin}(x, e_1, e_2) \textbf{ ok}}$$

$$\frac{(\rho \vdash e_i \textbf{ ok})_{i=1..n} \quad f \in \rho \cup \text{PRIM}}{\rho \vdash \texttt{App}(f, [e_1; \ldots; e_n]) \textbf{ ok}}$$

$$\frac{\rho \vdash e \textbf{ ok} \quad \rho + x \vdash p \textbf{ ok}}{\rho \vdash (\texttt{Set}(x, e) :: p) \textbf{ ok}} \qquad \frac{\rho \vdash e \textbf{ ok} \quad \rho \vdash p \textbf{ ok}}{\rho \vdash (\texttt{Print}(e) :: p) \textbf{ ok}}$$

$$\frac{\rho + l\ [+f] \vdash e \textbf{ ok} \quad \rho + f \vdash p \textbf{ ok}}{\rho \vdash (\texttt{Fun}(f, l, e) :: p) \textbf{ ok}}$$

Reconstruire l'ast décoré

$\rho \vdash e \rightsquigarrow e', d$

- e respecte les règles de portées dans l'environnement ρ
- e' ast correspondant avec des identifiants uniques pour les variables
- d ensemble des déclarations: pour chaque variable x, le nom dans le programme original et la nature:
  fonction; variable locale, globale ou paramètre.

Reconstruire l'ast décoré (2)

$\rho \vdash e \rightsquigarrow e', d$

$$\frac{\rho \vdash e_1 \rightsquigarrow e'_1, d_1 \quad \rho \vdash e_2 \rightsquigarrow e'_2, d_2}{\rho \vdash \texttt{Op}(e_1, op, e_2) \rightsquigarrow \texttt{Op}(e'_1, op, e'_2), d_1 \cup d_2}$$

$$\frac{(x \rightsquigarrow n) \in \rho}{\rho \vdash \texttt{Var}(x) \rightsquigarrow \texttt{Var}(n), \emptyset}$$

$$\frac{\rho \vdash e_i \rightsquigarrow e'_i, d_i \quad (f \rightsquigarrow f') \in \rho}{\rho \vdash \texttt{App}(f, [e_1; \ldots; e_n]) \rightsquigarrow \texttt{App}(f', [e'_1; \ldots; e'_n]), \bigcup_i d_i}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow e'_1, d_1 \quad (n, d) = \texttt{new}(x, \texttt{Loc}) \quad \rho + (x \rightsquigarrow n) \vdash e_2 \rightsquigarrow e'_2, d_2}{\rho \vdash \texttt{Letin}(x, e_1, e_2) \rightsquigarrow \texttt{Letin}(n, e'_1, e'_2), \{d\} \cup d_1 \cup d_2}$$

Reconstruire l'ast décoré (3)

$\rho \vdash p \rightsquigarrow p', d$

$$\frac{\rho \vdash e \rightsquigarrow e', d_1 \quad \rho \vdash p \rightsquigarrow p', d_2}{\rho \vdash (\texttt{Print}(e) :: p) \rightsquigarrow (\texttt{Print}(e') :: p'), d_1 \cup d_2}$$

$$\frac{\rho \vdash e \rightsquigarrow e', d_1 \quad (n, d) = \texttt{new}(x, \texttt{Glob}) \quad \rho + (x \rightsquigarrow n) \vdash p \rightsquigarrow p', d_2}{\rho \vdash (\texttt{Set}(x, e) :: p) \rightsquigarrow (\texttt{Set}(n, e') :: p'), \{d\} \cup d_1 \cup d_2}$$

$$\frac{\begin{array}{c}((n_i, d_i) = \texttt{new}(x_i, \texttt{Par}))_i \quad \rho + (x_i \rightsquigarrow n_i)_i \vdash e \rightsquigarrow e', d_e \\ (f', d_f) = \texttt{new}(f, \texttt{Fun}) \quad \rho + (f \rightsquigarrow f') \vdash p \rightsquigarrow p', d_p\end{array}}{\rho \vdash (\texttt{Fun}(f, [x_1; ..; x_p], e) :: p) \rightsquigarrow (\texttt{Fun}(f', [n_1; ..; n_p], e') :: p'), \bigcup_i d_i \cup d_e \cup d_f \cup d_p}$$

Remarque sur l'analyse de portée

Il faut bien distinguer:

- La table des symboles qui contient l'ensemble des déclarations locales et globales incluses dans le programme et qui fait partie de la représentation du programme.
- L'environnement local ρ qui conserve les variables visibles en un point de programme et n'est plus utilisé après l'analyse de portée.

Sommaire

Fonction d'analyse - structures de données

Table des symboles

peut être globale

```
exception ScopeError
type annot = Glob | Loc | Par | Func
(* store declaration in table, returns unique identifier *)
val add_decl : ident -> annot -> ident
```

Variables visibles

liste d'associations ou structure de map fonctionnelle qui associe un entier à une chaîne.

```
type t
val empty_vis : t
val add_vis : ident -> ident -> t -> t
(* raise ScopeError if not found *)
val find_vis : ident -> t -> int
```

Implantation

Génération de nouveaux noms:

```
let new_name = let i = ref 0 in
    fun x -> incr i; x ^ "_" ^ (string_of_int !i)
```

Création d'une table de hachage globale comme table des symboles.

```
let tsymb = Hashtbl.create 17
let add_decl x a = let x' = new_name x in
    Hashtbl.add tsymb x' (x,a); x'
```

Implantation (2)

Structure de `map` fonctionnelle pour l'environnement des variables visibles:

```
module Ident =
  struct type t = ident let compare = compare end

module Vis = Map.Make(Ident)
type t = ident Vis.t
let (empty_vis : t) = Vis.empty
let (add_vis : ident -> ident -> t -> t) = Vis.add
let find_vis (x : ident) (vis:t) =
    try Vis.find x vis with Not_found -> raise ScopeError
```

Fonction d'analyse des expressions

```
let rec scope_expr vis = function
    Var x -> Var (find_vis x vis)
  | Cst n as x -> x
  | Op (o,e1,e2) -> Op (o,scope_expr vis e1,
                        scope_expr vis e2)
  | Letin(x,e1,e2) -> let e1' = scope_expr vis e1
                      and x' = add_decl x Loc
                      in let vis' = add_vis x x' vis
                      in Letin(x',e1',scope_expr vis' e2)
  | App(f,le) -> let f' = find_vis f vis in
    let le' = List.map (scope_expr vis) le in App(f',le')
```

Fonction d'analyse des programmes

```
let rec scope_prog vis = function
    [] -> []
  | Print(e)::prog ->
      Print(scope_expr vis e)::scope_prog vis prog
  | Set(x,e)::prog ->
      let x' = add_decl x Glob in
      let vis' = add_vis x x' vis in
        Set(x',scope_expr vis e)::scope_prog vis' prog
  | Fun(f,lv,e)::prog ->
      let vise,lv' = List.fold_right
        (fun x (v,l) -> let x' = add_decl x Par in
                        let v' = add_vis x x' v in
                        (v',x'::l))
        lv (vis,[]) in
      let f' = add_decl f Func in
      let vis' = add_vis f f' vis in
        Fun(f',lv',scope_expr vise e)::scope_prog vis' prog
```

Remarques

- La tables des symboles est globale, elle comporte toutes les déclarations du programme et sera utilisée dans les étapes suivantes de la compilation.
- L'environnement des variables visibles est une structure temporaire qui varie suivant le point du programme. L'implantation fonctionnelle (persistante) permet de ne pas avoir à retirer d'éléments.

Variantes

- On peut aussi avoir des constructeurs différents `GVar`, `LVar`, `PVar` pour distinguer les variables locales, globales et les paramètres.

- On peut introduire des tables différentes en fonction de la nature des déclarations (variables, fonctions, types, . . .) si l'utilisation permet de choisir la bonne table.

- L'analyse de portée peut aussi être faite en même temps que le typage (par exemple ici vérifier que le nombre d'arguments à l'appel d'une fonction correspond à la déclaration).

- On s'est placé dans le cadre où la portée est résolue de manière syntaxique. Dans le cas de surcharge, des informations de typage peuvent être nécessaires.

Sommaire

Le typage

Diviser l'espace des valeurs en collections: un type est une collection de valeurs partageant une même propriété.

Exemple: représentation des flottants ou de structures chaînées.

Les types sont toujours plus ou moins vérifiés:

- soit à l'exécution (typage dynamique)
- soit à la compilation (typage statique)

Intérêts du typage statique

- détection précoce d'erreurs;
- documentation, structuration (une première information sur le programme);
- élimination de certaines erreurs d'exécution; plus de sûreté;
- une manière générale de décrire des analyses de programmes (e.g., analyses d'exception, effets de bords, sécurité du code).

Typage statique vs dynamique

- le typage dynamique est plus précis
  (`if B then 1 else 2+"a"`)
  avec `B` qui s'évalue en `true`
- mais les erreurs de type dépendent des valeurs à l'exécution
- les langages typés statiquement peuvent être compilés plus efficacement (e.g., pas de test de représentation à l'exécution), facilite les optimisations (enregistrements, accès, représentations,...)
- il est possible d'utiliser une même représentation pour deux valeurs de types différents sans risque de confusion à l'exécution, e.g.:
  - représenter `true` et `false` par des entiers;
  - les constructeurs de valeur de ocaml par des entiers consécutifs.

Propriété attendue du typage statique

le typage statique doit être conservatif :

- si le programme est bien typé statiquement alors il n'y a pas d'erreur de type à l'exécution :

$$\frac{e : \tau \quad \text{le calcul de } e \text{ termine sans erreur}}{e \rightsquigarrow v \in [\tau]}$$

- Contre exemple:

```
union test {
  int champ1;
  float champ2;
};
main() {union test s;
        s.champ2 = 3.3e15;
        printf("%d\n", s.champ1);}
```

La confusion entier/adresse donne des segmentation fault à l'exécution

Remarques

- Trouver des systèmes de types les plus riches possibles
- Des langages comme Java associent du typage statique et dynamique (le type dynamique fait partie de la valeur de l'objet)

Dans ce cours

- On étudie ici les principes du typage statique.
- Comment formuler les règles de typage ?
- Principes des algorithmes pour les mettre en oeuvre.

Vocabulaire

Qu'est qu'un type?

- Un type est une expression d'un langage de types
- Déclarer un type = introduire un nom de type
  Exemple : déclarations de **struct** ou **enum** ...
- Définir un type = associer un nom à une exp. de type
  Exemple : utilisation de **typedef**.

Typer une expression

- Typer = associer un type à une expression
- Typage fort = toute expression a un type principal

Vérification *vs* Inférence

Vérification des types le programmeur associe un type à chaque déclaration d'identificateur et le compilateur vérifie la correction (e.g, C, C++, Pascal, Ada,...)

```
int addition (int x1, int x2)
  int temp;
   temp = x1 + x2;
   return(temp);
```

Synthèse des types les types sont devinés (on dira synthétisés ou inférés) par le compilateur par une analyse des contraintes d'utilisation des variables (e.g, Caml, SML,...)

```
let addition x1 x2 =
 let temp = x1 + x2 in temp
val addition : int -> int -> int = <fun>
```

Sommaire

Règles d'inférence

La vérification de types se fait par rapport à des règles de typage.

"le programme P est bien typé de type ty"

$$P : ty$$

Vérification

- vérifier que ty est un type valide pour P

Synthèse

- trouver un ty valide

Typer une expression: parcourir l'expression en associant un type à chaque sous-expression

Règles de typage

- des axiomes:

$$P : ty$$

- des règles d'inférence

$$\frac{P_1 : ty_1 \quad P_n : ty_n}{C(P_1, ..., P_n) : ty}$$

- typer = construire un arbre où les feuilles sont des axiomes et les noeuds, des règles d'inférence
- Un programme est bien typé lorsque l'on peut construire une telle déduction

$$\frac{\begin{array}{cc}\vdots & \vdots \\ P_1 : ty_1 & P_n : ty_n\end{array}}{C(P_1, ..., P_n) : ty}$$

Exemple

Expressions arithmético-logiques

Axiomes:

$$\texttt{Cte}(i) : \texttt{int} \quad \texttt{true} : \texttt{bool} \quad \texttt{false} : \texttt{bool}$$

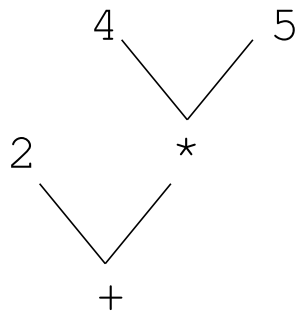
Une règle de déduction:

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int} \quad op \in \{\texttt{Plus}, \texttt{Mult}, \texttt{Div}, \texttt{Minus}\}}{\texttt{Op}(e_1, op, e_2) : \texttt{int}}$$

Exemples

Expression 1 : `2 + (4 * 5)`

$$\frac{2 : \texttt{int} \quad \dfrac{4 : \texttt{int} \quad 5 : \texttt{int}}{4 * 5 : \texttt{int}}}{2 + (4 * 5) : \texttt{int}}$$



Expression 2 : `2 + (true * 5)`

$$\frac{2 : \texttt{int} \quad \dfrac{\texttt{true} : \texttt{bool} \quad 5 : \texttt{int}}{\texttt{true} * 5 : ?}}{2 + (\texttt{true} * 5) : ?}$$

Identificateurs

Question: Que faire en présence d'identificateurs?

```
int x = 1;
int y;
y = 2 + (4 * x);
```

On introduit l'environnement de typage : *Env*.

- contient des liaisons (*idf*, *type*)
- $\langle(\texttt{y} : \texttt{int})(\texttt{x} : \texttt{int})\rangle = \mathit{Env}$

Notations

"Dans l'environnement *Env*, le programme *P* a le type *ty*"

$$Env \vdash P : ty$$

L'environnement de typage

- ensemble d'associations $\langle (x_1 : ty_1), ..., (x_n : ty_n) \rangle$
- une fonction d'accès *acces* tq:
  $$acces\ (x, \langle (x_1 : ty_1), ..., (x_n : ty_n) \rangle)$$
  $$= \begin{cases} ty_1 \text{ si } x = x_1 \\ acces\ (x, \langle (x_2 : ty_2), ..., (x_n : ty_n) \rangle) \text{ sinon} \end{cases}$$
- L'analyse de portée faite préalablement assure l'unicité des identifiants.

Sommaire

Langage de types

- *Types de base*: `int`, `float`, `char`, `string`,...
- *Types construits*
  - constructeur de type (opérateur sur les types). e.g, `int[]`

```
type typ = Tbool | Tint | Tfloat | Tchar | Tstring ...
         | Tarr of typ
```

On pourrait aussi avoir:

- des abréviations de type `Tdef of ident`
  des types définis par l'utilisateur `Tclass of ident`
- gestion de l'égalité, d'une table des symboles de types

Langage

```
type cte = Int of int | Float of float | Bool of bool ...
type primop = Sum | Diff | Prod | Quot | ...
type oper = Prim of primop | User of ident
type expr =
    Cst of cte
  | Var of ident
  | Op of oper * expr list
  | Letin of ident * expr * expr
type instr = Set of ident * expr | Print of expr
  | Fun of ident * (ident * typ) list * expr
type prg = instr list
```

Signature

On associe à une fonction une signature qui spécifie

- le nombre d'arguments attendus
- le type de chaque argument
- le type de retour

Plusieurs sortes d'opérateurs:

- opérateurs primitifs (arithmétiques) : la signature est fixée
- opérateurs génériques (accès dans un tableau) : la signature va dépendre du type des arguments
- opérateurs définis par l'utilisateur : la signature est donnée par le programme.

Une signature peut se représenter par le type ocaml suivant :

```
type sign = typ list * typ
```

Algorithme de typage - prérequis

Chaque expression a un type unique qui peut être inféré.
Une table associe à chaque déclaration son type.

```
val add_typ : ident -> typ -> unit
val find_typ : ident -> typ
```

De même pour les signatures des opérateurs:

```
val add_sign : ident -> sign -> unit
val find_sign : ident -> sign
```

Algorithme de typage

```
let rec type_expr = function
  Cte c -> type_cte c
| Var x -> find_typ x
| Op(op,le) -> let (lt,t) = find_sign op in
               check_lexpr lt le; t
| LetIn(x,e1,e2) -> let t1 = type_expr e1 in
                    add_typ x t1; type_expr e2
and check_lexpr = function
  [],[] -> ()
| t::lt,e::le
        -> if eqt (type_expr e) t then check_lexpr lt le
           else raise TypeError
| _,_ -> raise TypeError
```

Algorithme de typage (2)

```
let rec type_prog = function
  [] -> ()
| Set(x,e)::p
   -> let t = type_expr e
      in add_typ x t; type_prog p
| Print e::p -> let _ = type_expr e in  type_prog p
| Fun(f,lv,e)::p
   -> let lt = List.map (fun (x,t) -> add_typ x t;t) lv in
      let t = type_expr e
      in add_sign f (lt,t); type_prog p
```

Opérateurs génériques

Certains opérateurs n'ont pas une signature unique:

- conditionnelle **if** b **then** e_1 **else** e_2 a pour type τ si b : `bool` et $e_1, e_2 : \tau$.
- accès dans un tableau $t[n] : \tau$ si t de type τ `array` et n : `int`.

On leur associe souvent des constructeurs explicites dans la syntaxe abstraite.

```
| If(b,e_1,e_2) ->
    let tb = type_expr b
    and t1 = type_expr e1 and t2 = type_expr e2
    in if eqt tb Tbool && eqt t1 t2 then t1
       else raise TypeError
| Gop(op,le) -> let lt = List.map type_expr le in
                find_and_check_sign op lt
```

Le typage polymorphe à la ocaml permet de traiter uniformément ce cas.

Remarques

- Il est nécessaire de comparer les types suivant des règles qui dépendent du langage quand le système de type est complexe.
  Utilisation d'une fonction spécifique `eqt` et pas l'égalité structurelle.
- Notre algorithme utilise le fait que la portée a été préalablement résolue, sinon il faut introduire un environnement ρ qui contient les variables visibles.
- Il n'y a pas d'annotations de type dans les instructions **set** car les variables sont initialisées (leur type est calculé à partir du type de l'expression d'initialisation)
- Si les fonctions sont récursives, alors il faut aussi donner le type de retour attendu de la fonction: