

Rendu Final de Projet APS

Haotian XUE
Hejun CAO

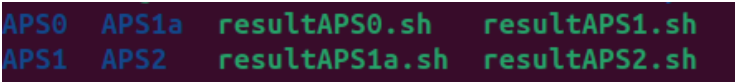
April 24, 2024

1 Introduction

Dans ce projet, nous devons maîtriser les connaissances nécessaires pour créer un nouveau langage, construire des arbres syntaxiques, effectuer la vérification des types ainsi que la vérification sémantique. Le projet est divisé en quatre versions : APS0, APS1, APS1a et APS2. Chaque nouvelle version vise à enrichir les fonctionnalités de la version précédente.

2 Installation

Afin d'exécuter la détection du projet de langage APS, nous devons exécuter le langage de script, saisir `chmod +x ./result*` dans le terminal Linux et exécuter l'entrée `./resultAPS{number}.sh`.



```
APS0  APS1a  resultAPS0.sh  resultAPS1.sh
APS1  APS2   resultAPS1a.sh  resultAPS2.sh
```

Figure 1: Le dossier principale

3 Progression

APS0

APS0 comprend les opérations de base et les définitions de la langue APS, y compris les opérations arithmétiques telles que l'addition, la soustraction, la multiplication, la division, ainsi que les opérations sur les valeurs booléennes.

Syntaxe :

ast.ml : ok , parser.mly : ok , lexer.mll : ok

Typage :

L'analyse commence à partir du type de prog vers l'intérieur et revient finalement à prog. Si le type de retour est nul, la vérification est réussie.

(PROG)(DEFS)(END)(CONST)(FUN)(FUNREC)(ECHO)(NUM)(ID)(IF)(AND)(OR)(APP)(ABS):ok

Sémantique :

Utilisez 42 comme résultat de la vérification. Si l'ECHO final est 42, la vérification est réussie.

(PROG)(DEFS)(END)(CONST)(FUN)(FUNREC)(ECHO)(NUM)(ID)(IF)(AND)(OR)(APP)(ABS):ok

APS1

APS1 ajoute deux nouveaux conteneurs, Mémoire et Flux, basés sur l'environnement. Mémoire est utilisé pour stocker les adresses, tandis que Flux est utilisé pour mettre à jour et afficher les résultats.

Syntaxe :

ast.ml : ok , parser.mly : ok , lexer.mll : ok

Typage :

(VAR)(PROC)(PROCREC)(SET)(WHILE)(CALL): ok

Sémantique :

(VAR)(PROC)(PROCREC)(SET)(WHILE)(CALL): ok

APS1a

L'extension APS1a à APS1 a été motivée par deux problèmes liés à la gestion des types dans APS1 : premièrement, certains programmes syntaxiquement corrects et bien typés peuvent tout de même causer des erreurs à l'exécution ; deuxièmement, le système de typage dans APS1 ne permet pas la définition de procédures qui modifient leurs arguments

Syntaxe :

ast.ml : ok , parser.mly : ok , lexer.mll : ok

Typage :

(VAR) (SET) (REF) (VAL) (IDV) (IDR) : ok

Sémantique :

(VAR) (SET) (REF) (VAL) (IDV) (IDR) : ok

APS2

APS2 continue de se développer sur la base d'APS1a, en ajoutant les opérations len, nth, alloc, vset et autres du tableau ainsi que les types vec.

Syntaxe :

ast.ml : ok , parser.mly : ok , lexer.mll : ok

Typage :

(ALLOC) (NTH) (LEN) (VSET) : ok

Sémantique :

(ALLOC) (NTH) (LEN) (VSET) : ok

4 Implémentation

```
rule token = parse
| [' ' '\t' '\r' '\n'] { token lexbuf } (* skip blanks *)
| '[' { LBRA }
| ']' { RBRA }
| '(' { LPAR }
| ')' { RPAR }
| ';' { SEMICOLON }
| ':' { COLON }
| ',' { COMMA }
| '*' { MUL }
| "->" { ARROW }
| "if" { IF }
| "and" { AND }
| "or" { OR }
| "int" { TYPEINT }
| "bool" { TYPEBOOL }
| "ECHO" { ECHO }
| "CONST" { CONST }
| "FUN" { FUN }
| "REC" { REC } (* "FUN REC" *)
(* TMS2 *)
```

- Définir l'identifiant du jeton du symbole de base.

```

and expr =
| ASTNum of int
| ASTId of string
| ASTApp of expr * expr list
| ASTIf of expr * expr * expr
| ASTAnd of expr * expr
| ASTOr of expr * expr
| ASTBra of args * expr

```

(a) ast.ml

```

expr:
  NUM { ASTNum($1) }
| IDENT { ASTId($1) }
| LPAR IF expr expr RPAREN { ASTIf($3, $4, $5) }
| LPAR AND expr expr RPAREN { ASTAnd($3, $4) }
| LPAR OR expr expr RPAREN { ASTOr($3, $4) }
| LPAR expr RPAREN { ASTApp($2, $3) }
| LBRA args RBRA { ASTBra($2, $4) }
;

```

(b) parser.mly

- Prétraitez le type d'arborescence syntaxique et AST défini dans ast.ml avec le type de jeton défini par parser.mly. Après préparation, utilisez `prologTerm` pour imprimer la déclaration.

```

expr->string
and print_expr e =
  match e with
  | ASTNum n -> "num(" ^ string_of_int n ^ ")"
  | ASTId x -> "ident(" ^ x ^ ")"
  | ASTIf(c,s,a) -> "if(" ^ print_expr c ^ "," ^ print_expr s ^ "," ^ print_expr a ^ ")"
  | ASTAnd(x1,x2) -> "and(" ^ print_expr x1 ^ "," ^ print_expr x2 ^ ")"
  | ASTOr(x1,x2) -> "or(" ^ print_expr x1 ^ "," ^ print_expr x2 ^ ")"
  | ASTApp(e, es) -> "app(" ^ print_expr e ^ "," ^ print_exprs es ^ ")"
  | ASTBra(a,e) -> "[" ^ print_args a ^ "]" ^ print_expr e

```

- Réorganisez et imprimez tout le contenu avec une sémantique définie dans le terminal.

```

g0(
  [(true,bool),
   (false,bool),
   (not,arrow([bool],bool)),
   (add,arrow([int,int],int)),
   (lt,arrow([int,int],bool)),
   (sub,arrow([int,int],int)),
   (mul,arrow([int,int],int)),
   (div,arrow([int,int],int))
  ]
).

```

```

env
let env0 : env = [
  ("true", InZ(1));
  ("false", InZ(0));
  ("add", Operator(Add));
  ("sub", Operator(Sub));
  ("eq", Operator(Eq));
  ("lt", Operator(Lt));
  ("mul", Operator(Mul));
  ("div", Operator(Div));
  ("not", Operator(Not));
]

```

- Initialiser l'environnement `env0`

```

and sem_op_unary op v =
  match op with
  | Operator(Not) -> if v = InZ(1) then InZ(0)
  else InZ(1)
  | _ -> failwith "Not defined this unary operation"

value -> int -> int -> value
and sem_op_binary op v1 v2 =
  match op with
  | Operator(Add) -> InZ(v1 + v2)
  | Operator(Sub) -> InZ(v1 - v2)
  | Operator(Mul) -> InZ(v1 * v2)
  | Operator(Div) -> InZ(v1 / v2)
  | Operator(Eq) -> if v1 = v2 then InZ(1) else InZ(0)
  | Operator(Lt) -> if v1 < v2 then InZ(1) else InZ(0)
  | _ -> failwith "Not defined this binary operation"

```

- Opérateurs unaires et binaires séparés

```

| Operator _ ->
  match List.length es with
  | 1 -> sem_op_unary e (List.hd es)
  | 2 -> sem_op_binary e (sem_InZ (List.hd es)) (sem_InZ (List.hd (List.tl es)))
  | _ -> failwith "neither unary nor binary"

```

- Rematch des opérateurs séparés

```

% "type_cmds(Z,[const(x,int,num{1}),const(y,int,num{2}),echo(false)],void)."
% "type_cmds(Z,[const(x,bool,false),const(y,int,num{2}),echo(false)],void)."
type_cmds(Env,[Def|Cs],void):-
  writeln("Def Starts"),
  writeln(Env),
  type_def(Env,Def,Env_New),
  writeln(Env_New),
  writeln("Def Ends, suit Starts"),
  type_cmds(Env_New,Cs,void),
  writeln("cmds Ends").

```

- Lorsque nous rencontrons une erreur introuvable, nous utilisons la méthode d'impression pour déterminer l'emplacement de l'erreur et la déboguer manuellement.

```

Syntaxe > ML > $ resultAPSO.sh
1 for file in ./APSO/MonSample/*.aps
2 do
3   echo "*****"
4   echo $file " "
5   type=$(./APSO/prologTerm $file | swipl -s ./APSO/prog.pl -g main 2>&1)
6   if [[ $type = "void" ]]; then
7     echo "Type Checking OK."
8   else
9     echo "Type Checking error!"
10  fi
11  res=$(./APSO/eval $file)
12  if [[ $res = "42" ]]; then
13    echo "Eval Checking OK."
14  else
15    echo "Eval Checking error! Result: $res "
16  fi

```

```

./APSO1/MonSample/set_2.aps :
Type Checking OK.
Eval Checking OK.
*****
./APSO1/MonSample/test_cours.aps :
Type Checking OK.
Eval Checking OK.
*****
./APSO1/MonSample/while.aps :
Type Checking OK.
Eval Checking OK.

```

- Nous avons mis en pratique un script qui inclut la ligne de commande pour détecter le type et l'évaluation, qui est utilisée pour tester tous les fichiers .aps de MonSample et afficher les résultats sur le terminal.
- Des fonctions pour implémenter la mémoire:
 1. get_value (id: string) (env:env) = ... : Trouver la valeur correspondant à l'identifiant dans l'environnement.
 2. mis_a_jour id_list value_list env = ... : Mettre à jour une liste (id, valeur) dans l'environnement.
 3. getInMem mem adr = ... : Interroger la valeur de l'adresse en mémoire.
 4. newAddress mem = ... : Calculer une nouvelle adresse.
 5. allocMemory mem = ... : Développez une nouvelle mémoire.
 6. replaceElem mem oldElem newElem = ... : Remplacer la valeur existante.
 7. editInMem mem id adr = ... : Remplacer le contenu dans Mem.

```

,procrec(loop,[(f,arrow([int],int)),[(n,int)]]
over(ident(f)) over(app(ident(sub), fident(n))

```

```

,procrec(loop,[(f,arrow([int],int)),(n,int)],b]
over(ident(f)) over(app(ident(sub), fident(n))

```

```

and print_argp argp =
  match argp with
  | ASTVarp0(ident,typ) -> "(" ^ ident ^ "," ^ print_typ typ ^ ")"
  | ASTVarp(ident,typ) -> "(" ^ ident ^ "," ^ "ref(" ^ print_typ typ ^ ")"

argsp->string
and print_argsp argsp =
  match argsp with
  | ASTArgp(arg) -> "[" ^ print_argp arg ^ "]"
  | ASTArgsp(arg,args) -> "[" ^ print_argp arg ^ "," ^ print_argsp_tails args ^ "]"

argsp->string
and print_argsp_tails argsp = |
  match argsp with
  | ASTArgp a -> print_argp a
  | ASTArgsp(a,ars) -> print_argp a ^ "," ^ print_argsp_tails ars

```

- Nous avons rencontré un problème : [] apparaissait de manière répétée lors des appels récursifs, nous avons donc créé une fonction tail pour éviter ce problème. Faites apparaître [] uniquement au début et à la fin des arguments.

5 Conclusion

Dans ce projet, nous avons débuté avec les opérations de base et le système de types dans APS0, amélioré la gestion de la mémoire et des flux dans APS1 et APS1a, et finalement supporté des structures de données complexes dans APS2. Chaque version a montré la profondeur et la complexité de la conception des langages de programmation. Ce processus nous a enseigné des techniques clés, allant de la construction des structures de base du langage à l'implémentation de systèmes de types avancés, et à la gestion des erreurs par des tests et du débogage. Le projet a non seulement amélioré nos compétences techniques, mais a aussi renforcé notre capacité à travailler en équipe et à résoudre des problèmes, préparant le terrain pour les défis futurs.