Rapport de projet : Extension du compilateur RAT

Mano DINNAT CREUSIER
Rémi AIRIAU

INP ENSEEIHT
2A Architecture Systèmes et Réseaux

Dernière modification : 15/01/2026

UE Traduction des langages

Résumé

Ce rapport présente notre extension du compilateur du langage RAT, réalisé lors des travaux pratiques de traduction de langages. Nous avons implanté toutes les fonctionnalités attendues : les pointeurs, les procédures, le passage de paramètres par références ainsi que les types énumérés. Pour chacune de ces fonctionnalités, nous expliquons et illustrons les modifications appliquées à chaque passe, ainsi que les choix que nous avons fait.

Table des matières

I – Introduction

Dans ce rapport, nous expliquons les modifications effectuées globalement dans chaque passe et dans chaque fonction pour les pointeurs, les procédures, le passage de paramètres par référence ainsi que les types énumérés. Entre autres, nous clarifions les choix de conception, les changements dans la structure des AST, les modifications importantes apportées aux fonctions existantes, ainsi que l'introduction de nouvelles fonctions auxiliaires. De plus, nous avons ajouté les jugements de typage pour chaque fonctionnalité et illustrons parfois nos propos avec du code. À la fin du rapport, nous partageons les difficultés rencontrées et concluons en apportant un point de vue objectif sur le travail rendu ainsi que les différentes perspectives d'amélioration.

II – Pointeurs

Cette extension introduit la gestion explicite de la mémoire dynamique via les mots-clés `new`, `null`, ainsi que les opérateurs d'adresse `&` et de déréférencement `*`.

II.1 Analyse lexicale et syntaxique

Afin de permettre l'utilisation de pointeurs, nous avons dû tout d'abord supprimer les règles suivantes `I -> id = E` ainsi que `E -> id`. En effet, une variable dans son affectation ou son utilisation n'est plus simplement un identificateur. Elle peut être un pointeur sur une variable, voire un pointeur de pointeur, etc. Nous introduisons donc les affectables dans la grammaire. Un affectable est soit un identificateur, soit un pointeur sur un affectable. Nous ajoutons les règles suivantes :

- **`I -> A = E;`** C'est l'affectation, par exemple `x = 5` si `x` est un entier, ou bien `*x = 5` si `x` est un pointeur sur un entier.
- **`A -> id;`** Un affectable peut être un identificateur `x`.
- **`A -> (* A);`** Un affectable peut être un pointeur sur un autre affectable. Cela permet d'exprimer `*x` mais aussi `(*(*(*x)))`
- **`TYPE -> TYPE *;`** Déclaration d'un pointeur. En utilisant la règle `I -> TYPE id = E`, on peut former une instruction telle que `int **x = &y;`.
- **`E -> A;`** Une expression peut contenir un affectable, ce sera alors un identificateur ("appel" d'une variable) ou bien un déréférencement sur un pointeur.
- **`E -> null;`** Pointeur null (adresse 0).
- **`E -> new TYPE;`** Allocation d'un pointeur sur un type.
- **`E -> &id;`** Adresse en mémoire d'un identificateur.

l'`AstSyntax` est ainsi modifié comme suit :

```
type affectable =
  | Ident of string
  | Deref of affectable

(* Expressions de Rat *)
type expression =
  ...
  | Affectable of affectable
  | Adresse of string
  | Alloc of typ
  | Null


(* Instructions de Rat *)
type bloc = instruction list
and instruction =
  ...
  | Affectation of affectable * expression
```

Pour les passes suivantes, les `string` sont transformés en `info_ast`.

II.2 Gestion des identifiants

Dans cette passe, il convient de rétablir le comportement des variables en fonction des nouveaux noeuds liés aux affectables, ainsi que de vérifier la bonne utilisation des pointeurs. Globalement, l'utilisation d'un pointeur est valide si le pointeur a déjà été déclaré auparavant.

Il y a alors deux manières d'analyser un affectable : en lecture (si c'est une expression) ou en écriture (s'il s'agit d'une affectation). On utilise la fonction `analyse_tds_affectable`. La différence entre les deux modes (lecture ou écriture) est expliquée en suivant.

```
let rec analyse_tds_affectable tds aff =
  match aff with
  | AstSyntax.Ident n ->
    begin
    match chercherGlobalement tds n with
    | None -> raise (IdentifiantNonDeclare n)
    | Some ia ->
        match info_ast_to_info ia with
        | InfoEnum _ -> raise (MauvaiseUtilisationIdentifiant n)
        | InfoConst _ -> raise (MauvaiseUtilisationIdentifiant n)
        | InfoFun _  -> raise (MauvaiseUtilisationIdentifiant n)
        | InfoVar _ -> AstTds.Ident ia
    end
  | AstSyntax.Deref a ->
    let na = analyse_tds_affectable tds a in
    AstTds.Deref na
```

II.2.1 `analyse_tds_expression`

Dans la fonction `analyse_tds_expression`, si l'expression est un affectable, on regarde d'abord si c'est un `Ident`. Dans ce cas, il peut s'agir d'une constante que l'on remplace par sa valeur, ou d'une variable à laquelle on associe l'information dans la tds (`ia`). Cependant, si l'expression est un `Deref aff` alors il faut appeler la fonction `analyse_tds_affectable` qui permet l'analyse en écriture, car on ne peut déréférencer que des variables.

Pour l'allocation et le pointeur `null`, on transmet seulement la valeur aux noeuds correspondants de l'AstTds. Pour le cas `AstSyntax.Adresse n`, on cherche globalement dans la tds si un identifiant possède ce nom. Si c'est le cas, on transmet au noeud `Adresse` la même information déjà stockée dans la tds, afin de pouvoir y accéder plus tard en mémoire.

II.2.2 `analyse_tds_instruction`

Pour l'affectation, on analyse l'affectable en écriture. J'ai déjà mentionné la fonction `analyse_tds_affectable`. Dans la passe de gestion des identifiants, on ne vérifie que la portée des variables, donc le déréférencement ne fait qu'appeler récursivement cette fonction. Si l'affectable est un identifiant, on lui associe l'information, si elle existe, présente dans la tds.

II.3 Typage

Il faut tout d'abord étendre les types avec `Pointeur of typ` et modifier la fonction de compatibilité entre deux types. Si `Pointeur (Undefined)` est le pointeur null, alors tout type de pointeur est compatible avec ce dernier. De plus, deux pointeurs sont compatibles si les types pointés sont compatibles. Par contre, aucun pointeur n'est compatible avec un type primitif de notre langage (`Int`, `Rat`, etc.).

Dans cette passe, la fonction `analyse_type_affectable` est valable en lecture et en écriture car il n'y a plus de distinction à faire entre les constantes et les variables, ou dans la portée de ces dernières selon l'expression ou l'instruction traitée. L'objectif est alors de récupérer le type réellement pointé par un pointeur. Pour un identifiant, on le renvoie avec son type. Pour un `AstTds.Deref aff`, on analyse récursivement l'affectable `aff` afin obtenir son type. Puis on analyse le type retourné. Si c'est un `Pointeur t`, on renvoie `t` afin de faire remonter le type dans la pile d'appel.

Les modifications dans l'analyse des expressions et des instructions sont alors évidentes.

II.3.1 `analyse_type_expression`

Pour les noeuds `Adresse ia` et `Alloc t`, on retourne aussi un pointeur vers le type déjà stocké.

```
let rec analyse_type_expression e =
  match e with
  ...
  | AstTds.Affectable a ->
    let (na, ta) = analyse_type_affectable a in
    ((AstType.Affectable na), ta)
  | AstTds.Adresse ia -> (AstType.Adresse ia, Pointeur (getType ia))

  | AstTds.Alloc t -> (AstType.Alloc t, Pointeur t)

  | AstTds.Null -> (AstType.Null, Pointeur Undefined)
```

Dans la gestion des opérateurs binaires, notre programme doit également supporter la comparaison entre deux pointeurs. Nous résolvons donc la surcharge en introduisant un nouvel opérateur `EquAddr` qui prend en paramètre deux expressions.

II.3.2 Jugements de typage

Pour les pointeurs, nous pouvons définir les jugements de typage suivants :

$$\frac{\sigma \vdash TYPE : T}{\sigma \vdash (\text{new } TYPE) : \text{Pointeur } T} \quad \frac{\sigma \vdash e : \text{Pointeur } T}{\sigma \vdash (*e) : T} \quad \frac{\sigma \vdash id : T}{\sigma \vdash \&id : \text{Pointeur } T}$$

$$\frac{\sigma \vdash TYPE : T}{\sigma \vdash (TYPE*) : \text{Pointeur } T} \quad \frac{}{\sigma \vdash null : Pointeur(Undefined)}$$

$$\frac{\sigma \vdash p_1 : \text{Pointeur } T_1 \quad \sigma \vdash p_2 : \text{Pointeur } T_2 \quad \text{compatible}(T_1, T_2)}{\sigma \vdash (p_1 = p_2) : \text{Pointeur } T_1}$$

$$\frac{\sigma \vdash p_1 : \text{Pointeur } T_1 \quad \sigma \vdash p_2 : \text{Pointeur } T_2 \quad \text{compatible}(T_1, T_2)}{\sigma \vdash (p_1 = p_2) : bool}$$

II.4 Placement mémoire

Mis à part l'affichage d'un pointeur, il n'y a presque aucune modification à faire dans la passe de placement mémoire. En effet, un pointeur prend un seul emplacement mémoire mais le fait qu'un affectable soit un pointeur ou non n'a aucune incidence sur le placement des variables ni la taille d'un bloc d'instructions.

II.5 Génération de code

Dans cette passe, on va devoir gérer avec le code TAM la déclaration, l'affectation et le déférencement de pointeurs.

Similairement à la passe de typage, nous avons besoin d'une fonction qui nous permet de récupérer le type associé à un affectable. Nous avons également besoin de deux autres fonctions auxiliaires :
`analyse_code_affectable_valeur` ainsi que
`analyse_code_affectable_adresse`

II.5.1 `analyse_code_affectable_valeur`

```
let rec analyse_code_affectable_valeur a =
  match a with
  | AstType.Ident info ->
      begin
        match info_ast_to_info info with
        | InfoVar (_, t, dep, reg, is_ref) ->
            let taille = getTaille t in
            if is_ref then
               (load 1 dep reg) ^ (loadi taille)
            else
               load taille dep reg
        | InfoConst (_, v) -> loadl_int v
        | InfoEnum (_, _, v) -> loadl_int v
        | _ -> failwith "Erreur interne: identifiant non valide en lecture"
      end
  | AstType.Deref a ->
      let t = get_type_affectable (AstType.Deref a) in
      let taille = getTaille t in
      (analyse_code_affectable_valeur a) ^ (loadi taille)
```

Cette fonction génère le code permettant d'obtenir la valeur d'un affectable. Si l'affectable est un `InfoVar` comme par exemple x, nous faisons simplement un `LOAD` pour récupérer son contenu. Si, par contre, ce n'est pas un identifiant mais un pointeur (`*p`), alors nous chargeons d'abord l'adresse contenue dans p (`LOAD`), puis nous utilisons `LOADI (taille)` pour récupérer sa valeur. Dans le cas où le pointeur ne pointe pas directement vers une valeur (c'est-à-dire possède plusieurs niveaux "d'imbrication"), nous analysons d'abord récursivement l'affectable avant de faire le `LOADI (taille)`. Cela permet de "parcourir" toutes les adresses jusqu'à arriver à la destination finale dans la mémoire et récupérer le contenu.

II.5.2 `analyse_code_affectable_adresse`

```
let analyse_code_affectable_adresse a =
  match a with
  | AstType.Ident info ->
      begin
        match info_ast_to_info info with
        | InfoVar (_, _, dep, reg, is_ref) ->
            if is_ref then
              load 1 dep reg
            else
              loada dep reg
        | _ -> failwith "Erreur interne: Impossible de prendre l'adresse d'une
    constante ou fonction"
      end
  | AstType.Deref a ->
      analyse_code_affectable_valeur a
```

Cette fonction génère le code permettant d'obtenir l'adresse d'un affectable et elle est utile pour gérer la compatibilité des pointeurs avec le passage par référence. Si l'affectable est une variable, on calcule son adresse dans la pile avec `LOADA`. Si c'est un pointeur p sur un autre affectable, son "adresse

de destination" est en réalité la valeur contenue dans p. Nous appelons la fonction précédente pour récupérer cette adresse.

II.5.3 `analyse_code_expression`

- Pour un affectable : on l'analyse avec la fonction `analyse_code_affectable_valeur`
- Pour une allocation : on récupère la taille du type, puis on fait un `LOADL (taille)` suivi de `SUBR "MAlloc"`
- Pour le pointeur null : on fait un `LOADL 0` (choix de conception)

II.5.4 `analyse_code_instruction`

Dans cette fonction, nous ne devons gérer que l'affectation. S'il s'agit d'une affectation simple, nous analysons le code de l'expression associée puis nous faisons un `STORE` afin de stocker le résultat dans la variable modifiée. Si c'est une affectation via un pointeur, comme `*p = e`, nous calculons la valeur de l'expression, chargeons l'adresse contenue dans le pointeur, puis nous stockons le résultat à cette adresse (`STOREI`).

III – Procédures

Cette extension introduit la gestion des procédures, des fonctions qui ne renvoient pas de résultats.

III.1 Analyse lexicale et syntaxique

Leur implémentation n'est en réalité pas très compliquée. Les seules règles de grammaire à rajouter pour le langage RAT sont les suivantes :

- `I -> return;` Comme mentionné, une procédure ne retourne pas de résultat.
- `I -> id (CP);` C'est l'appel d'une procédure avec la liste des paramètres associés. Il convient de remarquer que contrairement à une fonction, un appel de procédure sera toujours une instruction, et non une expression. En effet, nous n'autoriserons pas que le résultat ne soit pas utilisé, car cela impliquerait la déclaration de variables avec le type `Void`.
- `TYPE -> Void;` Ce type n'est utilisé que pour le retour des procédures. Il est important de préciser qu'il n'y a donc pas de pointeurs `Void *` comme en C. Dans ce cas, comme traité précédemment, le pointeur `null` est de type `Pointeur (Undefined)`.

Puis, pour l'`AstSyntax`, nous rajoutons dans les instructions les noeuds suivants :

```
| AppelProcedure of string * expression list
| RetourProcedure
```

Ces noeuds sont modifiés pour les `AST` des passes de gestion des identifiants et de typage en remplaçant le nom de la procédure par le pointeur vers son information dans la tds (`Tds.info_ast`). Nous devons aussi passer ce `Tds.info_ast` au retour de la procédure afin de vérifier l'identité de l'appelant.

```
| AppelProcedure of Tds.info_ast * expression list
| RetourProcedure of Tds.info_ast
```

Enfin, pour l'`AST` placement, le noeud de retour de procédure n'a plus besoin de connaître l'identité de la fonction ou la procédure appelante, mais il faut lui passer la taille des paramètres, comme pour les fonctions.

```
| RetourProcedure of int
```

III.2 Gestion des identifiants

III.2.1 Retour de procédure

Dans cette passe, on doit vérifier (similairement au retour de fonction), que l'instruction `return` pour une procédure n'est pas valide dans le programme principal. Si on trouve une `info_ast` associée à l'instruction dans la tds, alors il s'agit forcément de l'information portée par une procédure. On donne au noeud l'`info_ast` afin de pouvoir comparer le type de retour avec `Void` dans la passe suivante.

III.2.2 Appel de procédure

Toujours dans cette passe de gestion des identifiants, l'appel de procédure est identique (au noeud près de l'`AST`) à celui des fonctions. Une procédure étant considérée comme une fonction, nous lui associons un `InfoFun`. Cette considération est vraie en général. De plus, cela ne complexifie pas le code, et nous évite de rajouter un type supplémentaire que nous devons stocker et traiter, dans tous les noeuds de l'`AST`.

Je rappelle rapidement que le traitement du noeud consiste à vérifier que la procédure existe dans la tds en la recherchant globalement. Si on la trouve, on analyse la liste de ses paramètres puis on modifie le noeud avec l'information et les paramètres analysés.

III.2.3 Procédure vs Fonction

Ici, les noeuds de retour et d'appel de fonctions ne sont pas modifiés. Comme cette passe ne vérifie que la portée des déclarations et que nous avons défini syntaxiquement les procédures de la même façon que les fonctions, alors sémantiquement, ces deux éléments sont presque interchangeables (bien sûr le retour d'une fonction prend toujours une expression).

III.3 Typage

Spécifiquement pour cette passe, des modifications simples doivent être apportées à la fois au noeud `AstTds.RetourProcedure (ia)` mais aussi aux noeuds `AstTds.Retour (e, ia)` et

`AstTds.AppelFonction (ia, lp)`. Le noeud `AstTds.AppelProcedure (ia, lp)` n'a pas besoin d'être modifié. En effet, on considère que l'on peut appeler une fonction comme une procédure, en ignorant le type de retour. Par contre, il convient de vérifier qu'une procédure ne puisse pas être appelée comme une fonction, c'est-à-dire dans une expression. De plus, il faut vérifier que le type de retour d'une fonction ne soit pas `Void`, et inversement, que le type de retour d'une procédure ne soit pas différent de `Void`.

Plus précisément, les modifications à apporter sont les suivantes, pour le noeud :

- `AppelFonction` : si l'information est associée à un `InfoFun (_, Void,_)`, alors on lève une exception.
- `RetourFonction` : si le type de l'information associée à l'instruction est `Void`, on lève une exception.
- `RetourProcedure` : si le type de l'information associée à l'instruction n'est pas `Void`, on lève une exception.

III.3.1 Jugements de typage

Voici les jugements de typage.

III.3.1.1 Déclaration de procédure

$$\frac{A \quad B \quad C}{\sigma \vdash void\ id\ (\text{DP}) : void, [id, \tau_p \to void]}$$

- **A** : $void$
- **B** : $\sigma \vdash DP : \tau_p, \sigma_p$
- **C** : $(id, \tau_p \to void) :: \sigma_p @ \sigma, void \vdash BLOC : void$

III.3.1.2 Appel de procédure

$$\frac{\sigma \vdash id : \tau_1 \to void \quad \sigma \vdash E_1 : \tau_1 \quad ... \quad E_n : \tau_n}{\sigma \vdash id\ (E_1 ... E_n) : void}$$

III.3.1.3 Retour de procédure

$$\frac{}{\sigma, void \vdash return : void, []}$$

III.4 Placement Mémoire

Pour cette passe, il n'y a presque pas de modifications à apporter. Les compatibilités de types définissant les règles d'utilisation d'une procédure ou d'une fonction ayant été prises en compte dans la passe de typage, nous avons juste à calculer la taille des paramètres dans le retour afin de gérer la pile mémoire dans la passe suivante. Pour l'appel, nous gardons l'information du noeud et la taille d'un appel de procédure est 0.

III.5 Génération de code

Là encore, il n'y a pas beaucoup de modifications à faire. Pour le retour de procédure, nous appelons l'instruction `RETURN (0) tailleParam` et pour l'appel, nous faisons un `CALL "SB" (nom)`.

III.5.1 Sécurité d'exécution : le Return implicite

En RAT, une procédure n'a pas l'obligation syntaxique de se terminer par **`return`**, tout comme une fonction. D'ailleurs, sans se préoccuper du type de retour, utiliser une fonction qui ne renvoie rien est déjà possible. Nous l'avions déjà testé car un ancien sujet d'examen de Traduction des Langages le mentionnait. L'objectif principal est alors de s'assurer que le programme fonctionne correctement. Par exemple, si l'utilisateur omet le **`return`**, la machine TAM risquerait d'exécuter les instructions suivantes en mémoire, car elle ne rencontrerait jamais le `RETURN (0) tailleParam`.

Pour pallier ce problème, dans la fonction **`analyse_code_fonction`**, si le type de retour de la fonction traitée est void, nous injectons le `RETURN (0) tailleParam`, sinon nous injectons un `HALT` après l'analyse du bloc. Cela garantit un nettoyage propre de la pile et l'arrêt correct de la fonction ou de la procédure, même en l'absence du **`return`** explicite.

```
let analyse_code_fonction (AstPlacement.Fonction (info, _, (li, u))) =
  match info_ast_to_info info with
  | InfoFun (nom, t, params) ->
      let taille_params = List.fold_right
        (fun (typ, is_ref) acc -> acc + (if is_ref then 1 else getTaille typ))
        params 0
      in

      let fin_fonction =
        match t with
        | Void -> return 0 taille_params
        | _ -> halt
      in

      (label nom) ^ (analyse_code_bloc (li, u)) ^ fin_fonction

  | _ -> failwith "Erreur interne: Fonction mal formee"
```

IV – Types Énumérés

Les types énumérés permettent de définir des ensembles de constantes nommées, ce qui permet une meilleure lisibilité par rapport à l'utilisation de simples entiers.

IV.1 Représentation Sémantique (TDS)

Dans notre implémentation, les énumérations sont traitées comme des types distincts lors de l'analyse sémantique, puis comme des entiers pour la génération de code. En effet, un type énuméré peut être réduit à un index auquel on associe une étiquette (nom de l'énum).

Lors de la passe TDS, nous utilisons la fonction `analyse_labels_enum` pour parcourir les valeurs déclarées. À chaque étiquette, nous associons une `InfoEnum` contenant :

1. Le nom du type énuméré parent (pour vérifier l'unicité).
2. La valeur de l'étiquette.
3. Un index entier

```
(* Extrait de passeTdsRat.ml *)
let rec analyse_labels_enum tds nom_enum labels index =
  match labels with
  | [] -> []
  | label::reste ->
    (* Verification de l'unicite globale du label *)
    match chercherGlobalement tds label with
    | Some _ -> raise (DoubleDeclaration label)
    | None ->
      (* Creation de l'InfoEnum avec l'index courant *)
      let infoVal = InfoEnum(nom_enum, label, index) in
      (* ... ajout a la TDS et recursion avec index + 1 *)
```

IV.2 Typage

Le typage des énumérations est strict. Il est interdit de comparer un énuméré avec un entier ou avec un autre type énuméré.

IV.2.1 Implémentation

Nous avons modifié l'analyse des opérations binaires (`AstType.Binaire`) pour gérer l'égalité (`Equ`) de variables de type énumérés. Si les deux opérandes sont des `EnumType`, nous vérifions qu'ils ont bien le même nom ($n1 = n2$).

```
(* Extrait de passeTypeRat.ml *)
| Equ, EnumType n1, EnumType n2 ->
    if (n1 = n2) then (AstType.Binaire (EquInt, ne1, ne2), Bool)
    else raise (TypeBinaireInattendu (op, EnumType n1, EnumType n2))
```

Cette rigueur garantit qu'on ne peut pas comparer deux énums différents mais de même valeur (exemple : `Mois m = 0` et `Jour j = 0`).

IV.2.2 Règles d'Inférence

Voici la formalisation des règles de typage Γ l'environnement de typage.

Accès à une valeur énumérée Lorsqu'on utilise un identifiant correspondant à une énumération, son type est celui de l'énumération parente.

$$\frac{\Gamma(id) = \text{Enum}(E, \text{val})}{\Gamma \vdash id : \text{Enum}(E)}$$

Égalité L'égalité n'est autorisée que si les deux expressions sont du même type énuméré E.

$$\frac{\Gamma \vdash e_1 : \text{Enum}(E) \quad \Gamma \vdash e_2 : \text{Enum}(E)}{\Gamma \vdash (e_1 = e_2) : \text{Bool}}$$

IV.3 Génération de Code

À l'exécution, l'abstraction des types énumérés disparaît : ils sont représentés par des entiers. Concrètement, lors de la génération de code, dès qu'on rencontre une valeur énumérée (`EnumVal`) on charge son index numérique via l'instruction `LOADL`. De même, les tests d'égalité entre énumérations sont traduits en instructions de comparaison d'entiers (`IEq`). TAM ne manipule ainsi que des valeurs entières.

IV.4 Choix d'implémentation

Le choix du stockage des informations des types énumérés (index et étiquette) nous a paru plus logique et nous n'avons pas essayé d'autres implémentations. Une autre approche possible aurait consisté à représenter chaque valeur par sa chaîne de caractères. Cette méthode aurait été coûteuse à l'exécution : la comparaison de chaînes est une opération complexe ($O(n)$) et la gestion mémoire est plus lourde.

V – Le Passage par Référence

Le passage par référence permet à une fonction de modifier directement la variable passée en paramètre, en transmettant son adresse mémoire plutôt que sa valeur. Cette fonctionnalité introduit la possibilité d'effets de bord sur les arguments.

V.1 Analyse Sémantique et Choix d'Implémentation

Nous avons opté pour une architecture où la référence est traitée comme une propriété indépendante du type de la variable.

V.1.1 Représentation dans la TDS

Afin de distinguer la nature de la variable (son type) de son mode d'accès (direct ou indirect), nous avons enrichi les structures de données de la TDS (`Tds.ml`) sans altérer la définition des types eux-mêmes.

- **Modification de `InfoVar`** : Le constructeur a été étendu avec un champ booléen supplémentaire, nommé `is_ref`.
  — Si `is_ref` vaut `false`, la variable est locale et stockée par valeur.
  — Si `is_ref` vaut `true`, la variable est un paramètre référence contenant une adresse.
  Ce choix permet de conserver le type originel (par exemple `Int`) pour les vérifications arithmétiques, tout en signalant au générateur de code qu'un déréférencement est nécessaire.
- **Modification de `InfoFun`** : La signature des fonctions, initialement une liste de types, est devenue une liste de paires `(type * bool)`. Cela permet de mémoriser pour chaque argument s'il est attendu par valeur ou par référence, information indispensable lors de la validation des appels.

V.2 Contrôle de Type et Règles d'Inférence

La sûreté du langage impose des contraintes strictes lors des appels de fonctions. Dans le module `PasseTypeRat`, nous avons implémenté une fonction dédiée `verifier_types_params` qui valide simultanément deux invariants pour chaque argument.

V.2.1 Jugement de typage

Voici la règle de typage unifiée pour l'appel (fonction ou procédure) prenant en compte les références, structurée selon votre format :

$$\frac{A \quad B \quad C}{\sigma \vdash id(E_1, \ldots, E_n) : T_{ret}}$$

- **A** : $\sigma(id) = [(T_1, m_1), \ldots, (T_n, m_n)] \to T_{ret}$
- **B** : $\forall i \in [1..n], \quad \sigma \vdash E_i : \tau_i$
- **C** : $\forall i \in [1..n], \quad \tau_i \approx T_i \quad \wedge \quad (m_i = \text{ref} \implies \text{affectable}(E_i))$

Cette règle résume les invariants vérifiés par notre fonction `verifier_types_params` :

- **A** : On récupère la signature complète dans la TDS, incluant les types T_i et les modes m_i (valeur ou référence).
- **B** : Chaque argument E_i doit être correctement typé dans l'environnement courant.
- **C** : Le type réel doit être compatible avec le type attendu. De plus, si le paramètre est une référence ($m_i = \text{ref}$), l'argument E_i doit être une variable ou un déréférencement.

V.2.2 Implémentation

Voici l'extrait correspondant de notre vérificateur, qui applique strictement ces règles :

```
(* Extrait de passeTypeRat.ml *)
let rec verifier_types_params args_exprs args_types def_params =
  match args_exprs, args_types, def_params with
  (* ... *)
  | (exp::exps), (ty_arg::tys), ((ty_param, is_ref_param)::pars) ->
      (* Detection de la syntaxe 'ref' dans l'appel *)
      let is_ref_arg = match exp with AstType.Reference _ -> true | _ -> false in

      (* 1. Validation du mode de passage *)
      if is_ref_arg <> is_ref_param then raise ErreurReference

      (* 2. Validation du type *)
      else if not (est_compatible ty_arg ty_param) then
        raise (TypeInattendu (ty_arg, ty_param))
      else verifier_types_params exps tys pars
```

V.3 Gestion Mémoire et Génération de Code

Pour la machine cible (TAM), une référence n'est rien d'autre qu'un pointeur masqué.

V.3.1 Optimisation de l'Espace (Placement)

Puisqu'une référence est une adresse, sa taille mémoire est invariante et vaut toujours 1 mot, indépendamment du type de la donnée pointée. On utilise cette propriété dans `passePlacementRat.ml` : même un type comme `Rat` (qui occupe normalement 2 mots) ne consommera qu'un seul mot sur la pile s'il est passé par référence.

```
(* Extrait de passePlacementRat.ml *)
| InfoVar (_, t, _, _, is_ref) ->
    (* Si ref, taille 1 (pointeur), sinon taille du type *)
    let taille = if is_ref then 1 else getTaille t in
```

V.3.2 Génération de Code (TAM)

Dans la passe finale, nous utilisons le booléen `is_ref` pour piloter la stratégie d'accès aux variables, en insérant les déréférencements nécessaires.

Lecture d'une valeur L'accès en lecture à une variable référence nécessite une double opération :

- `LOAD 1` : Chargement de l'adresse contenue dans la variable (sur la pile).
- `LOADI taille` : Déréférencement pour lire la valeur stockée à cette adresse.

Écriture (L'Affectation) L'instruction d'affectation `x = e` est un point problématique. Si `x` est une référence, une affectation standard (`STORE`) écraserait l'adresse elle-même, causant des problèmes de mémoire. Nous utilisons donc l'instruction d'écriture indirecte :

1. Calcul et empilement de la valeur de l'expression `e`.
2. Chargement de l'adresse de destination contenue dans `x` (`LOAD 1`).
3. Exécution de `STOREI (taille)` pour écrire la valeur à l'adresse pointée.

Passage d'argument Lors d'un appel de fonction avec `ref x`, deux cas de figures se présentent pour générer l'adresse à passer :

- **Si x est une variable locale :** Nous calculons son adresse dans la pile via l'instruction `LOADA`.
- **Si x est déjà une référence (propagation) :** La variable contient déjà l'adresse souhaitée. Nous utilisons simplement `LOAD 1` pour copier cette adresse.

VI – Conclusion

VI.1 Bilan Technique

Ce projet de traduction des langages nous a permis de mettre en œuvre la chaîne complète de compilation, de l'analyse syntaxique jusqu'à la génération du code machine, dans un langage fonctionnel (`OCaml`). Les effets de bord ont seulement été utilisés pour les mises à jour dans la TDS. Aussi, comme précisé dans le résumé, toutes les fonctionnalités demandées ont été implantées avec succès : pointeurs, procédures, références et types énumérés. Notre code est aussi lisible, maintenable et robuste : l'ajout de ces extensions n'a en réalité nécessité que quelques lignes ou fonctions auxiliaires supplémentaires par passe. Certaines fonctions ne sont même pas modifiées (par exemple, les énumérateurs n'introduisent pas de gestion spécifique dans la passe de placement).

Par contre, nous avons rajouté un peu plus d'une centaine de tests ! Pour une question de simplicité, nous avons placé les fichiers dans les mêmes dossiers que les tests déjà présents. Le code `OCaml` est aussi dans les mêmes fichiers `test.ml`. Cependant, tous les `.rat` que nous avons ajoutés pour le projet sont préfixés de `projet_`, et dans le code, les tests unitaires sont à la fin, après un commentaire.

VI.2 Difficultés Rencontrées

La principale difficulté a résidé dans l'intégration du passage par référence. Contrairement aux autres fonctionnalités qui sont relativement isolées et indépendantes, les références modifient la nature même de l'accès aux variables dans toutes les passes. Il a fallu propager l'information `is_ref` de la TDS jusqu'à la génération de code et adapter dynamiquement le calcul des adresses (instruction `LOADA` vs `LOAD`) pour éviter les erreurs de segmentation ou de corruption de pile.

VI.3 Perspectives

Si notre compilateur est fonctionnel, il reste perfectible sur la gestion de la mémoire dynamique. Actuellement, la mémoire allouée par `new` n'est jamais libérée, ce qui entraîne des fuites de mémoire dans le tas. Même si ce n'était pas demandé, une évolution naturelle de ce projet serait l'implémentation d'une instruction `free`.

Enfin, nous aurions également pu rajouter d'autres fonctionnalités que nous avons vues dans des annales d'examens de Traduction des Langages, telles que l'instruction `switch` beaucoup utilisée en langage C, ou encore les listes ou les `struct`. Cependant, cela aurait demandé plus de travail et nous ne l'avons pas fait par manque de temps.

VI.4 IA générative

`Gemini 3` a de temps à autre été utilisé pour aider à la rédaction des tests et de certains commentaires. Cependant, aucune IA n'a été utilisée pour le code, car notre avancement après la dernière séance de Travaux Pratiques nous a permis de terminer assez rapidement.