APS - Notes de Cours

Jordi Bertran de Balanda

Cours 1

Introduction

Programme

Description statique (fichier) d'un comportement dynamique (exécution). Statique: exécutable/bytecode/assembleur/code source Code source: suite de caractères obéissant à des règles de syntaxe.

Syntaxe:

- Lexique: ensemble d'unités lexicales (mots/symboles)
- Grammaire: règles d'agencement des mots

définit les suites de caractères qui sont des programmes

donne un procédé de construction d'un mécanisme de reconnaissance des programmes (automates, langages formels)

Sémantique

- opérationnelle (grand pas/petit pas)
- dénotationnelle (lambda-calcul)
- axiomatique (cf. logique de Hoare en CPS)

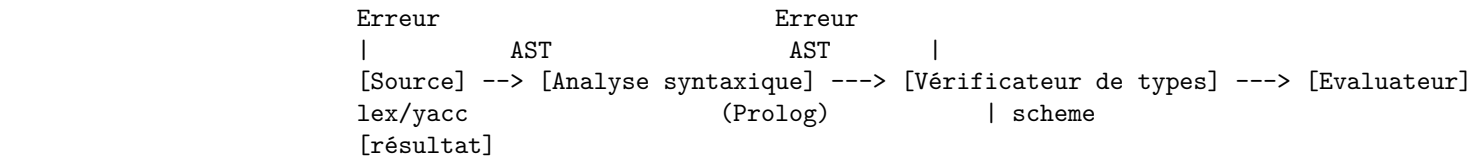
Comportement dynamique: résulte du traitement de données statiques (via CPU/interp bytecode/interp code source)

Règles/mécanismes d'interprétation = fonction d'interprétation

Analyse

Dirigée par la syntaxe.

- Analyse syntaxique: éliminer les sources non évaluables
- Analyse de type: éliminer les erreurs prévisibles à l'exécution.



Noyau impératif

- Expressions: bool, int, void
- Instructions: affectation, alternative, boucle, séquence, déclaration = Blocs

Syntaxe

Lexique Mots clés (réservés): Key | Bool | Int ——|——-|—— VAR | true | add CONST | false | sub SET | and | mul IF | or | div WHILE | |

Symboles réservés:

```
[ ] ( ) ;
```

Description du lexique des noms de variables et des entiers: expressions régulières.
En lex:

```
num   : [0-9][0-9]*
ident : [a-z][a-z]*
```

Grammaire Formalisation avec la Backus-Naur Form (BNF)

```
Prog ::= '[' Cmds ']'
Cmds ::= Cmd
       | Cmd ';' Cmds
Cmd  ::= Dec   #(declaration)
       | Stat  #(statement)
Dec  ::= 'CONST' ident Exp
       | 'VAR' ident TypeExp
```

```
TypeExp ::= 'bool' | 'int'
Stat ::= 'SET' ident Exp
       | 'IF' Exp Prog Prog
       | 'WHILE' Exp Prog
UnOp ::= '(' 'not' Exp ')'
BinOp ::= '(' 'and' Exp Exp ')'
        | '(' 'or' Exp Exp ')'
        | '(' 'add' Exp Exp ')'
        | '(' 'sub' Exp Exp ')'
        | '(' 'mul' Exp Exp ')'
        | '(' 'div' Exp Exp ')'
Exp  ::= 'true'
       | 'false'
       | num
       | ident
       | UnOp
       | BinOp
```

Cours 2

Typage

Définition: assigner un type (de données)

- aux expressions
- aux instructions
- aux programmes

Contexte de typage

Un ensemble d'assignations de types. *Noté* Γ.

$\Gamma \vdash x : t$ si $(x : t) \in \Gamma \; \forall$ type t

Dans le cadre du langage réalisé en APS:

Expressions

```
true  : bool
false : bool
n     : int  si n $\in$ num
x     : ?    si x $\in$ ident
(not e) : bool si e : bool
```

```
(and e1 e2) : bool si e1 : bool et e2 : bool
(or e1 e2) : bool si e1 : bool et e2 : bool

si e1 : int et e2 : int
(add e1 e2) : int
(sub e1 e2) : int
(mul e1 e2) : int
(div e1 e2) : int

si e1 : t et e2 : t pour tout type t
(eq e1 e2) : bool
(lt e1 e2) : bool
```

Instructions *Relation* entre:

- Un contexte de typage
- Une expression
- Un type

$\Gamma \vdash$ (SET x e) : void si $\Gamma \vdash$ x : t et $\Gamma \vdash$ e : t $\forall$ type t

$\Gamma \vdash$ (IF e b1 b2) : void si $\Gamma \vdash$ e : bool et $\Gamma \vdash$ b1 : void et $\Gamma \vdash$ b2 : void

$\Gamma \vdash$ (WHILE e b) : void si $\Gamma \vdash$ e : bool et b : void

Suite de commandes *Relation* entre:

- Un contexte de typage
- Un bloc (programme)
- void

Deux cas à considérer:

- [declaration; command sequence]
- [statement; command sequence]

$\Gamma \vdash$ [ ] : void

$\Gamma \vdash$ (s::cs) : void si s : void et $\Gamma \vdash$ cs : void $\forall$ s $\in$ statement

$\Gamma \vdash$ ((CONST x e)::cs) : void si Γ;x:t $\vdash$ cs : void et $\Gamma \vdash$ e : t

$\Gamma \vdash$ ((VAR x t)::cs) : void si Γ;x:t $\vdash$ cs : void

Notation ci-dessus: Γ;x:t abrège Γ U {x:t}

Programmes/blocs Bloc: suite de déclarations et d'instructions

$\vdash$ [cs] : void si $\vdash$ cs : void

Notation d'une règle d'inférence:

$$\frac{\Gamma \vdash e{:}bool \quad \Gamma \vdash b{:}void}{\Gamma \vdash (\text{WHILE e b}){:}void}$$

$$\frac{\Gamma, x{:}t \quad \vdash cs{:}void}{\Gamma \vdash ((\text{VAR x t})::cs{:}void}$$

$$\frac{x{:}bool \vdash true{:}bool \quad \frac{x{:}bool \vdash x{:}bool}{x{:}bool \vdash (not\, x){:}bool}}{\Gamma \vdash (\text{and true (not x)}){:}bool}$$

Simplification

On peut enlever des règles de typage en les ajoutant à la description du programme.

Par exemple:

true : bool; false : bool $\vdash$ [cs] : void

On peut ainsi enlever toutes les opérations dont les paramètres sont identiques à la valeur de retour.

$$\frac{\Gamma \vdash op{:}t_1 * t_2 -> t \quad \Gamma \vdash e_1{:}t_1 \quad \Gamma \vdash e_2{:}t_2}{\Gamma \vdash (\text{op } e_1, e_2){:}t}$$

En généralisant:

$$\frac{\Gamma \vdash op{:}t_1 * .. * t_n -> t \quad \Gamma \vdash e_1{:}t_1 \; .. \; \Gamma \vdash e_n{:}t_n}{\Gamma \vdash (\text{op} e_1, e_2, .. e_n){:}t}$$

Ajout des types dans la grammaire

```
type ::= 'bool'
      |  'int'
      |  type '*' .. '*' type -> type
      |  vartype
```

Cours 2

Sémantique opérationnelle

- *Expressions:* fonctionnel ~> valeurs
- *Instructions:* impératif ~> état mémoire
- *Déclaration:* ~> état mémoire, environnement

Environnement

Association entre identificateurs et * valeurs * adresses mémoire

Notation:

```
#v - valeurs
@a - adresses
```

Soit r un environnement.

- Accès, avec x : Id
  - r(x)
  - [ ](x) = error
- Ajout, avec w une valeur:
  - r[x = w]
  - r[x = w](x) = w
  - r[x = w](y) = r(y) avec x != y

Mémoire

Associations entre adresses et

- valeurs
- autres adresses

Avec m une mémoire:

- Accès:
  - m(a)
  - [ ](a) = error
- Extension (allocation):
  - m[a := new] – m(a) est non défini
- Modification: m[a := w]
  - m[a := w][a := w'] = m[a := w']
  - m[a' := w][a := w'] = m[a := w'][a' := w] avec a != a'
  - [ ][a := w] = error
- (m/r) (a) = m(a) si il existe x tq r(x) = @a
- error sinon

Jugement sémantique

- *Expression:* r, m ⊢ e ~> v
- *Instructions:* r, m ⊢ s ~> m'
- *Déclarations:* r, m ⊢ d ~> r', m'
- *Suite de commandes/Bloc:* r, m ⊢ [cs] ~> m'

Constantes

- r, m ⊢ true ~> #t
- r, m ⊢ false ~> #f
- r, m ⊢ n ~> #n (n : num)

Opérateurs booléens

- r, m ⊢ (not e) ~> #f si r, m ⊢ e ~> #t
- r, m ⊢ (not e) ~> #t si r, m ⊢ e ~> #f
- r, m ⊢ (or e1 e2) ~> #t si r, m ⊢ e1 ~> #t
- r, m ⊢ (or e1 e2) ~> #t si r, m ⊢ e1 ~> #f et r, m ⊢ e2 ~> #t
- r, m ⊢ (or e1 e2) ~> #f si r, m ⊢ e1 ~> #f et r, m ⊢ e2 ~> #f
- r, m ⊢ (and e1 e2) ~> #f si r, m ⊢ e1 ~> #f
- r, m ⊢ (or e1 e2) ~> #f si r, m ⊢ e1 ~> #t et r, m ⊢ e2 ~> #f
- r, m ⊢ (or e1 e2) ~> #t si r, m ⊢ e1 ~> #t et r, m ⊢ e2 ~> #t

Opérateurs arithmétiques

- r, m ⊢ (add e1 e2) ~> v1 + v2 si r, m ⊢ e1 ~> v1 et r, m ⊢ e2 ~> v2
- r, m ⊢ (sub e1 e2) ~> v1 - v2 si r, m ⊢ e1 ~> v1 et r, m ⊢ e2 ~> v2
- r, m ⊢ (mul e1 e2) ~> v1 * v2 si r, m ⊢ e1 ~> v1 et r, m ⊢ e2 ~> v2
- r, m ⊢ (div e1 e2) ~> v1 / v2 si r, m ⊢ e1 ~> v1 et r, m ⊢ e2 ~> v2 et v2 != 0

Instructions

- r, m ⊢ (SET x e) ~> m[a := v] si r, m ⊢ e ~> v et r(x) = @a
- r, m ⊢ (IF b e1 e2) ~> m' si r, m ⊢ b ~> #t et r, m ⊢ e1 ~> m'
- r, m ⊢ (IF b e1 e2) ~> m' si r, m ⊢ b ~> #f et r, m ⊢ e2 ~> m'
- r, m ⊢ (WHILE e b) ~> m si r, m ⊢ e ~> #f
- r, m ⊢ (WHILE e b) ~> m'‘si r, m ⊢ e ~> #t et r, m ⊢ b ~> m' et r, m' ⊢ (WHILE e b) ~> m"

Suite de commandes

- Si s une instruction: r, m ⊢ s;cs ~> r, m' ⊢ s ~> m'
- Si d ≡ CONST x e: r, m ⊢ d; cs ~> r[x := #v], m si r, m ⊢ e ~> #v
- Si d ≡ VAR x: r, m ⊢ d; cs ~> r[x := @a], m[a = new]

Bloc

- r, m ⊢ [cs] ~> r, (m'/r) si r, m ⊢ cs ~> r', m'

Typage

Distinguer les identificateurs modifiables des immuables.

Introduction du type **"pointeur vers"**: **@t** "pointeur vers t"

- Déclaration/suite de commandes
  - G ⊢ (VAR c t; cs) : void si G, x:@t ⊢ cs : void
- Expression
  - G, x:@t ⊢ x:t || G ⊢x:t si G(x) = t ou G(@x) = t
- Affectation
  - G ⊢ (SET x e) : void si G ⊢ x:@t et G ⊢ e:t

Sémantique

- r, m ⊢ (SET x e) ~> m[r(x) := v] si r, m ⊢ e ~> v

Cours 4

Retour sur la portée lexicale

Blocs Sémantique: si r, m ⊢ cs ~> m' alors r, m ⊢ [cs] ~> **m'/r**

```
[
    VAR x int;
    SET x 0;
    IF (true)
        [ VAR x int;
          SET x 12 ]
        [ SET x 1  ]
    SET x (x + 1)
]
```

À la fin de l'exécution du programme, x vaut 1. Si on retire /r de la règle de sémantique du bloc, on peut avoir le programme suivant où x vaut 4:

```
[
    VAR x int;
    IF (true)
        [ SET x 3 ]
        [   ...   ]
    SET x (x + 1)
]
```

Fonctions et procédures

- Fonctions: abstractions vis-à-vis d'une expression
- Procédures: abstractions vis-à-vis d'un bloc d'expressions

Fonction

- Domaine de départ, codomaine d'arrivée.
- Abstraction d'une expression avec ses variables associées
- En APS: [x:int] (x + 1) : **expression fonctionnelle**.

Syntaxe

```
Expr ::= ..
       | '[' TypeIds ']' Expr
       | '(' Exprs ')'

Exprs ::=  Expr
        | Expr Exprs

TypeIds ::= ident ':' type
         | ident ':' type ';' TypeIds
```

Exemple

```
CONST xor [b1:bool;b2:bool]
            (and (or b1 b2)
                (not (and b1 b2))); ..
(xor true false)
```

Typage

- **Si** G;x_1 : t_1;x_2 : t_2;..;x_n : t_n ⊢ e:t **alors** G ⊢ [x_1 : t_1;..;x_n : t_n]e : (t_1 ∗ .. ∗ t_n)->t
- **Si** G ⊢ e:(t1*..*tn)->t et G ⊢ e1:t1 .. G ⊢ en:tn **alors** G ⊢ (e e1 .. en):t

```
Type ::= ..
       | Typeargs '->' Type
       | '(' Type ')'

Typeargs ::= Type
           | Type '*' Typeargs
```

Sémantique

Ajouter les définitions des expressions fonctionnelles, des valeurs fonctionnelles (=> fermetures).

(([x_1 : t_1;..;x_n : t_n]e') e_1 .. e_n)

Fermeture: (notation) **< e, r +** [x_1=#]**..**[x_n**:**#] **>**

Sémantique de la fermeture: r, m ⊢ [x_1 : t_1;..;x_n : t_n]e ~> < e, r + [x_1=#]..[x_n:#] >

Application - Liaison statique

- **Si** r, m ⊢ e ~> < e', r" + [x_1=#]..[x_n=#] >
- **et** r, m ⊢ e_1 ~> v_1 .. r, m ⊢ e_n ~> v_n
- **et** r'[x_1 = v_1]..[x_n = v_n], m ⊢ e' ~> v
- **alors** r, m ⊢ (e e_1..e_n) ~> v

```
[ CONST f [x:int] (add x 1)
  CONST g [x:int] (add (f x) 1)
  CONST f [x:int] (add x 3)
  VAR r int
  SET r (g 42)
]
```

Liaison statique: la valeur de g est fabriquée au moment de sa déclaration, g capture la valeur de f à l'instant de la déclaration de g. La valeur de r est donc 44.

Procédures

Avec d = PROC p $[x_1 : t_1; ..; x_n : t_n]$ blk

Syntaxe

```
Dec ::= ..
      | 'PROC' ident '[' TypeIds ']' Prog

Stat ::= ..
       | 'CALL' ident Exprs
```

Typage **Si** G; p:$t_1 * .. * t_n$->void $\vdash$ cs : void et G $\vdash$ blk : void **alors** G $\vdash$ d;cs : void

Appel **Si** G $\vdash$ p:$t_1 * .. * t_n$ -> void **et** G $\vdash$ $e_1 : t_1$ .. G $\vdash$ $e_n : t_n$ **alors** G $\vdash$ CALL p $e_1..e_n$: void

Sémantique

Fermeture "procédurale": **< blk, r + [x_1=#]..[x_n=#] >**

Si r [p = < blk, r + [x_1=#]..[x_n=#] >], m $\vdash$ cs ~> (r', m') **alors** r, m $\vdash$ d; cs ~> (r', m')

Application - Liaison statique

- **Si** r(p) = < blk, r + [x_1=#]..[x_n=#] >
- **et** r, m $\vdash$ e_1 ~> v_1 .. r, m $\vdash$ e_n ~> v_n
- **et** r'[$x_1 = v_1..x_n = v_n$], m $\vdash$ blk ~> m'
- **alors** r, m $\vdash$ (CALL $pe_1..e_n$) ~> m'

Cours 5

L'instruction RETURN

```
stat ::= ..
       | RETURN Expr
```

Typage

- Expression:
  - *Si* G ⊢ e:t *alors* G ⊢ (RETURN e):t
- Séquences: soit s != (RETURN e)
  - *Si* G ⊢ void *et* G ⊢ cs:t *alors* G ⊢ (s;cs):t
  - *Si* G ⊢ s:t avec t != void *et* G ⊢ cs:t *alors* G ⊢ (s; cs):t
- *Si* G ⊢ e:t *et* G ⊢ x:t *alors* G ⊢ (SET x e):void
- *Si* G ⊢ e:bool *et* G ⊢ cs:t *alors* G ⊢ (WHILE e cs):t
- *Si* G ⊢ e:bool *et* G ⊢ cs1:t *et* G ⊢ cs2:t *alors* G ⊢ (IF e cs1 cs2):t

Sémantique

Instruction:

- une non valeur
- une valeur v

On note ω pour v ou $\emptyset$.

Relation sémantique des instructions:

- Env, Mem, Stat -> (Val, Mem)
- r, m ⊢ s ~> (ω, m')

RETURN Si r, m ⊢ e ~> v alors r, m ⊢ (RETURN e) ~> (v, m)

Affectation Si r, m ⊢ (SET x e) ~> ($\emptyset$, m[r(x):=v])

Boucle

- Si r, m ⊢ e ~> #f alors r, m ⊢ (WHILE e blk) ~> ($\emptyset$, m)
- Si r, m ⊢ e ~> #t alors
  - Si r, m ⊢ blk ~> ($\emptyset$, m') et r, m' ⊢ (WHILE e blk) ~> (ω, m“) alors r, m ⊢ (WHILE e blk) ~> (ω, m")
  - Si r, m ⊢ blk ~> (v, m') alors r, m ~> (WHILE e blk) ~> (v, m')

Alternative

- Si r, m ⊢ e ~> #t et r, m ⊢ blk1 ~> (ω, m') alors r, m ⊢ (IF e blk1 blk2) ~> (ω, m')
- Si r, m ⊢ e ~> #f et r, m ⊢ blk2 ~> (ω, m') alors r, m ⊢ (IF e blk1 blk2) ~> (ω, m')

Suites de commandes Pour toute instruction s et toute continuation cs: * Si r, m ⊢ s ~> ($\emptyset$, m') et r, m' ⊢ cs ~> (ω, m") alors r, m ⊢ (s;cs) ~> (ω, m") * Si r, m ⊢ s ~> (v, m') alors r, m ⊢ (s;cs) ~> (v, m')

Déclarations

- Si d ≡ CONST x e
  - Si r, m ⊢ e ~> v et r[x=v], m ⊢ cs ~> (ω, m') alors r, m ⊢ (d;cs) ~> (ω, m')
- Si d ≡ VAR x t
  - Si r[x=a], m[x=new] ⊢ cs ~> (ω, m') alors r, m ⊢ (d;cs) ~> (ω, m')

Fonctions procédurales

Syntaxe

```
Dec ::= ..
      | 'FUN' ident Type '[' TypeIds ']' Prog
      | 'FUN' ident Type '['  ']' Prog
```

Typage

- Expression/application:
  - Si G ⊢ f:void->t alors G ⊢ (f):t
- Déclaration/suite de commandes:
  - d ≡ FUN f t [ ] blk
    - Si G[f:void->t] ⊢ cs:t' et G ⊢ blk:t alors G ⊢ (d;cs):t'
  - d ≡ FUN f t [$x_1 : t_1; ..x_n : t_n$] blk
    - Si G[f:$t_1 * .. * t_n$] ⊢ cs:t' et G[$x_1 : t_1; ..; x_n : t_n$] ⊢ blk:t alors G ⊢ (d;cs):t'

Sémantique

Relation sémantique des expressions: r, m ⊢ e ~> (v, m')

Valeur d'une fonction procédurale: < bd, r+[x1=#] >

Application

- Si
  - r, m ⊢ e ~> (< r'+[x1=#..xn=#]>, m')
  - r, m' ⊢ e1 ~> (v1, m1) et .. et r, mn-1 ⊢ en ~> (vn, mn)
  - r'[x1=v1]..[xn=vn], mn ⊢ bd ~> m"
- alors r, m ⊢ (e e1 .. en) ~> (v, m“)

Fonctions récursives

Syntaxe

```
Dec ::= ..
      | 'FUN' 'REC' ident Type '[' TypeIds ']' Prog
      | 'FUN' 'REC' ident Type '[' ']' Prog
```

Sémantique

Valeur d'une fonction récursive: < bd, r+[f=!][x_1=#]..[x_n=#] >

Application

- Si
  - r, m ⊢ e ~> (< r'+[f=!][x_1=#]..[x_n=#]>, m')
  - r, m' ⊢ e_1 ~> (v_1, m_1) et .. et r, $m_n - 1$ ⊢ en ~> (v_n, m_n)
  - r'[f=< bd, r+[f=!][x1=#]..[xn=#] >][$x_1 = v_1$]..[$x_n = v_n$], m_n ⊢ bd ~> m"
- alors r, m ⊢ (e e_1 .. e_n) ~> (v, m“)