Algorithmes, Types, Preuves: Présentation du projet

Martin Strecker

Univ. J.-F. Champollion

Année 2021/2022

Plan

Plan

Survol du projet

Le développement à faire :

- Formalisation de la syntaxe d'un langage impératif en Caml ($\rightsquigarrow$ Langages et automates)
- Vérification de types de ce langage
- Conception d'un assembleur simple
- ... avec sémantique de cet assembleur
- Traduction langage impératif vers assembleur
- Génération et affichage d'un graphe de flot de contrôle

Survol du projet

Motivation et but d'apprentissage :

- Savoir transférer les concepts vus pour un langage fonctionnel vers langage impératif (surtout : typage)
- Développer une intuition pour différents niveaux de langage (de haut niveau, assembleur)
- S'initier à des concepts de compilation
- Transférer et appliquer des concepts de théorie des graphes

Plan

Inspiration

Langage inspiré du langage C, mais :

- Moins sauvage :
  - Distinction nette entre expressions et instructions.
    Par exemple : impossible d'écrire `(x = 3) + (x = 2);`
  - Permet la définition d'une sémantique précise et claire
- Pourtant compatible avec C : Le même programme peut être
  - compilé avec `gcc` et exécuté
  - compilé avec *votre* compilateur et exécuté

Exemple : Programme source

```
int fac (int n) {
  int res;
  res = 1;

  while (n > 1) {
    res = res * n;
    n = n - 1;
  }
  return(res) ;
}
```

Nous manipulons :

- des fonctions simples
- avec paramètres et variables locales
- contenant des instructions impératives
- sans appel d'autres fonctions
- qui renvoient une valeur (`return`)

Expressions : Type de données

Sémantique informelle : Une expression a une *valeur*

```
type 'a expr =
    Const of 'a * value   (* constante *)
  | VarE of 'a * vname      (* variable *)
  | BinOp of 'a * binop * ('a expr) * ('a expr)
                          (* operation binaire *)
  | IfThenElse of 'a *
        ('a expr) * ('a expr) * ('a expr)
                          (* if - then - else *)
```

Note : `IfThenElse` est une expression analogue à `(.. ? .. : ..)` en C

Instructions : Type de données

Sémantique informelle : Une instruction (*statement*) induit un *changement d'état*

```
type 'a stmt =
    Skip                                    (* ne fait rien *)
  | Assign of 'a * vname * ('a expr)  (* affectation *)
  | Seq of ('a stmt) * ('a stmt)  (* sequence *)}
  | Cond of ('a expr) * ('a stmt) * ('a stmt)
                                  (* if .. then .. else .. *)
  | While of ('a expr) * ('a stmt)
  | Return of ('a expr)   (* Renvoi d'une valeur *)
```

Variables et valeurs

Variables et fonctions :

```
(* variable names *)
type vname = string

(* function names *)
type fname = string
```

Valeurs :

```
type value =
    BoolV of bool
  | IntV of int
```

Typage : bases

Juste deux types :

```
type tp = BoolT | IntT
```

Déclaration de variable :

```
type vardecl =  Vardecl of tp * vname
```

Fonctions :

```
(* function declaration:
   return type; parameter declarations *)
type fundecl =
    Fundecl of tp * fname * (vardecl list)

(* function definition:
   function decl; local var decls; function body *)
type 'a fundefn =
    Fundefn of fundecl * (vardecl list) * ('a stmt)
```

Exemple

```
int fac (int n) {
  int res;
  res = 1;

  while (n > 1) {
    res = res * n;
    n = n - 1;
  }
  return(res) ;
}
```

```
# parse "Tests/fac.c" ;;
- : int Lang.fundefn =
Fundefn (
  Fundecl(IntT, "fac",
          [Vardecl (IntT, "n")]),
 [Vardecl (IntT, "res")],
 Seq
  (Seq (Assign (0, "res",
                Const (0, IntV 1)),
        ....),
  Return (VarE (0, "res"))))
```

Typage : Environnement

Le typage doit

- prendre en compte les déclarations des variables
- le type de résultat de la fonction

```
type environment =
    {localvar: (vname * tp) list;
     returntp: tp}
```

Typage

Le typage annote l'arbre syntaxique (expressions) avec le type des sous-expressions.
Résultat du parser :

```
Seq (Assign (0, "res", Const (0, IntV 1)),
  While (BinOp (0, BCompar BCgt,
          VarE (0, "n"), Const (0, IntV 1)), ...))
```

Après typage :

```
Seq (Assign (IntT, "res", Const (IntT, IntV 1)),
  While (BinOp (BoolT, BCompar BCgt,
          VarE (IntT, "n"), Const (IntT, IntV 1)),...))
```

A faire :

- Écrire typage d'expressions, instructions et fonctions
- Analyses de bone formation :
  noms des variables disjoints ; fonctions terminent avec `return`

Plan

Langage cible : Instructions

```
(* 'l for labels, 'v for variables *)
type ('l, 'v) instr =
  (* store result of simple_expr in variable *)
| Store of 'v * 'v simple_expr
  (* unconditional goto *)
| Goto of 'l
  (* if zero then branch to left, else to right *)
| Branch of 'v * 'l * 'l
  (*return from function leaving result in variable*)
| Exit of 'v
```

Langage cible : Expressions simples

```
type 'v simple_expr =
   (* Constant *)
   ConstSE of int
   (* Variable *)
| VarSE of 'v
   (* Application of binary operation *)
| BinOpSE of binop * 'v * 'v
```

Langage cible : Expressions simples

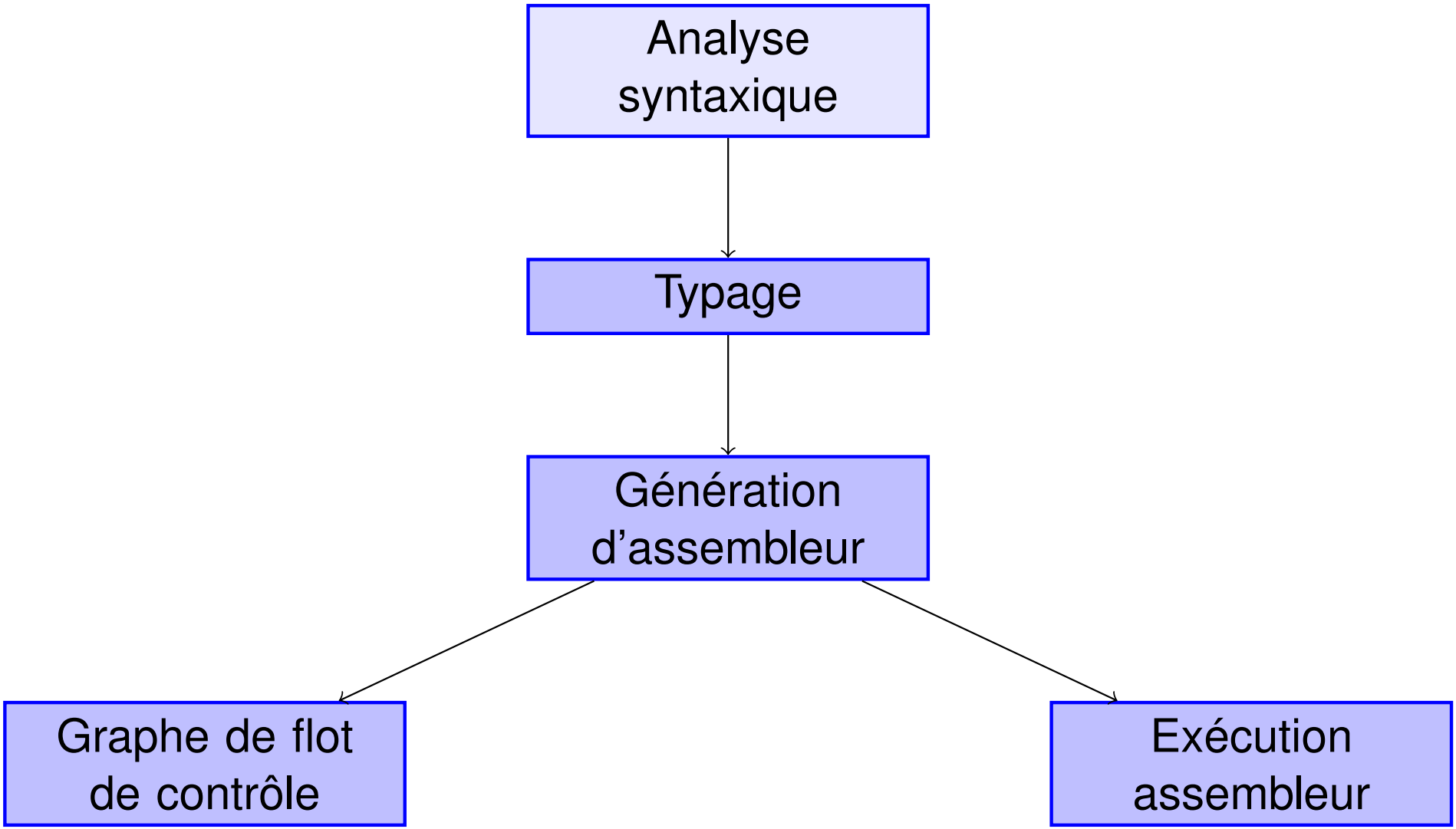
Petit extrait d'un programme :

```
[Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNum 2, ConstSE 0);
 Store (IVarNum 0, BinOpSE (BCompar BClt,
                            IVarNum 1, IVarNum 2));
 Branch (IVarNum 0, RelJump 2, RelJump 1);
 Goto (RelJump 5); ...]
```

Son type : `(rel_jump, instr_var) instr list`

Plan

Les étapes principales



Typage

- Presque comme d'habitude pour les instructions.
  Différence essentielle : Annotation des expressions avec un type.
- Très facile pour les instructions.
  Essentiellement : Vérification des sous-expressions
- Nouveau : Vérification des fonctions.
  Essentiellement : Vérification du résultat renvoyé.

Environnement étendu :

```
type environment =
    {localvar: (vname * tp) list;    (* as usual *)
     returntp: tp}                   (* new *)
```

Typage Expressions

- Au lieu de renvoyer un type :
- On annote les expressions avec un type

Code source : `n < 0`

Expression non typée (construite par le parser) :

```
BinOp (0, BCompar BClt,
      VarE (0, "n"),
      Const (0, IntV 0))
: int expr
```

Expression typée (annotation par la vérification de type) :

```
BinOp (BoolT, BCompar BClt,
      VarE (IntT, "n"),
      Const (IntT, IntV 0))
: tp expr
```

Typage Instructions

Passage de `int stmt` à `tp stmt`

- Vérification affectation :
  `b = 0` est incorrecte si `b : bool`
- Vérification des conditions de `if ... else` et `while`

Typage Fonctions

Passage de `int fundefn` à `tp fundefn`

Enjeu : la valeur renvoyée par un `return` doit correspondre à la déclaration de la fonction.

Exemple de typage incorrect :

```
bool f (int n) {
  return (n + 1) }
```

Démarche :

- Construction d'un environnement initial :
  - liste des paramètres et leur type
  - type de résultat de la fonction
- Vérification du corps de la fonction avec cet environnement

Génération d'assembleur

Entrée : expression / instruction / def. de fonction typée

Sortie : code (liste d'instructions) assembleur

Étapes :

1. Code avec adresses relatives, variables nommées
2. Code avec adresses absolues, variables nommées
3. Code avec adresses absolues, variables indexées

Code avec adresses relatives, variables nommées

- *Adresses relatives :* `rel_jump`
- *variables nommées :* `instr_var`

But des adresses relatives :

- Branchement vers des cibles à *n* instructions de l'instruction actuelle
- Utile si on connaît la longueur de code sur lequel il faut sauter.

Code avec adresses relatives, variables nommées

Exemple : Génération de code pour

```
while (n > 1) {
  n = n - 2;
}
...
```

Cond :	Store(V1, n)
	Store(V2, 1)
	Store(V0, V1 > V2)
	Branch(V0, 1, 5)
Instr :	Store(V0, n)
	Store(V1, 1)
	Store(n, V0 - V1)
	Goto (-7)
Reste :	...

- Dans : `Branch(V0, 1, 5)` :
  5 = longeur(Instr) + longeur(Goto) + 1
- Dans : `Goto(-7)` :
  7 = longueur(Cond) + longeur(Instr) + longeur(Branch)

Code avec adresses absolues, variables nommées

Les adresses relatives sont remplacées par les adresses absolues :

8		...
9	Cond :	Store(V1, n)
10		Store(V2, 1)
11		Store(V0, V1 > V2)
12		Branch(V0, 13, 17)
13	Instr :	Store(V0, n)
14		Store(V1, 1)
15		Store(n, V0 - V1)
16		Goto (9)
17	Reste :	...

Code avec adresses absolues, variables indexées

Idée : Toutes les variables

- seront stockées dans un tableau
- avec accès par *indice*

Exemple pour un programme contenant les variables nommées `i` et `n` et les variables temporaires `V0, V1, V2`.

Avant transformation :

variable	i	n	V0	V1	V2
valeur	2	5	42	17	9

- `IVarNamed "n"`
- `IVarNum 1`

Après transformation :

variable	0	1	2	3	4
valeur	2	5	42	17	9

- `IIndex 1`
- `IIndex 3`

Exécution assembleur

Idée :

- Machine virtuelle capable d'exécuter le code compilé
- Plus facile que l'exécution de code structuré

Démarche : Transformation d'une configuration composée de :

– Code (reste inchangé)

0	1	..	5	..	9
`Store(x, y + z)`	`Goto(5)`	..	`Branch(y, 0, 9)`	..	`Exit(x)`

– Tableau de variables (mise à jour avec `Store`)

x	y	z
30	40	2

– Compteur de programmes `pc`

pc : 0

Exécution assembleur : Sémantique des instructions

Chaque instruction fait transiter la configuration actuelle vers une nouvelle configuration.

- Instruction `Store` : modifie variables et `pc`
  Exemple : `Store(x, y + z)`
  - Lecture des variables : `y, z`
  - Exécution de l'opération : 40 + 2
  - Mise à jour de variable dans le tableau : `x := 42`
  - Incrément du pc : `pc := pc + 1`

x	y	z
30	40	2

$\rightsquigarrow$

x	y	z
42	40	2

Note : A ce point, les variables ne sont plus nommées, mais indexées.

Exécution assembleur : Sémantique des instructions

- Instruction `Goto` : modifie `pc` uniquement
  Exemple : `Goto(5)`
  pc : 1 $\rightsquigarrow$ 5
- Instruction `Branch` : modifie `pc` uniquement
  Exemple : `Branch(y, 0, 9)`
  - si valeur de `y` $\neq 0$:
    pc : $\rightsquigarrow$ 9
  - si valeur de `y` $= 0$:
    pc : $\rightsquigarrow$ 0

Exécution assembleur : Sémantique des instructions

- Instruction `Exit(x)` :
  - pas de nouvelle configuration
  - mais : arrêt du programme, renvoi de la valeur associée à la variable `x`

Mise en oeuvre :

```
type exec_result =
   (* Fin du programme, renvoie valeur entiere*)
   End of int
   (* Nouvelle config avec tableau des variables et pc
| Continue of (int array) * int
```

A écrire :

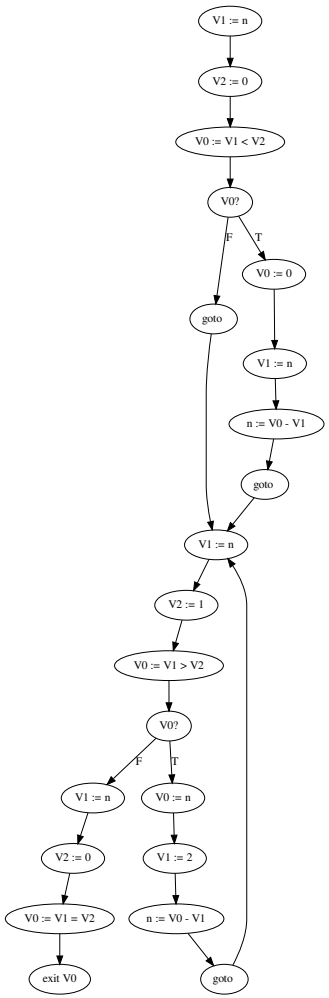
```
exec_instr : (abs_jump, instr_index) instr array ->
    int array -> int -> exec_result
```

Plan

Motivation

Graphe de flot de contrôle :

- Généré à partir du code assembleur
- Traduction vers un type générique de graphes
- Affiché avec Graphviz



Type des graphes

Définition mathématique : Graphe $G = (V, E)$ où

- V est un ensemble de noeuds
- $E \subseteq V \times V$ est un ensemble d'arcs

Ici : graphes étiquetés avec *label* de type `'l`

```
type ('n, 'l) node = Node of 'n * 'l
type ('n, 'l) edge = Edge of 'n * 'n * 'l
type ('n, 'nl, 'el) graph =
   Graph of ('n, 'nl) node list * ('n, 'el) edge list
```

Graphe avec fonctions d'affichage des noeuds / arcs :

```
type ('n, 'nl, 'el) display =
  Display of ('n, 'nl, 'el) graph *
             ('n -> string) *
             ('nl -> string) *
             ('el -> string)
```

Traduction code vers graphes

Création du graphe pour une liste d'instructions :

- Noeuds : les instructions de la liste
- Arcs :
  - `Store` : un arc entre l'instruction courante et l'instruction suivante
  - `Goto` : un arc entre instruction courante et cible du `Goto`
  - `Branch` : deux arcs : instruction courante et les deux cibles. Etiquette : T / F
  - `Exit` : sans successeur

Extrait de code Graphviz / Dot

```
digraph {
0 [label= "V1 := n"]
1 [label= "V2 := 0"]
2 [label= "V0 := V1 < V2"]
3 [label= "V0?"]
4 [label= "goto "]
...
0 -> 1
1 -> 2
2 -> 3
3 -> 5 [label= T]
3 -> 4 [label= F]
4 -> 9
...
19 -> 20
}
```

Projet - Statistiques

Lignes de code de l'implantation de références

impl. réf.	déjà fourni	fichier	
29	17	code2graph.ml	code $\rightarrow$ graphe
23	23	comp.ml	
98	9	gen.ml	génération de code
33	33	graph.ml	
186	48	instrs.ml	machine virtuelle
50	40	interf.ml	code glue
74	74	lang.ml	
117	10	typing.ml	typage
34	18	use.ml	
259	44	parser.mly	analyse syntaxique
79	43	lexer.mll	analyse lexicale

Evaluation expression simple (Machine virtuelle)

```
let eval_barith = function
  | BAadd -> ( + )
  | BAsub -> ( - )
  | BAmul -> ( * )
  (* incomplet *)

let eval_binop = function
    BArith b -> eval_barith b
  | BCompar b ->  (* a faire *)

let eval_simple_expr var_arr = function
  | ConstSE n -> n
  | VarSE (IIndex i) -> var_arr.(i)
  | BinOpSE (bop, IIndex i1, IIndex i2) ->
     eval_binop bop var_arr.(i1) var_arr.(i2)
```