Troisième partie

Programmation impérative

Plan

8 Analyse des types **114**
8.1 Données et opérations typées . . . 114
8.2 Analyse statique des types. . . . 116
8.3 Jugement de typage et règles d'inférence . . . 117
8.4 Des règles de typage au vérificateur de types . . . 119
8.5 Raisonner sur les expressions typées . . . 120
8.6 Approfondissement : sûreté des programme typés . . . 120
8.7 Approfondissement : le langage IMPScript . . . 122

9 Fonctions et pile d'appels **126**
9.1 Appel de fonction : mécanisme de base . . . 126
9.2 Pile d'appels . . . 127
9.3 Convention d'appel . . . 129
9.4 Approfondissement : génération de code pour ImpScript . . . 132
9.5 Approfondissement : convention d'appel avec registres . . . 135
9.6 Approfondissement : optimisation des appels terminaux . . . 137

10 Structures de données et tas mémoire **140**
10.1 Tableaux . . . 140
10.2 Gestion dynamique de la mémoire . . . 143
10.3 Structures de données . . . 147

8 Analyse des types

Maintenant que nous disposons d'outils d'analyse syntaxique qui peuvent s'appliquer à l'échelle d'un langage de programmation réaliste, nous allons nous intéresser à des langages plus riches que le noyau IMP étudié aux chapitres précédents. En particulier, nous allons diversifier les données que peut manipuler un langage.

8.1 Données et opérations typées

À l'intérieur de l'ordinateur, une donnée est une séquence de bits. Voici par exemple un mot mémoire de 32 bits.

```
1110 0000 0110 1100 0110 1111 0100 1000
```

Pour faciliter la lecture, on représente souvent un tel mot au format hexadécimal. En l'occurrence, on l'écrirait

```
0x e0 6c 6f 48
```

(le `0x` au début indique simplement le format hexadécimal, puis chaque caractère correspond à un groupe de 4 bits).

Que peut signifier cette donnée ? Pour le savoir, il faut une connaissance *très* précise du contexte :

- si la donnée est une adresse mémoire, il s'agira de l'adresse `3 765 202 760`,
- si la donnée est un nombre entier signé 32 bits en complément à 2, ce nombre sera $-529\ 764\ 536$,
- si la donnée est un nombre flottant simple précision de la norme IEEE754, ce nombre sera $-15\ 494\ 984 \times 2^{42}$,
- si la donnée est une chaîne de caractères au format Latin-1, il s'agira de `"Holà"`.

Exemple : appliquer l'addition entière aux représentations des chaînes de caractères `"5"` et `"37"` produit la nouvelle chaîne `"h7"`. Voyez-vous pourquoi ?

Si on oublie le contexte dans lequel une séquence de bits a du sens, on est susceptible de faire n'importe quoi.

Opérations incohérentes. Pour illustrer les conséquences de cette cohabitation entre des données de différents types, écrivons un évaluateur pour un ensemble d'expressions mêlant arithmétique, variables et tableaux. On se donne la syntaxe abstraite suivante :

```
e  ::=  n                    type expr = Int of int
     |  x                              | Var of string
     |  e - e                          | Sub of expr * expr
     |  [e, ... , e]                   | Arr of expr list
     |  e[e]                            | Get of expr * expr
```

où on demande que chaque tableau $[e_1, \ldots, e_k]$ contienne au moins un élément. L'évaluateur doit produire des ***valeurs*** qui peuvent être de deux natures différentes : des nombres entiers, ou des tableaux dont chaque élément est également une valeur. Notez que dans la représentation caml des valeurs « tableaux », on réutilise directement le type `array` des tableaux natifs de caml.

Techniquement, cette valeur est en caml un pointeur vers le tableau lui-même.

```
v  ::=  n                    type value = VInt of int
     |  [v, ... , v]                   | VArr of value array
```

Notre fonction d'évaluation, comme on a déjà pu le voir, prend en paramètres une expression e, un environnement ρ qui à chaque variable associe une valeur, et vérifie les équations suivantes.

$$
\begin{aligned}
\mathrm{eval}(n, \rho) &= n \\
\mathrm{eval}(x, \rho) &= \rho(x) \\
\mathrm{eval}([e_1, \ldots, e_k], \rho) &= [\mathrm{eval}(e_1, \rho), \ldots, \mathrm{eval}(e_k, \rho)] \\
\mathrm{eval}(e_1 - e_2, \rho) &= n_1 - n_2 \quad \text{si} \begin{cases} \mathrm{eval}(e_1, \rho) = n_1 \\ \mathrm{eval}(e_2, \rho) = n_2 \end{cases} \\
\mathrm{eval}(e_1[e_2], \rho) &= v_n \quad \text{si} \begin{cases} \mathrm{eval}(e_1, \rho) = [v_1, \ldots, v_k] \\ \mathrm{eval}(e_2, \rho) = n \end{cases}
\end{aligned}
$$

On voit avec ces équations quelques conditions supplémentaires, qui demandent à la valeur d'une sous-expression d'avoir une forme particulière. En effet, toutes les opérations ne s'appliquent pas à toutes les valeurs : il n'est pas possible par exemple d'additionner un nombre et un tableau

(`1 + [2, 3, 4]`), ou d'accéder à la troisième case d'un nombre entier (`12345[3]`). La fonction d'évaluation, mise en présence d'une telle situation, ne peut pas aboutir.

Dans le code caml, ces conditions se traduisent par un raisonnement par cas sur la forme des valeurs produites par l'évaluation des sous-expressions `e1` et `e2`, et le déclenchement éventuel d'une erreur `"unsupported␣operation"` si ces valeurs n'ont pas la bonne forme.

Dans un tel cas, le problème ne vient pas de eval, mais d'une erreur du programmeur !

```
module Env = Map.Make(String)
type env = value Env.t

let rec eval e env = match e with
  | Int n -> VInt n
  | Var x -> Env.find x env
  | Arr l -> VArr (Array.of_list(List.map (fun e -> eval e env) l))

  | Sub(e1, e2) -> (match eval e1 env, eval e2 env with
                    | VInt n1, VInt n2 -> VInt(n1+n2)
                    | _ -> failwith "unsupported␣operation")

  | Get(e1, e2) -> (match eval e1 env, eval e2 env with
                    | VArr a, VInt i -> a.(i)
                    | _ -> failwith "unsupported␣operation")
```

Nous avons donc un évaluateur qui peut échouer dans trois situations : d'une part avec l'erreur `"unsupported␣operation"` si, à un moment donné, l'une des valeurs obtenues n'est pas de la bonne nature, d'autre part avec une erreur `"not␣found"` si l'on tente d'accéder à une variable qui n'est pas dans l'environnement, et enfin avec une erreur `"index␣out␣of␣bounds"` si l'accès à un tableau se fait bien à un indice entier, mais avec une valeur trop grande ou trop petite. Notre objectif à ce chapitre est d'éradiquer les deux premières situations, par une analyse préalable du programme.

Les types : une classification des valeurs. Les différentes catégories de valeurs que manipule un langage de programmation sont appelées des ***types***. La classification précise dépend de chaque langage, mais on y retrouve souvent de nombreux éléments communs. On a d'une part des types de base du langage, pour les données simples. Par exemple :

- nombres : `int`, `double`, `number`,
- valeurs booléennes : `bool`,
- caractères et chaînes : `char`, `string`,
- pointeurs : `void*`.

D'autre part, on peut combiner ces types de base pour en construire d'autres plus riches. Par exemple :

- tableaux : `int[]` (C, java), `int array` (caml),
- fonctions : `int -> bool` (caml),
- structures : `struct point { int x; int y; };` (C),
- objets : `class Point { public final int x, y; ... }` (java).

Types et opérations. Une fois cette classification établie, chaque opération va s'appliquer à des éléments d'un type donné.

- L'addition `5 + 37` entre deux entiers est possible en caml.
- Les opérations `"5" + 37` ou `"5" + "37"` ou `5 + (fun x -> 37)` ou `5(37)` ne le sont pas.

Dans certains cas, un même opérateur peut s'appliquer à plusieurs types d'éléments, avec des significations différentes à chaque fois. On parle de ***surcharge***. En python et en java par exemple, l'opérateur + peut s'appliquer :

- à deux entiers, et désigne alors l'addition : `5 + 37 = 42`,
- à deux chaînes, et désigne alors la concaténation : `"5" + "37" = "537"`.

Les langages de programmation permettent également parfois le ***transtypage*** (*cast*), c'est-à-dire la conversion d'une valeur d'un type vers un autre. Cette conversion peut même être implicite. Ainsi l'opération `"5" + 37` mélangeant une chaîne et un entier aura comme résultat :

- `42` en php, où la chaîne `"5"` est convertie en le nombre `5`,
- `"537"` en java, où l'entier `37` est converti en la chaîne `"37"`.

Notez qu'une telle conversion peut demander une traduction de la donnée ! Le nombre `5` est représenté par le mot mémoire `0x 00 00 00 05`, mais la chaîne `"5"` par le mot mémoire `0x 00 00 00 35`. De même, le nombre `37` est représenté par le mot mémoire `0x 00 00 00 25`, mais la chaîne `"37"`

par le mot mémoire `0x 00 00 37 33`. Dans un sens comme dans l'autre, une conversion d'un type à l'autre demande de calculer la nouvelle représentation.

Bilan. Le type d'une valeur donne une clé d'interprétation de cette donnée, et peut être utile à la sélection des bonnes opérations. En outre, une incohérence dans les types révèle un problème du programme, dont l'exécution doit donc être évitée.

8.2 Analyse statique des types.

Gérer les types des données au moment de l'exécution génère des coûts variés :
- de la mémoire pour accompagner chaque donnée d'une indication de son type,
- des tests pour sélectionner les bonnes opérations,
- des exécutions interrompues en cas de problème...

Dans des langages ***typés dynamiquement*** comme python ou javascript, ces coûts sont la norme. En revanche, les langages ***typés statiquement*** comme C, java ou caml nous épargnent tout ou partie de ces coûts à l'exécution en gérant autant que possible tout ce qui concerne les types dès la compilation.

Dans l'analyse ***statique*** des types, c'est-à-dire l'analyse des types *à la compilation*, on associe un type *à chaque expression* d'un programme, et plus seulement aux valeurs. On attend du type associé à une expression qu'il *prédise* le type de la valeur qui sera produite par cette expression. Cette prédiction est basée sur des contraintes associées à chaque élément de la syntaxe abstraite. Par exemple, en présence d'une expression de soustraction de la forme `Sub(e1, e2)` nous pouvons noter deux faits :
- l'expression produira un nombre,
- les deux sous-expressions `e1` et `e2` doivent impérativement produire des valeurs numériques, sans quoi l'ensemble serait mal formé.

De même, le type d'une variable désigne le type de la donnée référencée par cette variable, et le type d'une fonction précise ce que sont les types attendus pour chacun des paramètres, ainsi que le type du résultat renvoyé.

Vérification ou inférence. L'analyse statique des types peut prendre plusieurs formes, demandant plus ou moins d'intervention du programmeur.

- Dans un langage comme C ou java, le programmeur doit déclarer les types de tous les éléments constituant son programme.

```
void swap(string[] a, int i, int j) {
  string tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

  L'analyse de types faite à la compilation doit alors simplement ***vérifier*** la cohérence des différentes opérations avec les indications du programmeur.
- Dans un langage comme caml, le programmeur ne donne en général pas d'indications.

```
let swap a i j =
  let tmp = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- tmp
```

  L'analyse est pourtant faite ! Le compilateur doit ici ***inférer*** le type de chaque variable et chaque expression, c'est-à-dire le *déduire* du programme, et en particulier de la manière dont chaque expression est définie ou utilisée.

Sûreté et efficacité des programmes typés. Le slogan associé à cette vérification de la cohérence des types avant l'exécution des programmes, du à Robin Milner, est

Well-typed programs do not go wrong.

Le premier objectif du typage statique est ainsi de repérer certaines erreurs du programmeur avant même que le programme soit exécuté (ou livré à un client...). Il est particulièrement efficace pour détecter de manière précoce les « petites » erreurs qui rendent un programme absurde, qui sont à la fois fréquentes et souvent simples à corriger, *une fois identifiées*. Dans le cas où un programme est incohérent, on attend de l'analyse des types qu'elle identifie aussi précisément que possible les endroits qui posent problème.

Autrement dit, une valeur ajoutée majeure de l'analyse des types est dans les erreurs qu'elle détecte ! mais aussi dans les messages qu'elle produit le cas échéant.

Dans l'absolu, on ne peut pas identifier avec certitude tous les programmes problématiques, les questions de ce genre étant généralement algorithmiquement indécidables. On cherche donc à établir des critères décidables qui :

— apportent de la ***sûreté***, c'est-à-dire qui rejettent les programmes absurdes,
— tout en laissant de l'***expressivité***, c'est-à-dire qui ne rejettent pas trop de programmes non-absurdes.

Lorsque le programme est cohérent, et qu'il n'y a donc pas d'erreur à détecter, l'analyse statique des types permet en outre de déterminer dès la compilation les versions concrètes à sélectionner pour certains opérateurs surchargés. On se dispense ainsi du coût de faire ce choix pendant l'exécution, et le programme obtenu est plus efficace.

8.3 Jugement de typage et règles d'inférence

Pour caractériser les programmes bien typés, on définit des règles permettant de justifier que « dans un contexte Γ, une expression e est cohérente et admet le type τ ». Cette phrase entre guillemets est appelée un ***jugement de typage*** et est notée

$$\Gamma \vdash e : \tau$$

Le contexte Γ mentionné dans le jugement de typage est l'association d'un type à chaque variable de l'expression e. On note $\Gamma(x)$ le type que le contexte Γ associe à la variable x.

Le jugement de typage n'est pas une fonction associant un type à chaque expression, mais simplement une relation entre ces trois éléments : contexte, expression, type. En particulier certaines expressions *e n'ont pas* de type (parce qu'elles sont incohérentes), et dans certaines situations on peut avoir plusieurs types possibles pour une même expression.

Règles de typage. Pour illustrer la manière dont on peut formaliser la cohérence et le type d'une expression, reprenons notre exemple d'expressions avec arithmétique, variables et tableaux. Nous aurons donc besoin de manipuler un type de base pour les nombres entiers, et des types de tableaux.

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & \mid & \tau\texttt{[]} \end{array}$$

Un type de la forme τ`[]` est le type d'un tableau dont les éléments sont tous du type τ. À chaque construction du langage, on va associer une règle énonçant

— le type que peut avoir une expression de cette forme, et
— les éventuelles contraintes qui doivent être vérifiées pour que l'expression soit cohérente.

Commençons avec la partie arithmétique. On donnera chaque règle sous deux formes : une description en langue naturelle, et sa traduction comme une ***règle d'inférence*** (voir cours de logique !). On donne un nom à chaque règle pour référence ultérieure.

— Une constante entière n admet le type `int`.

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{Int}$$

— Si les expressions e_1 et e_2 sont cohérentes et admettent le type `int`, alors l'expression e_1 `-` e_2 est cohérente et admet également le type `int`.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}}\ \textsc{Sub}$$

Dans la règle pour la soustraction, les deux jugements $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$ sont les ***prémisses***, et le jugement $\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}$ est la ***conclusion***. Autrement dit, si l'on a pu d'une manière ou l'autre justifier $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$, alors la règle permet d'en déduire $\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}$. À l'inverse, la règle pour la constante entière n'a pas de prémisse (on l'appelle un ***axiome***, ou ***cas de base***). Cela signifie que nous n'avons besoin de rien d'autre que cette règle pour justifier un jugement $\Gamma \vdash n : \texttt{int}$.

La règle concernant les variables va faire intervenir le contexte Γ, aussi appelé ***environnement***, puisque c'est lui qui consigne les types associés à chaque variable.

— Une variable a le type donné par l'environnement.

$$\frac{}{\Gamma \vdash x \ : \ \Gamma(x)}\ \textsc{Var}$$

Notez ici que l'on considère Γ comme une fonction : $\Gamma(x)$ désigne le type associé par Γ à la variable x. En outre, l'application de cette règle suppose que $\Gamma(x)$ est bien définie, c'est-à-dire que x appartient au domaine de Γ.

Un tableau a un type de la forme τ`[]`, où τ est le type des éléments du tableau.

— La construction explicite d'un tableau demande que tous les éléments aient le même type. On parle ici de ***tableaux homogènes***.

$$\frac{\Gamma \vdash e_1 \ : \ \tau \qquad \ldots \qquad \Gamma \vdash e_k \ : \ \tau}{\Gamma \vdash \texttt{[}e_1\texttt{, ..., }e_k\texttt{]} \ : \ \tau\texttt{[]}}\ \textsc{Arr}$$

— L'opération d'accès est cohérente si l'expression de gauche a un type de tableau τ`[]` (pour un type de contenu homogène τ quelconque) et si l'expression entre crochets admet le type `int`. L'ensemble a alors le type τ des éléments du tableau.

$$\frac{\Gamma \vdash e_1 \ : \ \tau\texttt{[]} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} \ : \ \tau}\ \textsc{Get}$$

Les règles des ***types simples*** pour notre petit langage d'expressions sont donc intégralement contenues dans les cinq règles d'inférence suivantes.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}}\ \textsc{Int} \qquad \frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 \ : \ \texttt{int}}\ \textsc{Sub} \qquad \frac{}{\Gamma \vdash x \ : \ \Gamma(x)}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau \qquad \ldots \qquad \Gamma \vdash e_k \ : \ \tau}{\Gamma \vdash \texttt{[}e_1\texttt{, ..., }e_k\texttt{]} \ : \ \tau\texttt{[]}}\ \textsc{Arr} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau\texttt{[]} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} \ : \ \tau}\ \textsc{Get}$$

Expressions typables et dérivations. Pour justifier un jugement de typage pour une expression concrète dans un contexte donné, on enchaîne les déductions à l'aide des règles d'inférence. Ainsi, dans le contexte $\Gamma = \{ x \ : \ \texttt{int}, \ t \ : \ \texttt{int[]} \}$ on peut tenir le raisonnement suivant.

1. $\Gamma \vdash x \ : \ \texttt{int}$ est valide par la règle Var.
2. $\Gamma \vdash t \ : \ \texttt{int[]}$ est valide par la règle Var.
3. $\Gamma \vdash \texttt{1} \ : \ \texttt{int}$ est valide par la règle Int.
4. $\Gamma \vdash t\texttt{[1]} \ : \ \texttt{int}$ est valide par la règle Get, avec les deux points 2. et 3. déjà justifiés.
5. $\Gamma \vdash x \texttt{ - } t\texttt{[1]} \ : \ \texttt{int}$ est valide par la règle Sub, avec les points 1. et 4. déjà justifiés.

Ce raisonnement, appelé une ***dérivation***, peut également être présenté sous la forme d'un ***arbre de dérivation*** ayant à la racine la conclusion que l'on cherche à justifier.

$$\dfrac{\dfrac{}{\Gamma \vdash x \ : \ \texttt{int}}\ \textsc{Var} \qquad \dfrac{\dfrac{}{\Gamma \vdash t \ : \ \texttt{int[]}}\ \textsc{Var} \qquad \dfrac{}{\Gamma \vdash \texttt{1} \ : \ \texttt{int}}\ \textsc{Int}}{\Gamma \vdash t\texttt{[1]} \ : \ \texttt{int}}\ \textsc{Get}}{\Gamma \vdash x \texttt{ - } t\texttt{[1]} \ : \ \texttt{int}}\ \textsc{Sub}$$

Dans un tel arbre, chaque barre correspond à une application de règle, et chaque sous-arbre à la justification d'un jugement intermédiaire (une prémisse).

Expressions non typables. Si une expression e est incohérente, les règles de typage *ne permettront pas* de justifier de jugements de la forme $\Gamma \vdash e \ : \ \tau$, quels que soient le contexte Γ ou le type τ. On peut le voir en montrant que la construction d'un hypothétique arbre de dérivation justifiant un tel jugement arrive nécessairement à une impasse, c'est-à-dire une situation dans laquelle il est clair que plus aucune règle ne permet de conclure.

Prenons l'exemple de l'expression `5[37]`. Il s'agit d'un accès à une case d'un tableau. La seule règle permettant de justifier un jugement $\Gamma \vdash \texttt{5[37]} \ : \ \tau$ est la règle Get relative aux accès à un tableau, qui demande de justifier au préalable les deux prémisses $\Gamma \vdash \texttt{5} \ : \ \tau\texttt{[]}$, pour un certain type τ, et $\Gamma \vdash \texttt{37} \ : \ \texttt{int}$. Or il est impossible de justifier une prémisse de la forme $\Gamma \vdash \texttt{5} \ : \ \tau\texttt{[]}$: aucune règle ne permet d'associer à une constante entière un type de tableau (en effet, la seule règle applicable à une constante entière est Int, qui donnerait $\Gamma \vdash \texttt{5} \ : \ \texttt{int}$).

8.4 Des règles de typage au vérificateur de types

Si suffisamment d'informations sont fournies dans le programme source pour préciser les types des différentes variables, on peut facilement déduire des règles de typage un ***vérificateur*** de types, c'est-à-dire un (autre) programme qui dit si le programme analysé est cohérent ou non. On va écrire un programme caml pour la vérification des types dans notre petit langage d'expressions. Ce programme prendra la forme d'une fonction `type_expr` qui prend en paramètres une expression e et un environnement Γ et qui :

— renvoie l'unique type qui peut être associé à e dans l'environnement Γ si e est effectivement cohérente dans cet environnement,
— échoue sinon.

On définit un type de données (caml) pour manipuler en caml les types de notre langage d'expressions. On va représenter les environnements comme des tables associatives associant des identifiants de variables (`string`) à des types (`typ`).

```
type typ =
  | TInt            (*  type int  *)
  | TArr of typ     (*  type t[]  *)

module Env = Map.Make(String)
type typ_env = typ Env.t
```

Le vérificateur est alors une fonction récursive

```
type_expr: expr -> typ_env -> typ
```

qui observe la forme de l'expression et traduit la règle d'inférence correspondante.

```
let rec type_expr e env = match e with
```

Une constante entière est toujours cohérente, et de type `int`.

```
  | Int _ -> TInt
```

$$\frac{}{\Gamma \vdash n : \texttt{int}}$$

Une variable est considérée comme cohérente si elle existe effectivement dans l'environnement (et l'environnement fournit justement son type).

```
  | Var x -> Env.find x env
```

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

Dans le cas contraire, c'est la fonction `Env.find` qui lèvera une exception (en l'occurrence : `Not_found`). Une soustraction demande que chaque opérande soit cohérent, et du bon type.

```
  | Sub(e1, e2) -> let t1 = type_expr e1 env in
                   let t2 = type_expr e2 env in
                   if t1 = TInt && t2 = TInt then TInt
                   else failwith "type error"
```

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}}$$

Notez que ce code peut échouer à plusieurs endroits différents : pendant la vérification de `e1` ou `e2` si l'une ou l'autre de ces expressions n'est pas cohérente, ou explicitement avec la dernière ligne si `e1` et `e2` sont toutes deux cohérentes mais que l'une n'a pas le type attendu.

Dans le cas de l'accès à un tableau, il faut vérifier que l'expression de gauche a bien un type de tableau et que celle de droite a un type entier. Ces deux points donnent deux raisons distinctes d'échouer dans la vérification.

```
  | Get(e1, e2) -> let t1 = type_expr e1 env in
                   let t2 = type_expr e2 env in
                   (match t1, t2 with
                    | TArr t, TInt -> t
                    | _ -> failwith "type error")
```

$$\frac{\Gamma \vdash e_1 : \tau\texttt{[]} \qquad \Gamma \vdash e_1 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau}$$

Dans le cas d'un tableau, on doit vérifier que tous les éléments sont cohérents et du même type. Ici, on prend comme référence le type du premier élément et on vérifie que le type de chaque élément suivant correspond bien.

```
  | Arr(e::tl) ->
     let t = type_expr e env in
     if List.for_all (fun e' -> type_expr e' env = t) tl then TArr t
     else failwith "type error"
  | Arr([]) -> failwith "empty array"
```

$$\frac{\Gamma \vdash e_1 : \tau \quad \ldots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \texttt{[}e_1\texttt{, } \ldots\texttt{, } e_n\texttt{]} : \tau\texttt{[]}}$$

On traite à part le cas du tableau vide, dont on ne peut pas extraire un premier élément. On peut affirmer qu'il aurait un type de la forme `TArr t`, mais on n'a pas de manière simple de déterminer le bon « type de contenu » `t`. Il nous faudrait donc ici : soit restreindre cette manière de créer le tableau vide (comme en C), soit obliger la création d'un tableau à être explicitement accompagnée d'un type de contenu (comme en java), soit faire un peu d'inférence en regardant comment ce tableau est utilisé (comme en caml).

8.5 Raisonner sur les expressions typées

Un jugement de typage $\Gamma \vdash e : \tau$ est valide si et seulement s'il existe une dérivation construite avec les règles d'inférence qui a ce jugement pour conclusion. Ainsi, pour établir qu'une propriété P est vraie pour toutes les expressions bien typées, il suffit de démontrer que les règles d'inférence ne permettent pas de construire une dérivation dont la conclusion ne vérifie pas la propriété P. On procède en raisonnant *par induction sur la structure de la dérivation* d'un jugement $\Gamma \vdash e : \tau$: on a donc un cas par règle d'inférence, et chaque prémisse de la règle correspondante nous donne une hypothèse de récurrence.

Raisonnement par induction sur une dérivation de typage. Ainsi, pour démontrer qu'une certaine propriété $P(\Gamma, e, \tau)$ est vraie pour tout triplet (Γ, e, τ) tel que le jugement $\Gamma \vdash e : \tau$ est valide, il suffit de :

- montrer que $P(\Gamma, e, \tau)$ est valide pour tout triplet correspondant à un axiome, c'est-à-dire à l'une des règles s'appliquant aux constantes entières ou aux variables,
- montrer que, si $P(\Gamma, e_1, \mathsf{int})$ et $P(\Gamma, e_2, \mathsf{int})$ sont toutes deux valides, alors $P(\Gamma, e_1\texttt{-}e_2, \mathsf{int})$ est également valide,
- montrer que, si $P(\Gamma, e_1, \tau\texttt{[]})$ et $P(\Gamma, e_2, \mathsf{int})$ sont toutes deux valides, alors $P(\Gamma, e_1\texttt{[}e_2\texttt{]}, \tau)$ est également valide,
- montrer que, si les $P(\Gamma, e_i, \tau)$ sont valides pour une série d'expressions e_1 à e_k, alors $P(\Gamma, \texttt{[}e_1\texttt{, ..., }e_k\texttt{]}, \tau\texttt{[]})$ est encore valide.

Exemple : monotonie du typage par rapport à l'environnement. Par exemple, démontrons que l'on peut ajouter des éléments à l'environnement de typage sans perturber les expressions qui étaient déjà typables : si $\Gamma \vdash e : \tau$ est valide, et $x \notin \mathsf{dom}(\Gamma)$, alors $\Gamma, x : \sigma \vdash e : \tau$ est valide également. On pose $P(\Gamma, e, \tau)$ la propriété « si $x \notin \mathsf{dom}(\Gamma)$ et si σ est un type, alors $\Gamma, x : \sigma \vdash e : \tau$ ».

Notation : on note $\Gamma, x : \sigma$ l'environnement obtenu à partir de Γ en ajoutant une association de la variable x avec le type σ. Si on note Γ' cet environnement étendu, on a les équations $\begin{cases} \Gamma'(x) = \sigma \\ \Gamma'(y) = \Gamma(y) \quad y \neq x \end{cases}$

- Cas $(\Gamma, n, \mathsf{int})$: par la règle Int de typage des constantes entières, le jugement cible $\Gamma, x : \sigma \vdash n : \mathsf{int}$ est bien valide.
- Cas $(\Gamma, y, \Gamma(y))$: notons Γ' l'environnement étendu $\Gamma, x : \sigma$. Par la règle Var de typage des variables, on a $\Gamma' \vdash y : \Gamma'(y)$. Par hypothèse, on a $y \in \mathsf{dom}(\Gamma)$ et donc $x \neq y$. Ainsi $\Gamma'(y) = \Gamma(y)$, et on a donc bien $\Gamma, x : \sigma \vdash y : \Gamma(y)$.
- Cas $(\Gamma, e_1\texttt{-}e_2, \mathsf{int})$, en supposant $P(\Gamma, e_1, \mathsf{int})$ et $P(\Gamma, e_2, \mathsf{int})$ (hypothèses de récurrence). Les hypothèses de récurrence affirment que $\Gamma, x : \sigma \vdash e_1 : \mathsf{int}$ et $\Gamma, x : \sigma \vdash e_2 : \mathsf{int}$ sont valides. On combine ces deux jugements valides avec la règle Sub de typage de la soustraction pour obtenir $\Gamma, x : \sigma \vdash e_1\texttt{-}e_2 : \mathsf{int}$.
- Cas $(\Gamma, e_1\texttt{[}e_2\texttt{]}, \tau)$, avec les deux hypothèses de récurrence $\Gamma, x : \sigma \vdash e_1 : \tau\texttt{[]}$ et $\Gamma, x : \sigma \vdash e_2 : \mathsf{int}$: on obtient $\Gamma, x : \sigma \vdash e_1\texttt{[}e_2\texttt{]} : \tau$ par la règle Get de typage de l'accès à un tableau.
- Cas $(\Gamma, \texttt{[}e_1\texttt{, ..., }e_k\texttt{]}, \tau\texttt{[]})$, avec hypothèse de récurrence $\Gamma, x : \sigma \vdash e_i : \tau$ pour chacune des e_i : on conclut de même $\Gamma, x : \sigma \vdash \texttt{[}e_1\texttt{, ..., }e_k\texttt{]} : \tau\texttt{[]}$ directement par la règle Arr de typage des tableaux.

8.6 Approfondissement : sûreté des programme typés

La vérification des types déclenche une erreur lorsqu'elle détecte une incohérence dans le programme analysé. Ceci révèle en général une erreur du programmeur susceptible d'empêcher la bonne exécution du programme. À l'inverse, lorsqu'un programme est jugé cohérent lors de l'analyse des types, on s'attend à ce que son exécution se passe sans encombre. Sur l'exemple de nos expressions avec arithmétique et tableaux, nous allons voir qu'en raisonnant conjointement sur les règles de typage et sur la fonction d'évaluation il est possible d'établir un *théorème de sûreté* formalisant cela. Nous allons en particulier démontrer que l'évaluation d'une expression bien typée ne produit pas d'erreur `"unsupported␣operation"`.

Théorème de préservation du typage. Soient une expression e, un contexte de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$ soit valide. Soit ρ un environnement d'évaluation, nous souhaitons montrer que si e s'évalue en v dans l'environnement ρ alors cette valeur a également pour type τ.

Nous devons définir formellement ce que signifie pour une valeur le fait d'avoir le type τ. Même si nous utilisons la même syntaxe concrète pour représenter des valeurs et des programmes, ce sont des objets de nature différente. Par ailleurs une valeur dans notre langage ne comporte pas de variable, on peut donc lui attribuer un type sans introduire d'environnement de typage. Nous définissons un jugement $\vdash_v v : \tau$ entre une valeur v et un type τ avec deux règles d'inférence : une pour les valeurs entières et l'autre pour les valeurs tableau.

$$\frac{}{\vdash_v n : \texttt{int}}\ \textsc{VInt} \qquad \frac{\vdash_v e_1 : \tau \quad \dots \quad \vdash_v e_k : \tau}{\vdash_v [e_1, \dots, e_k] : \tau[]}\ \textsc{VArr}$$

Pour que la valeur du programme ait le type attendu, il faut que l'environnement d'évaluation associe à chaque variable, une valeur du type attendu. Nous pouvons maintenant énoncer le théorème de préservation du typage :

Soient une expression e, un contexte de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$ soit valide. Soit un environnement d'évaluation ρ compatible avec Γ, c'est-à-dire tel que $\text{dom}(\rho) = \text{dom}(\Gamma)$ et que pour tout $x \in \text{dom}(\rho)$, $\vdash_v \rho(x) : \Gamma(x)$ est valide. Alors, si v est une valeur telle que $\text{eval}(e, \rho) = v$, on a nécessairement $\vdash_v v : \tau$ valide.

Preuve de la préservation des types. On démontre le théorème de préservation du typage par induction sur la dérivation de $\Gamma \vdash e : \tau$.

- Cas $\Gamma \vdash n : \text{int}$. On a $\text{eval}(n, \rho) = n$, et on a bien $\vdash_v n : \text{int}$.
- Cas $\Gamma \vdash x : \Gamma(x)$. On a $\text{eval}(x, \rho) = \rho(x)$, et comme ρ est compatible avec Γ on sait que $\vdash_v \rho(x) : \Gamma(x)$. Ce qui est le résultat attendu.
- Cas $\Gamma \vdash e_1\text{-}e_2 : \text{int}$ avec $\Gamma \vdash e_1 : \text{int}$ et $\Gamma \vdash e_2 : \text{int}$. Par définition, si $\text{eval}(e_1\text{-}e_2, \rho) = v$ alors $v = n_1 - n_2$ avec $n_1 = \text{eval}(e_1, \rho)$ et $n_2 = \text{eval}(e_2, \rho)$. Autrement dit, v est une constante entière, qui vérifie bien $\Gamma \vdash v : \text{int}$.
- Cas $\Gamma \vdash [e_1, \dots, e_k] : \tau[]$, avec $\Gamma \vdash e_i : \tau$ pour tout i. Si $\text{eval}([e_1, \dots, e_k], \rho) = v$, alors par définition $v = [v_1, \dots, v_k]$, avec $v_i = \text{eval}(e_i, \rho)$ pour tout i. Or par hypothèse de récurrence, si $\text{eval}(e_i, \rho) = v_i$, alors $\vdash_v v_i : \tau$. Donc par règle de typage des valeurs tableaux $\vdash_v v : \tau[]$.
- Cas $\Gamma \vdash e_1[e_2] : \tau$ avec $\Gamma \vdash e_1 : \tau[]$ et $\Gamma \vdash e_2 : \text{int}$. Par définition, si $\text{eval}(e_1[e_2], \rho) = v$, alors $\text{eval}(e_1, \rho) = v_1 = [u_1, \dots, u_k]$ et $\text{eval}(e_2, \rho) = v_2 = n$ et $v = u_n$. Par hypothèse de récurrence, $\vdash_v [u_1, \dots, u_k] : \tau[]$. Or, seule la règle de typage des valeurs tableaux permet de dériver un tel jugement. Les prémisses $\vdash_v u_1 : \tau$, ..., $\vdash_v u_k : \tau$ sont donc nécessairement toutes dérivables. En particulier, $\vdash_v u_n : \tau$ est dérivable.

Théorème de sûreté. Soient une expression e, un contexte de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$ soit valide, et un environnement d'évaluation `env` compatible avec Γ, c'est-à-dire tel que $\text{dom}(\texttt{env}) = \text{dom}(\Gamma)$ et que pour tout $x \in \text{dom}(\texttt{env})$, $\vdash_v \texttt{Env.find}\ x\ \texttt{env} : \Gamma(x)$ est valide. Alors l'évaluation `eval e env` ne produit pas d'erreur `"unsupported operation"` ni d'erreur `Not_found`.

Dans ce théorème et sa preuve, on fait l'amalgame entre une expression e et la représentation caml de son arbre de syntaxe abstraite.

Preuve du théorème de sûreté. On démontre le théorème de sûreté des expressions bien typées par induction sur la dérivation de typage $\Gamma \vdash e : \tau$.

- Cas $\Gamma \vdash n : \text{int}$: on a `eval` n `env` = `VInt` n. L'évaluation n'échoue pas.
- Cas $\Gamma \vdash x : \Gamma(x)$: on a `eval` x `env` = `Env.find` x `env`. Il n'y a donc pas d'erreur `"unsupported operation"`. En outre, $x \in \text{dom}(\Gamma)$, et donc par hypothèse $x \in \text{dom}(\texttt{env})$, dont l'appel à `Env.find` ne produit pas non plus d'erreur `Not_found`.
- Cas $\Gamma \vdash e_1\text{-}e_2 : \text{int}$ avec $\Gamma \vdash e_1 : \text{int}$ et $\Gamma \vdash e_2 : \text{int}$. Par hypothèses de récurrence, `eval` e_1 `env` et `eval` e_2 `env` ne produisent pas d'erreur `"unsupported operation"` ni `Not_found`. Supposons que `eval` e_1 `env` = v_1. Alors par théorème de préservation du typage $\vdash_v v_1 : \text{int}$, et nécessairement v_1 est une constante entière `VInt` n_1. Supposons que `eval` e_2 `env` = v_2. Alors de même $\vdash_v v_2 : \text{int}$ et nécessairement v_2 est une constante entière `VInt` n_2. Donc par définition, `eval (Sub(`e_1`,`e_2`)) env` = `VInt (`n_1`-`n_2`)`. Cette évaluation ne produit pas d'erreur `"unsupported operation"` ni `Not_found`.
- Cas $\Gamma \vdash [e_1, \dots, e_k] : \tau[]$, avec $\Gamma \vdash e_i : \tau$ pour tout i. Par hypothèse de récurrence, `eval` e_i `env` ne produit pas d'erreur `"unsupported operation"` ni `Not_found`, pour aucun

i. Donc `eval (Arr(`e_1`, ..., `e_k`)) env` ne produit pas d'erreur `"unsupported␣operation"` ni `Not_found`.

— Cas $\Gamma \vdash e_1[e_2] : \tau$ avec $\Gamma \vdash e_1 : \tau[]$ et $\Gamma \vdash e_2 :$ int. Par hypothèse de récurrence, eval(e_1, ρ) ne produit pas d'erreur `"unsupported␣operation"` ni `Not_found`. Supposons que `eval` e_1 `env` $= v_1$, alors par théorème de préservation du typage on a $\vdash_v v_1 : \tau[]$, et nécessairement v_1 a la forme `VArr` a. Par hypothèse de récurrence, `eval` e_2 `env` ne produit pas d'erreur `"unsupported␣operation"` ni `Not_found`. Supposons `eval` e_2 `env` $= v_2$, alors par théorème de préservation du typage on a $\vdash_v v_2 :$ int, et nécessairement v_2 est une constante entière `VInt` i. Ainsi, `eval (Get(`e_1`, `e_2`)) env` ne produit pas d'erreur `"unsupported␣operation"` ni `Not_found`.

Notez qu'il reste deux situations dans lesquelles l'évaluation n'aboutit pas à une valeur, malgré le bon typage : le cas où l'évaluation ne termine pas (impossible ici, mais deviendra réel avec un langage plus riche), et le cas d'une erreur d'une autre nature (accès en dehors des bornes d'un tableau, division par zéro...).

8.7 Approfondissement : le langage IMPScript

IMPScript est un langage créé pour les besoins de ce cours, qu'on reverra dans les prochains chapitres.

Pour conclure ce chapitre, on considère le langage IMPScript, qui peut être vu comme une extension du noyau impératif de IMP avec des fonctions et des tableaux, le tout dans une syntaxe compatible avec JavaScript.

```
let a = [3, 1, 7, 4, 2, 6, 5];

function binary_search_right(v, a, lo, hi) {
  while (lo < hi) {
    let mid = lo + ((hi-lo)>>1);
    if (a[mid] <= v) { lo = mid+1; }
    else             { hi = mid; }
  }
  return hi;
}

function binary_sort(a, n) {
  for (let i=1; i<n; i=i+1) {
    let v = a[i];
    let m = binary_search_right(v, a, 0, i);
    for (let j=i; j>m; j=j-1) { a[j] = a[j-1]; }
    a[m] = v;
  }
}

binary_sort(a, 7);
```

Syntaxe abstraite. On peut définir les arbres de syntaxe des expressions et instructions IMPScript en utilisant des types algébriques, comme on l'avait déjà fait pour IMP. En se concentrant sur les formes visibles dans notre exemple, on aurait donc au moins les constructions suivantes pour les expressions.

```
type binop = Add (* + *) | Sub (* - *)  | Asr (* >> *)
           | Lt  (* < *) | Le  (* <= *) | Gt  (* > *)
type expr =
  | Int    of int                    (*  42             *)
  | Var    of string                 (*  x              *)
  | Binop  of binop * expr * expr    (*  e1 + e2        *)
  | Call   of expr * expr list       (*  f(e1, ..., eN) *)
  | ArrGet of expr * expr            (*  e1[e2]         *)
  | Array  of expr list              (*  [e1, ..., eN]  *)
```

Parmi les nouvelles formes on voit :

— l'appel de fonction représenté par le constructeur `Call`, avec une première expression désignant la fonction et une liste d'expressions passées en paramètres,

— l'accès à un élément d'un tableau représenté par `ArrGet`, construit avec une expression désignant un tableau et une désignant un indice,

— la manipulation directe d'un tableau représentée par `Array` et une liste d'éléments.

Pour les instructions :

```
type instr =
  | Set    of string * expr        (*  x = x+1;                    *)
  | If     of expr * seq * seq     (*  if (e) { s1 } else { s2 }   *)
  | While  of expr * seq           (*  while (e) { s }             *)
  | Return of expr                 (*  return home;                *)
  | Expr   of expr                 (*  f(3);                       *)
  | ArrSet of expr * expr * expr   (*  a[0] = 27                    *)
and seq = instr list
```

Parmi les nouvelles formes on voit :

— l'instruction `Return`, qui termine l'exécution d'une fonction en renvoyant une valeur,
— l'écriture dans une case d'un tableau représentée par `ArrSet`,
— la possibilité d'utiliser une expression comme une instruction (courant par exemple pour un appel de fonction qui ne renvoie pas de résultat).

Notez en revanche l'absence de **`for`**. Cette boucle peut être codée à l'aide de **`while`**.

Représentation des programmes typés. Nous allons décrire un vérificateur de types pour IMPScript. Pour cela, nous allons inclure des informations de types dans les structures de données représentant les fonctions et les programmes IMPScript. On se donne déjà un type de données pour représenter les types eux-mêmes.

```
type typ =
  | TInt                      (*  int                *)
  | TBool                     (*  bool               *)
  | TArray of typ             (*  t[]                *)
  | TFun   of typ list * typ  (*  (t1, ..., tn) -> t  *)
  | TVoid                     (*  absence de valeur   *)
```

En plus des éléments déjà vus, il contient des types pour les booléens et pour les fonctions. Pour simplifier, on ajoute également un élément `TVoid`, qui *n'est pas* un type, mais qui permet d'indiquer simplement qu'une certaine fonction ne renvoie pas de résultat.

La définition d'une fonction indique le nom de la fonction, la liste de ses paramètres et de ses variables locales, ainsi que son *corps*, c'est-à-dire la séquence d'instructions à exécuter. La structure contient également des informations de types pour chaque paramètre et chaque variable locale, ainsi qu'un type de retour (qui pourra être la valeur spéciale `TVoid` pour indiquer l'absence de valeur de retour).

```
type function_def = {
    name:   string;
    params: (string * typ) list;
    locals: (string * typ) list;
    body:   seq;
    return: typ;
  }
```

Un programme est constitué d'un ensemble de variables globales, d'un ensemble de définitions de fonctions, et d'une séquence d'instructions principale. Comme dans les fonctions, chaque variable est accompagnée d'une information de type.

```
type program = {
    globals:   (string * typ) list;
    functions: function_def list;
    code:      seq;
  }
```

Vérificateur de types. Pour donner un minimum d'information sur les erreurs détectées, on prévoit des messages indiquant le type de l'expression fautive et le type qu'elle aurait du avoir.

Pour ajouter une localisation, il faudrait mémoriser les numéros de ligne dans l'AST.

```
exception Type_error of string
let error s = raise (Type_error s)
let type_error ty_actual ty_expected =
  error (Printf.sprintf "expected␣%s,␣got␣%s"
           (typ_to_string ty_expected) (typ_to_string ty_actual))
```

L'environnement de typage va associer chaque nom de variable et chaque nom de fonction à un type. Lors de l'analyse d'un programme (fonction principale `typecheck_prog`), on initialise un tel environnement avec les types de toutes les variables globales et toutes les fonctions. La fonction auxiliaire `add_env` permet d'ajouter en bloc une liste de telles associations. On utilise cet environnement pour analyser le code du programme lui-même et de chacune de ses fonctions, à l'aide des fonctions `typecheck_fdef` et `typecheck_code` décrites plus bas.

```
module Env = Map.Make(String)
type tenv = typ Env.t

let add_env l tenv =
  List.fold_left (fun env (x, t) -> Env.add x t env) tenv l

let typecheck_prog p =
  let f_types = List.map
    (fun f -> f.name, TFun(List.map snd f.params, f.return))
    p.functions
  in
  let tenv = add_env p.globals Env.empty in
  let tenv = add_env f_types tenv in
  List.iter (fun f -> typecheck_fdef f tenv) p.functions;
  typecheck_code p.code TVoid tenv
```

La fonction `typecheck_code` réalise l'essentiel du travail. Elle servira à la fois pour le code principal du programme et pour le corps de chaque fonction. Elle prend en entrée une séquence d'instructions, un éventuel type de retour attendu (`TVoid` s'il n'y a pas de retour attendu) et un environnement de typage, et déclenche une erreur si elle rencontre une incohérence. L'analyse elle-même est réalisée par quatre fonctions internes :

- `check` prend en paramètres une expression e et un type attendu τ, et vérifie que e a bien le type τ (ou déclenche une erreur),
- `type_expr` calcule le type d'une expression, ou déclenche une erreur si l'expression prise en argument n'est pas bien typée,
- `check_instr` et `check_seq` vérifient le bon typage des instructions et séquences d'instructions.

Notez que ces quatre fonctions ont implicitement accès à l'environnement de typage et au type de retour éventuellement attendu, et que seule `type_expr` renvoie un résultat significatif.

```
  let typecheck_code s ret tenv =
```

Les fonctions `check` et `type_expr` fonctionnent ensemble : l'une calcule le type d'une expression et l'autre vérifie que cette expression est utilisée à bon escient. Le type des variables est récupéré dans l'environnement, et la vérification de typage d'un appel de fonction compare le type de chaque argument au type attendu par la fonction, de même que la vérification de typage d'une opération arithmétique vérifie le type de chaque opérande.

```
  let rec check e ty_expected =
    let te = type_expr e in
    if te <> ty_expected then type_error te ty_expected

  and type_expr = function
    | Int _  -> TInt
    | Var x  -> Env.find x tenv
    | Binop((Add | Sub | Asr), e1, e2) ->
       check e1 TInt; check e2 TInt; TInt
    | Binop((Lt  | Le  | Gt ), e1, e2) ->
       check e1 TInt; check e2 TInt; TBool
    | Call(f, args) ->
      (match type_expr f with
         | TFun(targs, tret) -> List.iter2 check args targs; tret
         | t -> error ("expected␣a␣function,␣got␣" ^ typ_to_string t)
    | ArrGet(e1, e2) ->
      (check e2 TInt;
       match type_expr e1 with
         | TArray t -> t
         | t -> error ("expected␣an␣array,␣got␣" ^ typ_to_string t))
    | Array([]) -> TArray(TVoid)
```

```
    | Array(e::elts) ->
       let t = type_expr e in
       List.iter (fun elt -> check elt t) elts;
       TArray(t)
  in
```

Notez que cette fois, on a accepté de donner un type au tableau vide, mais en forçant le type factice `TVoid` pour le contenu, qui ne pourra donc pas être utilisé.

Les fonctions de vérification du bon typage des instructions et des séquences sont mutuellement récursives, en plus de faire appel aux deux précédentes. En cas d'instruction `return`, on vérifie d'une part qu'un retour était bien attendu et d'autre part, le cas échéant, que la valeur renvoyée est du bon type.

```
  let rec check_instr = function
    | Set(x, e)     -> let t = Env.find x tenv in check e t
    | If(e, s1, s2) -> check e TBool; check_seq s1; check_seq s2
    | While(e, s)   -> check e TBool; check_seq s
    | Return e      -> if ret = TVoid then error "unexpected␣return"
                                      else check e ret
    | Expr e        -> check e TVoid
    | ArrSet(e1, e2, e3) ->
      (check e2 TInt;
       match type_expr e1 with
         | TArray t -> check e3 t
         | t -> error ("expected␣an␣array,␣got␣" ^ typ_to_string t))
  and check_seq s =
    List.iter check_instr s
  in
  check_seq s
```

La vérification de bon typage d'une définition de fonction consiste essentiellement à appeler la fonction `typecheck_code` précédente sur le corps de la fonction, après avoir ajouté à l'environnement de typage le contexte local de cette fonction (paramètres et variables locales). On y ajoute un ultime test pour s'assurer que, si la fonction est censée renvoyer un résultat, alors il est bien certain qu'elle exécutera une instruction `return` (fonction `return_seq` définie ci-dessous).

```
let typecheck_fdef f tenv =
  let tenv = add_env f.params tenv in
  let tenv = add_env f.locals tenv in
  typecheck_code f.body f.return tenv;
  if f.return <> TVoid && not (return_seq f.body) then
    error "function␣might␣not␣return␣a␣value"
```

Les fonctions `return_instr` et `return_seq` ont la propriété suivante : si l'instruction/la séquence prise en argument peut être exécutée intégralement sans passer par une instruction `return`, alors la fonction renvoie **`false`**. À l'inverse, ces fonctions renvoient **`true`** lorsque la seule forme de l'instruction/de la séquence garantit qu'un `return` sera bien rencontré avant la fin de l'exécution.

```
let rec return_instr = function
  | Set _         -> false
  | If(_, s1, s2) -> return_seq s1 && return_seq s2
  | While _       -> false
  | Return _      -> true
  | Expr _        -> false
  | ArrSet _      -> false
and return_seq = function
  | []   -> false
  | i::s -> return_instr i || return_seq s
```

À noter : l'analyse faite par `return_seq`/`return_instr` est incomplète. On peut trouver une instruction i qui exécute à coup sûr un `return` mais telle que `return_instr` i renvoie **`false`**. En revanche, s'il y a la possibilité de ne pas passer par un `return`, alors la fonction renvoie **`false`** à coup sûr.

Trouvez-vous une telle instruction ?

9 Fonctions et pile d'appels

Un point clé du développement de programmes réalistes est la possibilité de définir et d'appeler des fonctions. Du point de vue de la compilation, cela demande un peu de technologie supplémentaire. Fixons d'abord le vocabulaire, en observant l'***appel de fonction*** `f(6)` dans le code suivant.

```
1   function g() {
2     let y;
3     y = f(6);
4     print(y);
5   }

6   function f(x) {
7     return x*7;
8   }
```

L'expression `6` est un ***paramètre effectif*** (on dit aussi un *argument*) de l'appel de fonction `f(6)`. Le bloc de code allant des lignes 2 à 4 est le ***contexte appelant***, et la fonction `f` définie aux lignes 6 à 8 est la ***fonction appelée***. Du côté de la fonction appelée, la variable `x` est un ***paramètre formel*** de la fonction, et l'expression `x*7` calcule le résultat ***renvoyé*** par la fonction.

Certains utilisent *retourner* à la place de *renvoyer*, mais il s'agit d'un faux ami.

9.1 Appel de fonction : mécanisme de base

Un appel de fonction implique des transferts de différentes natures entre le contexte appelant et la fonction appelée.

Articulation appelant/appelé. On a d'une part un transfert de données, avec
- un ou plusieurs paramètres effectifs transmis par le contexte appelant à la fonction appelée,
- un résultat, renvoyé au contexte appelant par la fonction appelée.

Pour cela, on peut utiliser la pile et/ou les registres, d'une manière à convenir. Outre ce transfert de données, on a un transert *temporaire* de contrôle :
- lors de l'appel, l'exécution *saute* au code de la fonction appelée,
- à la fin de l'appel, l'exécution *revient* au contexte appelant, en reprenant « juste après » le point où l'on a réalisé l'appel.

Ce transfert temporaire demande, au moment de l'appel, de mémoriser l'adresse de l'instruction à laquelle il faudra revenir après l'appel. En MIPS, on utilise pour cela le registre `$ra` (*Return Address*). Pour sauter au code de la fonction tout en mémorisant dans `$ra` l'adresse de l'instruction suivante du contexte appelant, on utilise l'instruction `jal` (*Jump And Link*) avec l'étiquette de la fonction appelée, ou sa variante `jalr` lorsque l'adresse de la fonction est donnée par un registre (qui est éventuellement le résultat d'un calcul).

```
  jal   f
  jalr  $t0
```

À la fin de l'appel de fonction, on rend alors la main à l'appelant avec un saut à l'adresse donnée par `$ra`.

```
  jr    $ra
```

Pour obtenir des adresses réalistes en MIPS, il faut ajouter à chacune la constante `0x400000`, qui est la première adresse de la région de la mémoire dédiée au programme.

Exemple. Voici une traduction possible du fragment de code précédent, où les nombres à gauche donnent l'adresse de chaque instruction. On décide ici que l'unique argument de `f` est placé dans le registre `$a0` et que son résultat est placé dans le registre `$v0`.

```
12  g:  li   $a0, 6          # définition du paramètre effectif
16      jal  f               # appel de fonction
20      move $a0, $v0        # affichage du résultat
24      li   $v0, 1
28      syscall

36  f:  li   $v0, 7          # calcul
40      mul  $v0, $a0, $v0
44      jr   $ra             # fin de la fonction
```

Voici une trace d'exécution de ce code assembleur. On y observe en particulier que l'exécution du saut `jal` rencontré à l'adresse `16` stocke dans `$ra` l'adresse suivante `20` (tout en sautant à l'adresse `36`).

instruction	$a0	$v0	$ra	pc	
				12	
`li $a0, 6`	6			16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42

Problème : appels imbriqués. Observons maintenant ce qui se passe si on ajoute un code principal réalisant un appel à notre fonction g. On complète le code comme suit.

```
00  main:  jal  g           # appel à g (sans argument)
04         li   $v0, 10     # fin du programme principal
08         syscall
12  g:     li   $a0, 6
16         jal  f
20         move $a0, $v0
24         li   $v0, 1
28         syscall
32         jr   $ra         # fin de g
36  f:     li   $v0, 7
40         mul  $v0, $a0, $v0
44         jr   $ra
```

La trace d'exécution est maintenant la suivante.

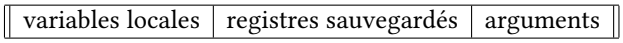
instruction	$a0	$v0	$ra	pc	
				0	
`jal g`			4	12	
`li $a0, 6`	6		4	16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42
`jr $ra`	42	1	20	20	
`move $a0, $v0`	1	1	20	24	
`li $v0, 1`	1	1	20	28	
`syscall`	1	1	20	32	affiche 1
`jr $ra`	1	1	20	20	
`move $a0, $v0`	1	1	20	24	
…	…	…	…	…	

Nous voilà coincés dans une boucle ! Problème : lorsque l'instruction `jal f` d'appel à `f` stocke son adresse de retour dans `$ra`, elle écrase la valeur qui était déjà présente dans `$ra`, à savoir l'adresse de retour de l'appel à g. Ainsi, lorsque l'on arrive à l'instruction `jr $ra` qui achève le code de g à l'adresse 32, le registre `$ra` ne contient plus la valeur qui avait été sauvegardée au moment de l'appel à g, et notre saut se fait vers la mauvaise cible.

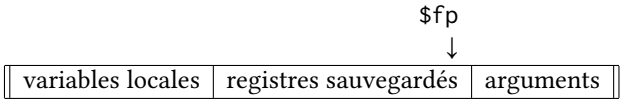
9.2 Pile d'appels

Lorsque des appels de fonction sont emboîtés, c'est-à-dire lorsque le code d'une fonction appelée réalise lui-même un appel de fonction et ainsi de suite, on se retrouve avec plusieurs appels actifs à un même moment. Plus précisément, l'appel le plus récent est en cours d'exécution et un certain nombre d'autres appels plus anciens sont en suspens.

Tableaux d'activation. Chaque appel de fonction actif possède son propre contexte, contenant notamment les arguments qui lui ont été passés, les valeurs de ses variables locales, ou encore l'adresse de retour stockée dans `$ra`. Toutes ces informations doivent être stockées de manière à survivre à des appels de fonction internes. La structure utilisée pour stocker les informations d'un appel donné est appelée ***tableau d'activation*** (en anglais *call frame*). On considère ici des tableaux d'activation avec la forme générale suivante, où la partie « registres sauvegardés » contient en particulier la valeur sauvegardée du registre `$ra`.



Ces tableaux sont stockés dans la mémoire. On utilise un registre dédié `$fp` (*Frame Pointer*) pour localiser le tableau actif, c'est-à-dire le tableau d'activation de l'appel actuellement en cours d'exécution. On peut ensuite, à partir de `$fp`, accéder aux différents éléments stockés dans le tableau. Le pointeur de base `$fp` désigne l'adresse du dernier mot dédié aux registres sauvegardés.

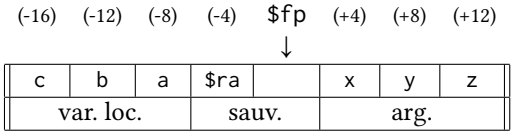


On accède donc aux arguments à partir de `$fp` avec un décalage positif, et aux variables locales avec un décalage négatif.

Exemple. En supposant une fonction avec trois paramètres `x`, `y` et `z` et trois variables locales `a`, `b` et `c`

```
function h(x, y, z) {
  let a, b, c;
  ...
```

on pourrait avoir un tableau d'activation de la forme suivante, où la valeur de `$ra` est sauvegardée à l'adresse `fp-4`, où les valeurs des paramètres effectifs sont stockées à partir de `fp+4` (vers les adresses supérieures), et les variables locales sont stockées à partir de `fp-8` (vers les adresses inférieures).

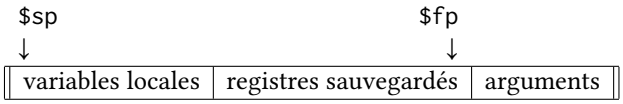


On laisse libre pour l'instant la case à l'adresse `fp`, qui servira à un autre registre sauvegardé.

Pile d'appels. La vie des tableaux d'activation se déroule ainsi :
- création d'un tableau d'activation au début d'un appel,
- destruction du tableau à la fin de l'appel.

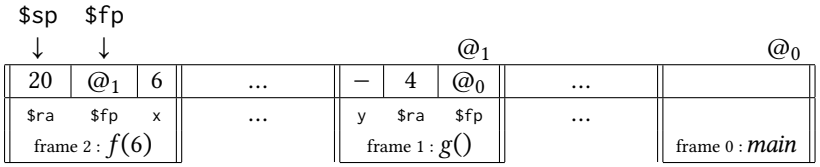
En outre, tout appel doit se terminer avant de rendre la main à son contexte appelant. Autrement dit, c'est toujours l'appel le plus récent qui terminera en premier, et toujours le tableau d'activation le plus récent qui sera détruit en premier. L'ensemble des tableaux d'activation a donc une structure de pile (*Last In First Out*), nommée ***pile d'appels*** (*call stack*). La pile d'appels est réalisée dans la pile MIPS elle-même, et en forme l'ossature principale.

Ainsi le premier mot du tableau d'activation correspond au sommet de la pile, désigné par le pointeur `$sp` (*Stack Pointer*).



En outre, pour faciliter l'identification du tableau d'activation précédent (qui est celui du contexte appelant), on peut ajouter dans tout tableau d'activation une sauvegarde de la valeur précédente de `$fp`, c'est-à-dire de l'adresse de base du tableau d'activation précédent. On obtient ainsi un chaînage des tableaux d'activation, allant du contexte courant vers les contextes plus anciens, jusqu'à l'appel de la fonction principale de notre programme.

Exemple. On reprend les fonctions `f` et `g` prises en introduction. Pendant l'appel `f(6)`, la pile contient trois tableaux d'activation : un pour le contexte principal (*main*) que l'on ne détaille pas, un pour l'appel `g()`, qui contient notamment un emplacement non initialisé pour la variable locale `y`, et un pour l'appel `f(6)`, qui contient notamment un emplacement pour la valeur du paramètre `x`. Le pointeur `$fp` courant pointe dans ce dernier tableau, qui est au sommet de la pile. Les tableaux de `f(6)` et `g()` contiennent une adresse `$ra` sauvegardée qui fait référence aux numéros de l'extrait de code assembleur p.125. On désigne dans ce tableau par $@_0$ et $@_1$ respectivement l'adresse de base du tableau d'activation principal et du tableau d'activation de `g()` (il s'agit des valeurs de `$fp` pour ces tableaux).



9.3 Convention d'appel

Pour assurer les bons transferts d'information et de contrôle, et la bonne gestion de la pile d'appels, le contexte appelant et la fonction appelée doivent respecter un protocole commun appelé ***convention d'appel***.

« Convention » : choix d'un cadre commun, qui ne préjuge pas de l'existence d'alternatives viables.

Voir approfondissements pour la convention MIPS standard, qui est plus riche.

Protocole minimal. Un protocole d'appel se découpe en plusieurs étapes, à la charge de l'un ou l'autre des participants. On propose ici une version simple, que l'on utilisera pour compiler ImpScript.

1. **Appelant, avant l'appel :** évalue les arguments et place les valeurs obtenues sur la pile, puis déclenche l'appel avec `jal` ou `jalr` pour stocker l'adresse de retour dans `$ra` tout en passant la main à l'appelé.
2. **Appelé, au début de l'appel :** sauvegarde les valeurs de `$fp` et `$ra` sur la pile, réserve de l'espace supplémentaire pour ses variables locales, et donne à `$fp` sa nouvelle valeur (ceci complète la création du tableau d'activation).
3. **Appelé :** exécution du corps de la fonction appelée.
4. **Appelé, à la fin de l'appel :** place le résultat renvoyé dans le registre `$t0`, restaure `$fp` et `$ra`, libère l'espace de pile réservé à l'étape 2, et redonne la main à l'appelant avec `jr $ra`.
5. **Appelant, après l'appel :** retire les arguments placés sur la pile.

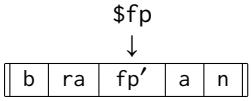
Le tableau d'activation est donc créé aux étapes 1 et 2 et détruit aux étapes 4 et 5 : il existe uniquement pendant l'exécution du corps de la fonction appelée. À la fin du protocole, la pile a retrouvé son état d'origine, et le résultat de l'appel est dans le registre `$t0`, prêt à être utilisé.

Exemple. On considère le programme ImpScript suivant, qui définit et utilise une fonction d'exponentiation rapide.

```
function power(a, n) {
  if (n == 0) { return 1; }
  else {
    let b = power(a*a, n>>1);
    if (n&1 == 0) { return b; }
    else { return a*b; }
  }
}

console.log(power(2, 9));
```

La fonction `power` a deux arguments `a` et `n`, et une variable locale `b`. Un tableau d'activation pour un appel de cette fonction a donc la forme suivante.



On accède aux arguments `a` et `n` respectivement aux adresses `fp+4` et `fp+8`, et à la variable locale `b` à l'adresse `fp-8`.

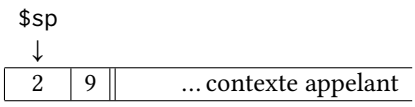
Protocole d'appel à `power`, ordre chronologique. Pour réaliser l'appel `power(2, 9)`, il faut d'abord placer les deux arguments au sommet de la pile,

```
li      $t0, 9              # chargement de 9
addi    $sp, $sp, -4        # passage sur la pile
sw      $t0, 0($sp)
li      $t0, 2              # chargement de 2
addi    $sp, $sp, -4        # passage sur la pile
sw      $t0, 0($sp)
```

puis appeler la fonction,

```
jal     power
```

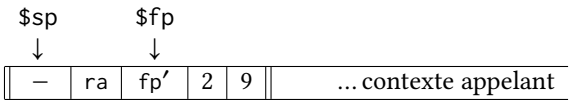
La fonction `power` prend alors le relais. La pile contient déjà les deux arguments



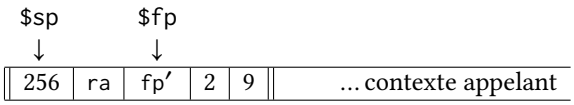
et il faut maintenant construire le reste du tableau d'activation. Pour cela, on déplace le pointeur `$sp`, on sauvegarde les valeurs des registres `$fp` et `$ra`, et on donne sa nouvelle valeur à `$fp`.

```
addi    $sp, $sp, -12       # allocation
sw      $fp, 8($sp)         # sauvegarde fp
sw      $ra, 4($sp)         # sauvegarde ra
addi    $fp, $sp, 8         # définition fp
```

Avec un déplacement de 12 octets (3 mots mémoire), on prévoit la place pour la variable `b` en plus des deux registres sauvegardés.



Le corps de la fonction peut alors s'exécuter. Après divers événements, dont des appels récursifs, nous arrivons à la fin de cette exécution avec l'instruction `return a*b;`, alors que la variable locale `b` a reçu la valeur 256 (résultat de `power(4, 4)`).



On peut alors calculer le résultat à renvoyer `a*b` (512) et le placer dans le registre `$t0`,

```
lw      $t0, 4($fp)         # t0 <- a
lw      $t1, -8($fp)        # t1 <- b
mul     $t0, $t0, $t1       # t0 <- a*b
```

restaurer les registres `$ra` et `$fp` et libérer l'espace correspondant sur la pile,

```
lw      $ra, 4($sp)         # restauration ra
lw      $fp, 8($sp)         # restauration fp
addi    $sp, $sp, 12        # nettoyage de la pile
```

et enfin rendre la main à l'appelant.

```
jr      $ra
```

Quand l'appelant reprend la main, la pile a donc retrouvé l'état qu'elle avait juste avant l'appel, avec les arguments encore en place. Il suffit de déplacer à nouveau `$sp` pour les retirer et conclure le protocole d'appel.

```
addi    $sp, $sp, 8         # nettoyage de la pile
```

Enfin, l'appelant peut maintenant utiliser le résultat de l'appel, qui est disponible dans le registre `$t0`.

```
move    $a0, $t0
li      $v0, 1
syscall                     # affichage
```

Code de l'exponentiation, dans l'ordre. Le protocole que nous venons de décrire permet de gérer un nombre arbitraire d'appels de fonction imbriqués. En particulier, il s'applique même aux appels récursifs de notre fonction `power`, dont on peut maintenant voir le code complet. Le code assembleur généré pour la fonction `power` commence à s'exécuter lorsque les arguments sont déjà placés sur la pile. Il commence par l'étape 2 du protocole d'appel (*appelé, au début de l'appel*), pour finir de mettre en place le tableau d'activation.

```
    power:
    # Construction du tableau d'activation (ÉTAPE 2)
052     addi    $sp, $sp, -12
056     sw      $fp, 8($sp)
060     sw      $ra, 4($sp)
064     addi    $fp, $sp, 8
```

Cette initialisation passée, on peut enchaîner avec le code correspondant au corps de la fonction (étape 3).

```
    # Exécution du corps de la fonction (ÉTAPE 3)
068     lw      $t0, 8($fp)     # test (n == 0)
072     bnez    $t0, power_rec
076     li      $t0, 1          # t0 <- résultat (return 1)
080     b       power_end
    power_rec:
```

On voit réapparaître l'étape 1 du protocole (*appelant, avant l'appel*) au moment où la fonction `power` s'apprête elle-même à réaliser un appel de fonction.

Ici, un appel récursif, mais cela ne change rien.

```
    # Déclenchement de l'appel récursif (ÉTAPE 1)
084     lw      $t0, 8($fp)     # push argument (n>>1)
088     sra     $t0, $t0, 1
092     addi    $sp, $sp, -4
096     sw      $t0, 0($sp)
100     lw      $t0, 4($fp)     # push argument (a*a)
104     mul     $t0, $t0, $t0
108     addi    $sp, $sp, -4
112     sw      $t0, 0($sp)
116     jal     power           # appel
```

Note : dans ce fragment une instruction est inutile, car place dans un registre une valeur qui s'y trouve déjà. La voyez-vous ?

Dans le code assembleur, l'étape 5 (*appelant, après l'appel*) apparaît immédiatement après l'instruction de saut, puisque c'est précisément là que l'exécution reviendra quand l'appelé rendra la main.

```
    # Après l'appel récursif (ÉTAPE 5)
120     addi    $sp, $sp, 8     # nettoyage arguments
```

Une fois le protocole d'appel terminé, la fonction (appelante) peut récupérer le résultat de l'appel et reprendre son travail.

```
    # Retour à l'exécution du corps de la fonction (ÉTAPE 3, suite)
124     sw      $t0, -8($fp)    # b <- résultat power(a*a, n>>1)
128     lw      $t0, 8($fp)     # test (n&1 == 0)
132     andi    $t0, $t0, 0x1
136     bnez    $t0, power_odd
140     lw      $t0, -8($fp)    # t0 <- résultat (return b)
144     b       power_end
    power_odd:
148     lw      $t0, 4($fp)
152     lw      $t1, -8($fp)
156     mul     $t0, $t0, $t1   # t0 <- résultat (return a*b)
```

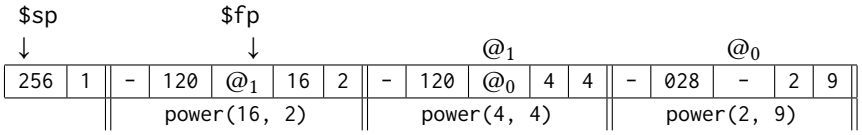
Enfin, le code généré termine par l'étape 4 du protocole (*appelé, à la fin de l'appel*), pour rendre la main au contexte appelant après avoir nettoyé la pile.

```
    power_end:
    # Destruction du tableau d'activation, fin de l'appel (ÉTAPE 4)
160     lw      $ra, 4($sp)
164     lw      $fp, 8($sp)
168     addi    $sp, $sp, 12
172     jr      $ra
```

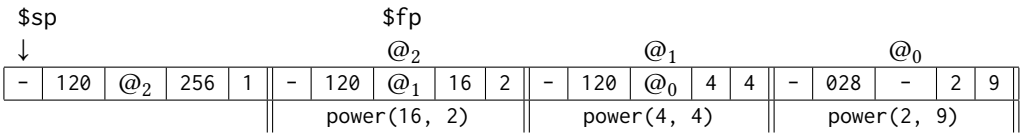
Suivi de la pile d'appels. Pour mémoire, voici également le code complet de l'instruction principale du programme (`console.log(power(2, 9))`).

```
    # Appel power(2, 9)
000     li      $t0, 9          # préparation des arguments
004     addi    $sp, $sp, -4
008     sw      $t0, 0($sp)
012     li      $t0, 2
016     addi    $sp, $sp, -4
020     sw      $t0, 0($sp)
024     jal     power           # appel
028     addi    $sp, $sp, 8     # nettoyage des arguments
    # Affichage du résultat
032     move    $a0, $t0
036     li      $v0, 1          # code 1 : affichage d'un entier
040     syscall
    # Fin du programme
044     li      $v0, 10         # code 10 : arrêt de l'exécution
048     syscall
```

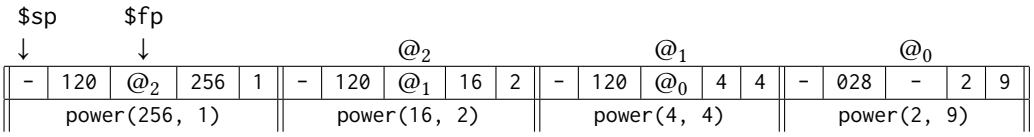
L'appel `power(2, 9)` réalisé ligne `024` va engendrer un appel récursif `power(4, 4)` (ligne `116`), qui engendre un appel `power(16, 2)` (ligne `116` encore), qui lui-même engendre un appel `power(256, 1)` (ligne `116` toujours). Plaçons-nous à la ligne `052`, au début de ce dernier appel. La pile d'appels a alors la forme suivante, avec les deux arguments `256` et `1` au sommet de la pile, et les tableaux d'activation des trois appels précédents. À ce stade, le pointeur de base `$fp` courant est celui de notre contexte appelant (l'appel `power(16, 2)`. On note $@_1$ et $@_0$ les adresses de base des autres tableaux d'activation présents. Dans chaque tableau, la case devant accueillir la variable locale `b` est présente, mais pas encore initialisée.



Les lignes `052` à `060` déplacent le sommet `$sp` de la pile et enregistrent les valeurs courantes de `$fp` et `$ra` dans cet espace ajouté à la pile. On y note également $@_2$ l'adresse de base du contexte appelant qui, pour l'instant, est encore donnée par `$fp`.



Puis la ligne `064` complète l'initialisation du tableau d'activation en définissant la nouvelle valeur du pointeur de base `$fp`. À nouveau, la case devant accueillir la variable locale `b` est déjà présente mais pas encore initialisée.



À la fin de l'appel, les lignes `160` à `168` effectuent les opérations inverses, et nous ramènent à la situation du premier schéma.

9.4 Approfondissement : génération de code pour ImpScript

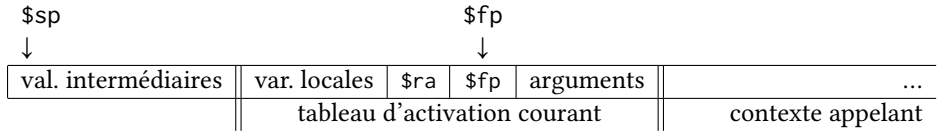
Voyons maintenant comment ce protocole d'appel basé sur la pile peut engendrer une traduction générale d'un programme ImpScript vers du code assembleur MIPS. Pour cette section, on prend le langage ImpScript tel que présenté au chapitre précédent, moins les tableaux qui nécessiteront un traitement particulier (voir prochain chapitre).

On fixe les éléments suivants pour cette traduction :

- la valeur calculée pour une expression est placée dans le registre `$t0`,
- chaque valeur intermédiaire du calcul d'une expression est stockée sur la pile,
- les valeurs des arguments d'une fonction sont passées sur la pile,

ImpScript passe les paramètres exclusivement « par valeur » (comme caml, python, java).

Sur la pile, les tableaux d'activation des différents appels en cours sont donc entremêlés avec des séries de valeurs intermédiaires des expressions en cours d'évaluation.



On s'intéresse à la traduction d'une définition de fonction ImpScript en code assembleur MIPS. Voici la structure représentant une telle définition :

```
type function_def = {
  name:   string;
  params: string list;
  locals: string list;
  body:   seq;
}
```

La fonction `tr_function` de traduction prend en paramètre la définition `fdef` de la fonction à traduire, et commence par initialiser une table qui associe à chaque variable locale et chaque argument sa position dans le tableau d'activation, exprimée par rapport au pointeur de base `$fp`.

```
let tr_function fdef =
  let env = Hashtbl.create 16 in
  List.iteri (fun k id -> Hashtbl.add env id (4*(k+1))) fdef.params;
  List.iteri (fun k id -> Hashtbl.add env id (-4*(k+2))) fdef.locals;
```

Les fonctions `tr_expr`, `tr_instr` et `tr_seq` de traduction des expressions et des séquences d'instructions peuvent ensuite être définies dans ce contexte, et auront naturellement accès à cette table. Voici le début de la fonction de traduction des expressions, qui utilise le module auxiliaire MIPS présenté au chapitre 4 pour générer le code assembleur. Une constante entière est directement chargée dans le registre `$t0`, et dans le cas d'un identifiant de variable `x`, on commence par consulter la table pour voir si `x` est le nom d'une variable locale ou d'un paramètre. Le cas échéant, on lit sa valeur à la position correspondante du tableau d'activation.

```
  let rec tr_expr = function
    | Int n -> li t0 n
    | Var x ->
       if Hashtbl.mem env x then
         let offset = Hashtbl.find env x in
         lw t0 offset(fp)
       else ...
```

Sinon on suppose qu'il s'agit d'une variable globale, dont le nom est directement associé à une étiquette dans les données statiques. Il suffit d'aller lire à cette adresse.

```
       else
         la t0 x @@ lw t0 0(t0)
```

Pour les opérations binaires, comme au chapitre 4 on va stocker sur la pile la valeur du premier opérande le temps de calculer la valeur du deuxième opérande. Voici un échantillon, traitant les opérateurs utilisés dans l'exemple de la fonction `power`.

```
    | Binop(bop, e1, e2) ->
       let op = match bop with
         | Mul -> mul    (*  *   *)
         | Asr -> sra    (*  >>  *)
         | And -> and_   (*  &   *)
         | Eq  -> seq    (*  ==  *)
       in
       tr_expr e1 @@ push t0 @@ tr_expr e2 @@ pop t1 @@ op t0 t1 t0
```

Pour l'appel de fonction, la syntaxe abstraite de ImpScript introduit une petite subtilité par rapport à ce que nous avons vu au début du chapitre : la « fonction » `f` d'un appel `Call(f, args)` n'est pas directement un identifiant, mais une expression. L'évaluation de cette expression nous donnera l'adresse de la fonction à appeler, et en conséquence les appels de fonctions utiliseront l'instruction `jalr` plutôt que `jal`. Ainsi, le code d'un appel de fonction est réalisé en quatre parties, qui intègrent notamment les étapes du protocole d'appel qui concernent l'appelant (étapes 1 et 5) :

Dans les cas simples, cette expression ne sera rien d'autre qu'une variable globale contenant justement l'adresse de la fonction correspondante. Dans le chapitre sur la programmation objet, nous emploierons cependant des expressions plus complexes.

1. (appelant, avant l'appel) évaluer chacun des arguments et le placer sur la pile (fonction auxiliaire `tr_params` qui, remarquez, inverse l'ordre des arguments pour que le dernier soit bien le plus profond dans la pile),
2. évaluer l'expression désignant la fonction,
3. sauter avec `jalr` à l'adresse calculée à l'étape précédente,
4. (appelant, après l'appel) retirer les arguments de la pile.

En code :

```
  | Call(f, params) ->
     tr_params params
     @@ tr_expr f
     @@ jalr t0
     @@ addi sp sp (4 * List.length params)

and tr_params = function
  | []        -> nop
  | e::params -> tr_params params @@ tr_expr e @@ push t0
in
```

La traduction des instructions est similaire à celle vue au chapitre 4, et nécessite toujours la génération de nouvelles étiquettes pour les sauts.

Je profite de l'occasion pour vous montrer une traduction légèrement différente, mais également valide, de `if` et `while`. Essayez de comparer les deux !

```
let new_label =
  let cpt = ref (-1) in
  fun () -> incr cpt; Printf.sprintf "__%s_%i" fdef.name !cpt
  in
```

```
let rec tr_seq = function
  | []   -> nop
  | i::s -> tr_instr i @@ tr_seq s

and tr_instr = function
  | If(c, s1, s2) ->
     let then_label = new_label()
     and end_label = new_label()
     in
     tr_expr c @@ bnez t0 then_label
     (* fall to else branch *) @@ tr_seq s2 @@ b end_label
     @@ label then_label @@ tr_seq s1 (* fall through *)
     @@ label end_label

  | While(c, s) ->
     let test_label = new_label()
     and code_label = new_label()
     in
     b test_label
     @@ label code_label @@ tr_seq s
     @@ label test_label @@ tr_expr c @@ bnez t0 code_label
     (* fall through *)
```

Dans le cas d'une instruction `Set`, il faut ajouter comme pour les expressions `Var` une distinction entre les variables locales et les variables globales.

```
  | Set(x, e) ->
     tr_expr e
     @@ (if Hashtbl.mem env x then
           let offset = Hashtbl.find env x in
           sw t0 offset(fp)
         else
           la t1 x @@ sw t0 0(t1))
```

Enfin, l'instruction `Return` doit intégrer une partie du protocole d'appel (étape 4 : appelé, à la fin de l'appel). Une fois la valeur de l'expression `e` placée dans `$t0`, il faut détruire le tableau d'activation, et restaurer les registres `$ra` et `$fp` avant de rendre la main à l'appelant.

```
  | Return(e) ->
     tr_expr e
     @@ addi sp fp (-4)
```

```
      @@ pop ra @@ pop fp
      @@ jr ra
  in
```

Ces fonctions auxiliaires étant posées, nous pouvons revenir au travail principal de `tr_function` : la fabrication de code assembleur pour la fonction décrite par `fdef`. Ce code est constitué du corps `fdef.body` traduit par `tr_seq`, entouré des étapes du protocole d'appel qui sont à la charge de l'appelé (étapes 2 et 4).

```
  (* étape 2 *)
  push fp @@ push ra
  @@ addi fp sp 4
  @@ addi sp sp (-4 * List.length fdef.locals)
  (* exécution du corps de la fonction *)
  @@ tr_seq fdef.body
  (* étape 4 *)
  @@ li t0 0
  @@ addi sp fp (-4)
  @@ pop ra @@ pop fp
  @@ jr ra
```

Notez que les instructions données ici pour l'étape 4 ne seront effectivement exécutées que si l'exécution du corps de la fonction n'a pas déjà rencontré de `return`. Dans ce cas, on place ici la valeur `0` dans le registre `$t0` avant d'appliquer le protocole lui-même, ce qui correspond à insérer un `return 0;` ou `return null;` à la fin du code de la fonction.

Pour traduire un programme ImpScript complet `p`, tel que décrit par une structure

```
type program = {
  globals:   string list;
  functions: function_def list;
  code:      seq;
}
```

il suffit alors de :

- générer du code pour chaque définition de fonction dans `p.functions`,
- générer le code principal pour `p.code` (éventuellement en l'encapsulant dans une fonction factice que l'on appellerait `_main`, qui serait appelée au début de l'exécution),
- déclarer dans la section `.data` les variables globales énumérées par `p.globals`,
- déclarer une variable globale supplémentaire pour chaque fonction, dont la valeur initiale est l'adresse de la fonction correspondante.

On pourrait aussi se passer de ces variables supplémentaires, à condition de faire un cas particulier dans `Var` pour le cas d'une variable désignant un nom de fonction. De même, ces variables supplémentaires ne sont pas utiles si l'appel de fonction `Call(f, args)` prend systématiquement la fonction sous la forme d'un identifiant, par exemple avec une signature `Call of string * expr list` dans la définition de la syntaxe abstraite.

9.5 Approfondissement : convention d'appel avec registres

La convention d'appel standard de MIPS utilise les registres d'une manière plus fine que ce que nous avons fait dans la convention simplifiée.

Passage des arguments. Dans la convention standard de MIPS, les quatre premiers arguments sont passés par les registres `$a0` à `$a3`. Seuls les suivants éventuels, à partir du cinquième, sont passés par la pile. Le résultat, lui, est renvoyé via le registre `$v0`.

Notez que cela est cohérent avec les appels système. Dans l'appel système d'affichage d'un entier par exemple (ligne `040` dans le code p.130), la valeur à afficher est placée dans le registre `$a0`. De même, les appels système qui produisent un résultat le placent dans `$v0`.

Sauvegarde des registres. Dans le code assembleur que nous avons vu jusqu'ici, on utilisait beaucoup la pile mais très peu les registres : uniquement les registres spéciaux `$sp`, `$ra`, `$fp` pour gérer la pile et les appels de fonction, plus `$t0` et `$t1` de manière temporaire pour les calculs. On y a vu que `$sp` devait être maintenu à jour par chaque fonction, et que `$ra` et `$fp` devaient être sauvegardés chaque fois que leur valeur risquait d'être écrasée, et restaurés ensuite.

Lorsqu'un code assembleur tire réellement parti de l'ensemble des registres, pour stocker plus de valeurs intermédiaires des calculs, ou pour encore pour stocker des variables locales,

la convention d'appel doit également régler quand et comment la valeur de chacun doit être sauvegardée/restaurée. Pour cela, les registres sont séparés en deux paquets.

— Les registres *callee-saved* doivent être préservés par l'appelé. Si l'appelé les utilise, il doit donc au préalable sauvegarder leurs valeurs, puis les restaurer avant de rendre la main. En MIPS, il s'agit des registres `$s*` (*Saved*) et `$fp`.
— Les registres *caller-saved* peuvent être écrasés par l'appelé. C'est à l'appelant de les sauvegarder avant l'appel et de les restaurer ensuite s'il a encore besoin de leur valeur après l'appel. En MIPS, il s'agit des registres `$a*` (*Argument*), `$t*` (*Temporary*) et `$ra` (*Return Address*).

Notez que l'on a déjà mentionné que le `$ra` d'un appel de fonction devait être sauvegardé dans son tableau d'activation avant d'éventuels appels imbriqués, et que le `$fp` du contexte appelant devait être sauvegardé par la fonction appelée avant que celle-ci n'en modifie la valeur pour désigner son propre tableau d'activation.

Autrement dit, dans l'étape 1 du protocole d'appel, l'appelant doit également sauvegarder les valeurs des registres `$a*` ou `$t*` dont il aurait encore besoin après l'appel (et il peut ensuite les restaurer à l'étape 5). Inversement, l'appelé n'a besoin à l'étape 2 de sauvegarder que les valeurs des registres `$s*`, et même plus précisément que les valeurs des registres qu'il modifiera.

En conséquence, on utilise les registres `$t*` en priorité pour des valeurs intermédiaires à la durée de vie courte, pour ne pas avoir besoin de les sauvegarder avant de réaliser un appel. À l'inverse, les éléments que l'on souhaite préserver sur une durée plus longue seront avantageusement stockés dans les registres `$s*` : ils n'auront alors à être sauvegardés que lorsqu'une fonction appelée aura effectivement besoin d'utiliser ces mêmes registres.

C'est exactement ce qu'on a fait dans la convention simplifiée avec `$t0` et `$t1`, dont la durée de vie était si courte qu'ils n'ont jamais eu besoin d'être sauvegardés avant un appel.

Exponentiation rapide, avec la convention MIPS. On peut écrire une nouvelle version assembleur de notre fonction `power`, utilisant cette nouvelle convention. On aura donc ici le premier argument `a` transmis par le registre `$a0`, le deuxième argument `n` transmis par le registre `$a1`, et on va choisir en plus de stocker la variable locale `b` dans un registre temporaire `$t2` (on ne s'intéresse à la valeur de cette variable que pendant un petit intervalle de temps, qui ne contient pas d'appel).

Avec ces nouveaux choix, le code principal économise la manipulation de la pile.

```
    # Appel power(2, 9)
000'    li      $a1, 9          # préparation des arguments
012'    li      $a0, 2
024     jal     power           # appel
    # Affichage
032'    move    $a0, $v0
036     li      $v0, 1
040     syscall
    # Fin du programme
044     li      $v0, 10
048     syscall
```

Le tableau d'activation va maintenant contenir 4 cases, pour stocker `$ra` et `$fp` comme avant, mais aussi pour préserver les arguments `$a0` et `$a1` en cas d'appel récursif. En revanche, il n'y a plus de nécessité de prévoir une place pour `b`, dont le registre suffira.

```
    power:
    # Construction du tableau d'activation
052     addi    $sp, $sp, -16   # 4 mots réservés au lieu de 3
056     sw      $fp, 12($sp)
060     sw      $ra, 8($sp)
064     addi    $fp, $sp, 12
    # Exécution du corps de la fonction
072'    bnez    $a1, power_rec  # test (n == 0)
076     li      $v0, 1          # v0 <- résultat (return 1)
080     b       power_end
```

Avant et après l'appel récursif, on insère de nouvelles instructions pour sauvegarder puis restaurer les registres contenant les arguments. En revanche, la préparation des arguments de l'appel est simplifiée.

```
    power_rec:
    # Appel récursif
084'    sw      $a0, -8($fp)    # sauvegarde a0 et a1
```

```
084''   sw      $a1, -12($fp)
088'    sra     $a1, $a1, 1     # prép. argument (n>>1)
104'    mul     $a0, $a0, $a0   # prép. argument (a*a)
116     jal     power           # appel
120'    lw      $a0, -8($fp)    # restauration a0 et a1
120''   lw      $a1, -12($fp)
```

Le calcul du résultat est lui aussi simplifié, et le nettoyage final du tableau d'activation reste identique à la version précédente.

```
    # Retour à l'exécution du corps de la fonction
124     move    $t2, $v0        # b <- résultat power(a*a, n>>1)
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136     bnez    $t0, power_odd
140     move    $v0, $t2        # v0 <- résultat (return b)
144     b       power_end
    power_odd:
156'    mul     $v0, $t2, $a0   # v0 <- résultat (return a*b)
    power_end:
    # Destruction du tableau d'activation
160     lw      $ra, 8($sp)
164     lw      $fp, 12($sp)
168     addi    $sp, $sp, 16
172     jr      $ra
```

Simplifications. En observant un peu le dernier code obtenu, on peut remarquer quelques simplifications possibles. D'une part, on a deux transferts directs entre le registre temporaire `$t2` choisi pour stocker la valeur de la variable `b` et le registre `$v0` dans lequel doit se trouver le résultat de tout appel de fonction :

— ligne `124`, où le résultat de l'appel récursif obtenu dans `$v0` est stocké dans `b`,

— ligne `140`, où cette même valeur de `b` revient dans `$v0` comme résultat de l'appel courant.

On pourrait économiser un registre et ces deux transferts en utilisant directement `$v0` pour stocker la valeur de `b`. La fin du corps de la fonction deviendrait donc

```
    # Retour à l'exécution du corps de la fonction
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136     bnez    $t0, power_odd
144     b       power_end
    power_odd:
156'    mul     $v0, $v0, $a0   # v0 <- résultat (return a*b)
    power_end:
```

où les lignes `124` et `140` ont disparu, et où la ligne `156'` a été adaptée. Ensuite, on peut voir que la disparition de la ligne `140` a achevé de vider la branche positive du test (`n&1 == 0`), qui ne fait plus rien d'autre que sauter à la séquence de fin. On peut alors réorganiser ces branches pour une version finale plus directe.

```
    # Retour à l'exécution du corps de la fonction
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136'    beqz    $t0, power_end
156'    mul     $v0, $v0, $a0   # v0 <- résultat (return a*b)
    power_end:
```

9.6 Approfondissement : optimisation des appels terminaux

On pourrait imaginer aller plus loin dans les simplifications. Si on peut réorganiser le code de `power` pour que l'un ou l'autre des arguments `a` et `n` ne soit plus utile après l'appel récursif, la sauvegarde de cet argument avant chaque appel ne serait plus utile. C'est ce qui se serait passé par exemple pour `n` si nous avions pris comme code d'origine la version suivante.

```
function power(a, n) {
  if (n == 0) { return 1; }
  else { if (n&1 == 0) { return power(a*a, n>>1); }
         else          { return a*power(a*a, n>>1); } }
}
```

En approfondissant cette idée, on peut aboutir à une simplification bien plus spectaculaire, dans laquelle ni les arguments, ni `$ra`, ni `$fp` n'ont besoin d'être sauvegardés. Pour cela, il nous faut une version de `power` dans laquelle tous les appels sont ***terminaux*** :

```
1   function power(a, n) {
2     return power_aux(a, n, 1);
3   }

4   function power_aux(a, n, acc) {
5     if (n == 0) { return acc; }
6     else {
7       if (n&1 == 0) { return power_aux(a*a, n>>1, acc); }
8       else          { return power_aux(a*a, n>>1, a*acc); }
9     }
10  }

11  console.log(power(2, 9));
```

Dans ce code, l'appel à `power_aux` ligne `2` est quasiment la dernière action de la fonction `power` : après cet appel, il ne reste à `power` qu'à nettoyer son propre tableau d'activation et sauter au contexte appelant, en transférant directement le résultat qui a été donné par `power_aux`.

Remarquez que ce dernier point est particulièrement simple : le résultat que `power` doit placer dans `$v0` a justement été laissé là par `power_aux` ! On pourrait alors imaginer que la fonction `power`, plutôt que de demander un résultat `power_aux` et transmettre ce résultat tel quel au contexte appelant, demande à `power_aux` de transmettre directement son résultat au contexte appelant. Pour cela, on réalise l'appel à `power_aux` par un saut simple

```
  j   power_aux
```

au lieu de l'habituel saut `jal`. Ainsi, l'appel à `power_aux` aura comme adresse de retour la valeur courante de `$ra`, c'est-à-dire ici justement l'adresse à laquelle il fallait reprendre après l'appel à `power`.

Il reste cependant à régler la question du nettoyage du tableau d'activation de `power`. Pour cela, on peut partir d'une deuxième remarque : plus aucun élément du tableau d'activation de l'appel à `power` ne sera utile après l'appel à `power_aux`, exceptée la valeur sauvegardée du pointeur `$fp` du contexte appelant qui doit être restaurée. On pourrait donc détruire le tableau d'activation de `power` (et restaurer le `$fp` appelant) *avant* de réaliser l'appel à `power_aux`. Ou encore, pour limiter les sauvegardes/restaurations redondantes, réutiliser le tableau d'activation de `power` pour l'appel à `power_aux` sans modifier les valeurs sauvegardées de `$ra` et `$fp`. Ou enfin ici, ne même jamais construire ce tableau d'activation, puisque dans notre cas il n'aura jamais besoin de sauvegarder quoi que ce soit.

Remarquez en outre que les deux appels récursifs à `power_aux`, aux lignes `7` et `8`, sont encore terminaux. La simplification que nous venons de décrire pour `power` peut donc encore être appliquée à `power_aux`. Voici le code que l'on obtiendrait, en passant les arguments `a` et `n` par les registres `$a0` et `$a1` (pour `power` comme pour `power_aux`), en passant l'argument `acc` de `power_aux` par le registre `$a2`, en réalisant chaque appel terminal par un saut simple et en se passant des tableaux d'activation superflus.

On commence avec le même code que précédemment pour l'instruction principale réalisant l'appel `power(2, 9)` et affichant le résultat. Celui notamment est bien toujours réalisé avec `jal`.

```
    # Appel power(2, 9)
00      li      $a0, 2
04      li      $a1, 9
08      jal     power
    # Affichage
12      move    $a0, $v0
16      li      $v0, 1
20      syscall
    # Fin du programme
24      li      $v0, 10
28      syscall
```

Après, les choses se simplifient drastiquement. La fonction `power` réalise un unique appel terminal à `power_aux`. Il n'y a pas de tableau d'activation à créer, les deux premiers arguments sont déjà dans `$a0` et `$a1`, il suffit de placer le troisième argument dans `$a2` puis sauter à la fonction auxiliaire.

```
    power:
32      li      $a2, 1          # initialisation de l'accumulateur
36      j       power_aux       # appel power_aux(a, n, 1)
```

Notez que l'on utilise l'instruction de saut simple `j` et pas l'instruction d'appel `jal`, puisque `$ra` a déjà la bonne valeur.

Vient ensuite la fonction auxiliaire `power_aux`. Celle-ci n'a pas non plus besoin de tableau d'activation, puisqu'elle n'a pas de variables locales et n'opère que des appels terminaux à elle-même. On réalise donc directement le test du cas d'arrêt. S'il est positif, on saute à la branche « `then` », ici l'étiquette `power_end` qui terminera la fonction. Petite simplification possible au passage : le saut ligne `36` est superflu, puisque sa cible `power_aux` est justement l'instruction suivante.

```
    power_aux:
40      beqz    $a1, power_end  # cas d'arrêt
```

Sinon, `power_aux` va ensuite préparer les arguments `a′`, `n′` et `acc′` pour le prochain appel récursif, puis relancer `power_aux`, toujours avec un saut simple. Les arguments `a` et `n` sont mis à jour systématiquement aux lignes `56` et `60`. En revanche, le test ligne `48` est susceptible de court-circuiter la mise à jour de `acc` ligne `52`. On obtient donc bien l'un des deux appels récursifs (ligne `7` ou ligne `8` du code source) selon la parité de `n`.

```
44      andi    $t0, $a1, 0x1   # test (n == 0)
48      beqz    $t0, power_even
52      mul     $a2, $a0, $a2   # acc' <- a*acc
    power_even:
56      mul     $a0, $a0, $a0   # a' <- a*a
60      sra     $a1, $a1, 1     # n' <- n>>1
64      j       power_aux
```

Dans le cas d'arrêt, on place le résultat dans `$v0`, et on revient au contexte appelant avec l'instruction habituelle `jr`, en ciblant le registre `$ra` dont la valeur n'a jamais été écrasée au cours de l'exécution !

```
    power_end:
68      move    $v0, $a2        # return acc
72      jr      $ra
```

On a donc finalement un code qui travaille uniquement avec les registres, et plus du tout avec la pile. Le résultat est efficace, et évite tout risque de d'erreur « `StackOverflow` ». En réalité, ce code assembleur est essentiellement équivalent à celui que nous aurions obtenu en compilant le code suivant, qui utilise une boucle plutôt qu'une fonction auxiliaire récursive.

```
function power(a, n) {
  let acc = 1;
  while (n != 0) {
    if (n&1 != 0) { acc = a*acc; }
    a = a*a;
    n = n>>1;
  }
  return acc;
}
```

L'optimisation des appels terminaux décrite ici est particulièrement utile pour des langages utilisant massivement des fonctions récursives (comme caml), mais reste pertinente y compris en dehors. Elle est par exemple également réalisée par GCC, lorsque l'on demande à ce compilateur un niveau d'optimisation élevé. Elle est cependant assez rarement réalisée, certains langages préférant maintenir une pile d'appels complète à des fins de débogage (python, java).

10 Structures de données et tas mémoire

Jusqu'ici, les petits langages de programmation que nous avons compilés ne manipulaient que des données simples, comme des nombres entiers. L'objectif de ce chapitre est d'y ajouter des structures de données plus riches.

Données. Les données manipulées par un programme, selon leur nature, peuvent être représentées de différentes manières en mémoire :

- un nombre entier ou un booléen est représenté par un unique mot mémoire (par exemple de 4 octets),
- une donnée plus complexe comme un tableau, un *n*-uplet, une structure, un objet ou une fermeture sera en revanche représentée par plusieurs mots, formant généralement un *bloc*, c'est-à-dire une suite de mots consécutifs en mémoire, et on peut désigner de telles données à l'aide de pointeurs,
- une structure plus riche encore comme une liste chaînée, un arbre ou une table de hachage pourra même être représentée par plusieurs blocs, non consécutifs en mémoire mais reliés par des pointeurs.

Nous allons voir ici en particulier comment réaliser les tableaux présents dans ImpScript, qui serviront de base à des structures riches au prochain chapitre, et voir aussi plus largement comment gérer un ensemble de données en mémoire.

10.1 Tableaux

Dans quelle région précise de la mémoire, on le verra bientôt.

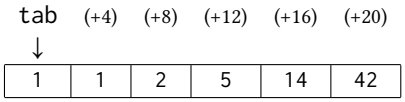
Un ***tableau*** est représenté par une séquence de mots consécutifs en mémoire, et identifié par l'*adresse* de son premier élément. Une définition C comme

```
int tab[] = {1, 1, 2, 5, 14, 42};
```

ou son équivalent ImpScript

```
let tab = [1, 1, 2, 5, 14, 42];
```

pourra ainsi mener à un schéma comme le suivant, où chaque élément du tableau occupe quatre octets.



C'est similaire à l'accès à un élément du tableau d'activation d'un appel de fonction.

Ici, le premier élément du tableau est directement à l'adresse donnée par la variable `tab`, le dernier est à l'adresse `tab+20`, et de manière générale on accède à chaque élément en partant de l'adresse de base `tab`, et en ajoutant un décalage correspondant à l'indice de la case cherchée (par multiples de 4 octets).

Tableaux alloués statiquement. Considérons d'abord la situation où le tableau `tab` précédent est un tableau global, défini à la racine du programme que l'on cherche à compiler. Comme les autres variables globales vues jusqu'ici, ce tableau peut être placé en mémoire dans les données statiques. En revanche, au lieu de lui allouer un unique mot mémoire on en donne six consécutifs : un pour chaque case du tableau. On initialise un tel tableau en MIPS avec le fragment suivant.

```
        .data
tab:    .word  1 1 2 5 14 42
```

Une caractéristique de cette version est qu'elle demande que les éléments du tableau soient connus « statiquement », c'est-à-dire dès la compilation du programme. Autrement dit, le texte même du programme doit les fournir explicitement et entièrement. On peut assouplir un petit peu cette contrainte, à condition de connaître statiquement au moins la taille du tableau. Ainsi, la déclaration d'un tableau non initialisé *de taille fixe*, écrite en C

```
int tab[6];
```

et en ImpScript

```
let tab = Array(6);
```

pourra être traduite par le code MIPS suivant où on a toujours bien une séquence de six valeurs (mais un choix arbitraire pour leur contenu).

```
        .data
tab:    .word  0 0 0 0 0 0
```

Dans l'une ou l'autre version, l'étiquette `tab` désigne l'adresse de base du tableau statique, et c'est d'elle que l'on se sert pour calculer l'adresse de chaque élément. On accède ainsi au quatrième élément à l'aide d'un décalage de 12 par rapport à l'adresse de base désignée par l'étiquette `tab` :

```
        la   $t0, tab
        lw   $t1, 12($t0)
```

Dans cette première version l'indice est connu statiquement : on peut calculer le décalage en amont, et l'écrire explicitement dans le code. Si en revanche l'indice auquel chercher est donné par un registre, il va falloir ajouter un peu de code MIPS pour calculer dynamiquement (c'est-à-dire : à l'exécution) le bon décalage. Pour accéder à une case dont l'indice est donné par le registre `$a0`, il faut donc multiplier la valeur de `$a0` par 4 pour obtenir le bon décalage (on le fait efficacement par un décalage de 2 bits vers la gauche, avec l'instruction `sll`), puis ajouter ce décalage à l'adresse de base.

Le décalage donné dans l'instruction `lw` ne peut être qu'une constante entière explicite, on ne peut pas y mettre `$a0`.

```
        la   $t0, tab
        sll  $a0, 2
        add  $t0, $t0, $a0
        lw   $t1, 0($t0)
```

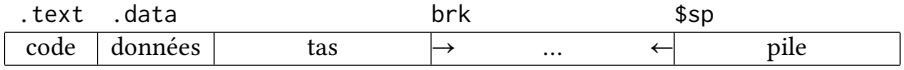
Lors de l'instruction `lw` on n'indique donc plus de décalage par rapport à l'adresse donnée par `$t0`, puisque que celui-ci contient déjà précisément l'adresse voulue.

Tableaux alloués dynamiquement. Imaginons cette fois le cas d'un tableau créé dynamiquement, avec une taille qui n'est pas connue explicitement à l'avance.

```
int f(int n) {
  int tab[n];
  ...
}
```

C'est beaucoup plus souple pour le programmeur, mais demande une nouvelle stratégie pour le compilateur. On peut imaginer que ce tableau soit représenté par n mots consécutifs en mémoire, rangés sur la pile avec les autres variables locales. C'est effectivement ce qui se passe en C, par défaut. Attention cependant : comme tout ce qui se trouve sur la pile, ce tableau a une durée de vie limitée et sera détruit à la fin de l'appel.

Nous allons nous concentrer ici sur une troisième stratégie, permettant d'allouer dynamiquement des structures de données ***persistantes***, qui vont survivre à l'appel de fonction qui les a créées. On utilise pour cela une nouvelle région de la mémoire : le ***tas***, qui se trouve entre les données statiques et la pile.



La plus petite adresse du tas est immédiatement au-dessus de la zone des données statiques. Un pointeur `brk` appelé *memory break* identifie la fin du tas, ou plus précisément la première adresse au-delà du tas. Ainsi le dernier mot du tas est à l'adresse `brk-4`. L'espace situé entre le tas et la pile, c'est-à-dire entre les pointeurs `brk` (inclus) et `$sp` (exclu), est vide. Lorsque l'on a besoin de plus de place sur le tas, on incrémente le pointeur `brk`, pour absorber une partie de l'espace libre entre le tas et la pile.

Le pointeur `brk` est géré par le système d'exploitation, aussi son déplacement est fait par un appel système `sbrk`, qui a le code 9 dans notre simulateur MIPS. Cet appel système prend comme argument, via `$a0`, le nombre d'octets dont `brk` doit être incrémenté. Il renvoie comme résultat, via `$v0`, l'ancienne valeur de `brk`, c'est-à-dire la première adresse de la zone qui vient d'être ajoutée au tas.

On garde un point commun avec la version précédente : tout tableau alloué ainsi par extension du tas est désigné par un pointeur vers sa première case. La différence est que ce pointeur ne correspond plus à une étiquette du code assembleur, mais transite par les registres comme une valeur ordinaire.

Voici un exemple d'utilisation de ce mécanisme, dans lequel on alloue sur le tas de l'espace pour un tableau de 6 cases (24 octets), avant d'écrire `1` dans la première case et `32` dans la sixième.

instruction	$a0	$v0	$t0	brk
				0x10040000
`li $a0, 24`	24			0x10040000
`li $v0, 9`	24	9		0x10040000
`syscall`	24	0x10040000		0x10040018
`li $t0, 1`	24	0x10040000	1	0x10040018
`sw $t0, 0($v0)`	24	0x10040000	1	0x10040018
`li $t0, 32`	24	0x10040000	32	0x10040018
`sw $t0, 20($v0)`	24	0x10040000	32	0x10040018

La forme du tas associée à cet exemple est la suivante, où l'adresse $@_1$ est la position d'origine de `brk` (0x10040000) et $@_2$ est la nouvelle position après appel à `sbrk` (0x10040018).

	$@_1$					$@_2$	
...	1					32	...

Approfondissement : extension du compilateur ImpScript. La manipulation des tableaux en ImpScript est réalisée par quatre constructions de sa syntaxe abstraite : une expression d'accès (en lecture) à un élément d'un tableau et une instruction de modification d'une case d'un tableau, plus deux expressions de création d'un nouveau tableau (spécifié par une liste explicite d'éléments ou par une simple taille).

```
type instr = ...
  | ArrSet of expr * expr * expr  (*  e1[e2] = e3;   *)

type expr = ...
  | ArrGet of expr * expr         (*  e1[e2]         *)
  | Array  of expr list           (*  [e1, ..., eN]  *)
  | EArray of expr                (*  Array(e)       *)
```

Nous devons donc compléter les fonctions `tr_expr` et `tr_instr` de notre compilateur pour traiter ces nouveaux cas. Pour cela, on peut déjà isoler une partie commune à `ArrSet` et `ArrGet` : toutes deux demandent de calculer l'adresse de la case de tableau ciblée `e1[e2]`, qui est égale à `e1 + 4*e2`. La fonction auxiliaire `tr_access` prend en paramètres deux expressions `e1` et `e2`, et produit un code assembleur qui évalue ces deux expressions et en déduit l'adresse de la case `e1[e2]` (cette adresse étant elle-même à la fin placée dans le registre `$t0`). Voici cette fonction, qui doit être définie conjointement avec `tr_expr`.

```
let rec tr_access e1 e2 =
  tr_expr e1
  @@ push t0
  @@ tr_expr e2
  @@ sll t0 t0 2
  @@ pop t1
  @@ add t0 t1 t0
```

Les deux cas `ArrSet` et `ArrGet` sont alors simplement une combinaison de ce calcul d'adresse avec une opération de lecture ou d'écriture.

```
let tr_expr = ...
  | ArrGet(e1, e2) ->
     tr_access e1 e2
     @@ lw t0 0(t0)

let tr_instr = ...
  | ArrSet(e1, e2, e3) ->
     tr_access e1 e2
     @@ push t0
     @@ tr_expr e3
     @@ pop t1
     @@ sw t0 0(t1)
```

La création d'un tableau non initialisé se résume à un appel à `sbrk` pour réserver la bonne quantité de mémoire sur le tas. Attention toutefois : le nombre d'éléments du tableau donné par

l'expression e doit être multiplié par 4 pour obtenir le bon nombre d'octets. Le « résultat » d'une telle expression est un pointeur vers la première case du tableau créé, que l'on transfère dans `$t0` à la fin de la séquence.

```
let rec tr_expr = ...
  | EArray e ->
     tr_expr e
     @@ sll a0 t0 2 @@ li v0 9 @@ syscall
     @@ move t0 v0
```

La création d'un tableau initialisé nécessite d'abord l'allocation du tableau lui-même selon le même modèle que ci-dessus, à ceci prêt que la taille est connue statiquement. Ensuite, il faut évaluer chaque élément et l'écrire dans la bonne case. On propose pour cela d'enregistrer sur la pile l'adresse du tableau créé le temps de finir l'initialisation (fonction auxiliaire `write_elts` à définir conjointement avec `tr_expr`).

```
  | Array(elts) ->
     let n = List.length elts in
     li a0 (4*n) @@ li v0 9 @@ syscall
     @@ push v0
     @@ write_elts elts 0
     @@ pop t0

and write_elts elts offset = match elts with
  | [] -> nop
  | e::tl ->
     tr_expr e
     @@ lw t1 0(sp)  (* rappel de l'adresse de base *)
     @@ sw t0 offset(t1)
     @@ write_elts tl (offset+4)
```

Petit détail dans la fonction `write_elts` : chaque évaluation d'une nouvelle expression est susceptible d'utiliser les registres `$t0` et `$t1` et d'écraser ce qui s'y trouve. En particulier, il faut donc à chaque fois recharger l'adresse de base du tableau, qui a été sauvegardée au sommet de la pile dans ce but.

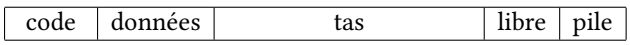
Dans ce modèle, on considère que la *valeur* d'une expression désignant un tableau est un *pointeur* vers ce tableau (le tableau lui-même *n'étant pas* une valeur). Ainsi, lorsque l'on passe un tableau en argument à une fonction, on réalise toujours un passage « par valeur » (d'une valeur qui est un pointeur).

C'est identique à ce que pratiquent caml, python et java. C'est différent du passage « par référence », possible en C, qui consiste à transmettre un pointeur vers une valeur.

Un aspect agréable de ce modèle partagé avec caml, python et java est que toute valeur est représentée par un mot de 4 octets (une valeur est soit une donnée de base comme un entier, soit un pointeur vers un bloc mémoire).

10.2 Gestion dynamique de la mémoire

Nous avons évoqué avec les tableaux l'utilisation d'une nouvelle région de la mémoire, appelée le ***tas***, que nous allons maintenant détailler.



Le tas est une région de la mémoire qui est adaptée au stockage des données persistantes. Cette région est divisée en ***blocs*** et est gérée par un programme dédié qui permet en particulier de :

- demander des blocs de mémoire pour stocker de nouvelles données,
- libérer des blocs de mémoire qui avaient été affectés à des données qui ne sont plus utiles, pour qu'ils puissent être à nouveau utilisés pour d'autres données.

Ces deux opérations sont qualifiées de ***dynamiques***, car elles ont lieu *pendant l'exécution* du programme principal. Dans cette section, on détaille *une* manière d'organiser une telle mémoire dynamique.

Les variations possibles sont nombreuses !

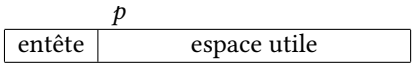
Cadre pour la gestion de la mémoire dynamique. Selon les cas, le programme dédié à la gestion de la mémoire peut prendre différentes formes.

- En C, on utilise la bibliothèque `malloc`, et notamment les fonctions `malloc` et `free` pour allouer ou libérer manuellement de la mémoire.
- En caml, java, python, un gestionnaire automatique appelé *GC* (*garbage collector*, ou *glaneur de cellules*) est intégré à l'environnement d'exécution. Il tourne en parallèle du programme principal et identifie puis libère les parties du tas qui ne sont plus utiles.

Nous allons détailler ici le fonctionnement des opérations manuelles `malloc` et `free`, qui peuvent ensuite servir de base à la réalisation d'un gestionnaire automatique.

Par exemple au prochain chapitre.

Blocs mémoire. Les ***blocs*** sont l'unité de gestion de la mémoire dynamique. Un bloc est une zone de mémoire contiguë, qui peut être libre ou utilisée. Chaque bloc est constitué d'un entête et d'un espace utile, et est identifié par un pointeur.



L'espace utile est la partie effectivement utilisée pour stocker des données, tandis que l'entête contient des informations additionnelles utilisées exclusivement par le gestionnaire de mémoire. On accède au contenu d'un bloc à l'aide d'un pointeur p, qui donne l'adresse du premier octet utile. On a couramment des contraintes sur l'adresse désignée par le pointeur p, par exemple qu'il s'agisse d'un multiple de 8 (on le supposera effectivement dans la suite).

En reprenant la notation d'accès à un tableau, on pourrait écrire également `p[-1]` pour l'entête et `p[i]` pour le mot d'indice i.

En supposant que l'entête soit représenté par un mot mémoire, et chaque donnée stockée dans l'espace utile soit également représentée par un mot mémoire, on peut donc accèder à l'entête à l'adresse `p - 4` et au mot d'indice i à l'adresse `p + 4*`i.

L'entête d'un bloc doit contenir les informations utiles au gestionnaire de mémoire. Les deux principales sont :

- le statut *libre* ou *réservé* du bloc, correspondant à un bit d'information valant `1` pour un bloc réservé et `0` pour un bloc libre,
- la *taille* totale du bloc (espace utile + entête), exprimée par un nombre entier d'octets.

Si l'on suppose que la taille d'un bloc est un multiple de 8, cette dernière est représentée par un nombre dont les trois derniers bits ne donnent pas d'information (pour un multiple de 8 ces trois bits valent nécessairement `0`). On peut alors enregistrer à la fois la taille t et le statut s dans un unique mot mémoire e, en plaçant le bit de statut à la place du dernier bit de taille. On peut réaliser ceci à l'aide d'une opération de « ou » bit à bit : `e = t | s`.

t	s	e
32	libre	$0b100000$
24	réservé	$0b011001$

On peut à l'inverse récupérer le statut ou la taille à partir d'un mot d'entête e à l'aide d'opérations de masquage :

- le statut est donné par le dernier bit : `s = e & 0x1`,
- la taille est obtenue en masquant les trois derniers bits : `t = e & ~0x7`.

Gestion manuelle : `malloc` et `free`. On veut donc deux opérations `malloc` et `free` répondant aux spécifications suivantes.

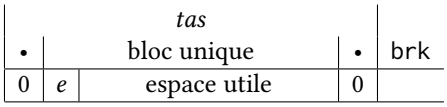
- Un appel `malloc(`n`)`, avec n un entier positif, trouve dans le tas un bloc *libre* ayant au moins n octets utiles, le réserve, et renvoie un pointeur vers ce bloc.
- Un appel `free(`p`)` libère le bloc de mémoire à l'adresse donnée par le pointeur p.

Si, lors d'un appel à `malloc`, le tas ne contient pas de bloc libre adapté, la fonction peut soit étendre le tas à l'aide d'un appel `sbrk`, soit échouer. Par ailleurs, l'appel `free(`p`)` présuppose que le pointeur p désigne un bloc mémoire qui a effectivement été alloué par `malloc` : dans le cas contraire, l'effet est indéterminé.

Conjuguer la contiguïté des blocs et les adresses multiples de 8 peut imposer « d'arrondir » vers le haut les tailles de bloc qu'un utilisateur demande avec `malloc`. C'est normal et (quasi-)invisible.

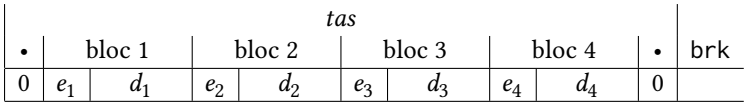
On considère à partir de maintenant que l'ensemble du tas est formé de blocs consécutifs en mémoire, sans espace libre entre les blocs, et est intégralement géré par l'intermédiaire des fonctions `malloc` et `free` que nous allons construire.

Initialisation. Pour initialiser un tel tas, on demande au système une zone de mémoire à l'aide d'un appel `sbrk`. Pour marquer les extrémités de cette zone, on laisse une valeur spéciale (en l'occurrence zéro) dans le premier mot et le dernier mot. Tout le reste forme un grand bloc.



Si on a alloué 1024 octets pour l'ensemble du tas, une fois sacrifiés les premier et dernier mots il reste 1016 octets pour le bloc : 4 octets d'entête et 1012 octets utiles. Ce bloc est vide, son bit de statut vaut donc 0. Ainsi l'entête e vaut 1016. Notez également que le pointeur p vers l'espace utile de cet unique bloc est un multiple de 8, puisque cette adresse se situe deux mots après le début du tas (lui-même à l'adresse `0x1004000` en MIPS).

Lorsque le tas est étendu par un nouvel appel à `sbrk`, on déplace le zéro final pour préserver la forme générale : une séquence de blocs consécutifs, donc les extrémités sont marquées par deux mots nuls.



Fonction `malloc` v1, parcours séquentiel. Considérons un appel `malloc(`n`)`. La fonction `malloc` doit rechercher dans le tas un bloc libre suffisamment grand. On peut imaginer une première version qui parcourt séquentiellement tous les blocs du tas jusqu'à en trouver un adapté. Ce parcours nécessite deux opérations principales.

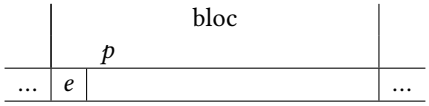
1. **Tester un bloc désigné par un pointeur** p**.** Pour cela, il faut regarder l'entête (`p[-1]`), la décomposer en un bit de statut s et une taille t, tester si le bloc est libre (`!`s), et s'il contient assez d'octets utiles (`n <= t - 4`).
2. **Passer au bloc suivant.** En ajoutant au pointeur p désignant un bloc la taille t de ce bloc, on obtient l'adresse du bloc suivant : on a un chaînage implicite des blocs.

Ces opérations de base étant fixées, on peut varier les stratégies. Par exemple :

— commencer au premier bloc et s'arrêter au premier bloc convenable trouvé (*first fit*),
— commencer au bloc touché le plus récemment et s'arrêter au premier bloc convenable trouvé (*next fit*), ou
— chercher le plus petit des blocs convenables (*best fit*).

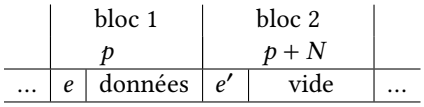
Enfin, si on a parcouru l'ensemble des blocs sans en trouver de convenable on peut effectuer un nouvel appel à `sbrk` pour étendre le tas, du moins tant qu'il reste de l'espace disponible. On réserve alors un nouveau bloc taillé dans cette extension.

Réservation d'un bloc. Une fois qu'on a trouvé un bloc convenable, désigné par un pointeur p vers son premier octet utile, on peut le réserver.



Il y a encore deux possibilités :

— on peut réserver intégralement le bloc, en modifiant son bit de statut ($e \leftarrow e$ `| 0x1`),
— si le bloc est grand, on peut aussi le découper en deux blocs, un qui sera réservé et l'autre laissé libre.



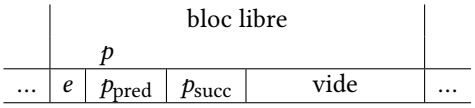
Dans ce schéma, on a réservé un premier bloc de taille N, pour une certaine valeur $N \geq n+4$ multiple de 8, et laissé un deuxième bloc libre de taille $t - N$.

Le choix de découper ou non est à la discrétion de l'allocateur mémoire, et n'est pas si simple a priori : découper permet d'éviter de sous-employer un bloc, mais peut aussi impliquer à terme une fragmentation de la mémoire en de multiples petits blocs difficilement réutilisables. De même, le choix de la valeur N appartient à la stratégie de l'allocateur.

Fonction `free` v1. Il y a une manière simple de libérer un bloc désigné par un pointeur p : aller modifier dans son entête le bit de statut, pour le repasser à `0` ($e \leftarrow e$ `& ~0x1`).

Approfondissement : fonction `malloc` v2, listes libres. La première version de la recherche parcourait séquentiellement tous les blocs, libres ou réservés, jusqu'à avoir choisi le bloc à réserver. On pourrait faire nettement plus efficace en ne considérant que les blocs libres. Cela suppose en revanche d'avoir un chaînage entre les blocs libres, c'est-à-dire d'être capable, étant donné un bloc libre, de déterminer efficacement l'adresse d'un bloc libre « suivant ». Une astuce pour cela consiste à remarquer que tout bloc libre est… libre ! Autrement dit, l'espace utile d'un bloc libre n'est pas utilisé par le programme client, et l'allocateur peut l'utiliser pour stocker des méta-données supplémentaires. En l'occurrence, on prend les deux premiers champs de l'espace utile d'un bloc libre pour stocker les adresses p_{pred} et p_{succ} du bloc libre précédent et du bloc libre suivant.

D'une certaine manière, on stocke des méta-données dans les interstices de la structure principale.



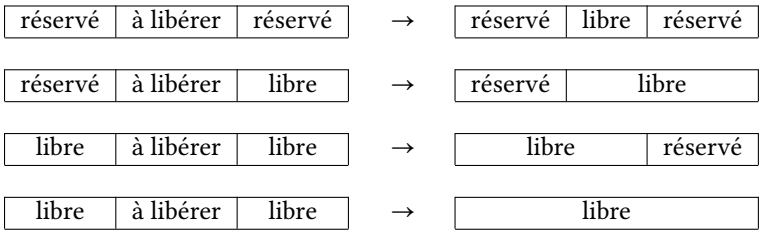
À noter : pour bien avoir la place de stocker ces deux pointeurs, il faut que la taille de chaque bloc soit d'au moins 12 octets, et même 16 octets en tenant compte de la contrainte « multiple de 8 ».

On peut donc coder, à l'aide des ces zones temporairement inutilisées de la mémoire, une liste doublement chaînée des blocs libres (*free list*), qui sera aisément mise à jour à chaque changement de statut d'un bloc. Il faut simplement mémoriser quelque part, dans une variable globale, l'adresse d'un premier bloc libre. On améliore encore cette technique en n'utilisant pas une seule mais plusieurs listes doublement chaînées de blocs libres, en distinguant plusieurs catégories de tailles (*segregated free lists*).

Attention, adopter une stratégie *free list* ou *segregated free lists* demande également une petite mise à jour de la fonction `free`, pour réintégrer le bloc libéré dans la liste des blocs libres. On peut le faire par exemple juste avant le premier bloc, en suivant un algorithme standard d'insertion dans une liste doublement chaînée.

Approfondissement : fonction `free` v2, libérer et fusionner. Le défaut de la première version de `free` est l'apparition progressive de *fragmentation* de la mémoire : si on a découpé un grand bloc pour éviter de perdre de la place (ce qui peut être une bonne chose), on aura après libération plusieurs petits blocs. Ces petits blocs ne pourront être utilisés que pour de petites allocations, et il sera à terme de plus en plus difficile de trouver un bloc de grande taille pour une une allocation d'un grand nombre d'octets.

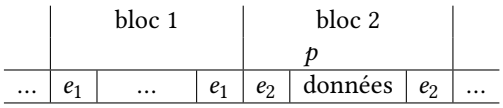
On peut améliorer la situation en fusionnant un groupe de blocs libres adjacents, c'est-à-dire consécutifs en mémoire. Cette opération de fusion peut même être intégrée à l'opération `free` elle-même : si on trouve un bloc libre juste avant ou juste après le bloc libéré par `free`, on les fusionne. Avec les différents statuts « libre » ou « réservé » des deux blocs entourant le bloc libéré par `free`, on a les quatre situations possibles suivantes.



À noter : si ces fusions sont faites systématiquement, on a la garantie de ne jamais avoir deux blocs libres consécutifs.

Pour réaliser ces fusions il faut pouvoir consulter les méta-données, c'est-à-dire les entêtes, des blocs situés juste avant et juste après le bloc libéré.

- Partant du pointeur p du bloc à libérer, on accède facilement au bloc suivant : il suffit d'ajouter la taille du bloc à libérer (et l'entête est le mot précédant l'adresse obtenue).
- En l'état, on n'a en revanche pas de manière directe de calculer l'adresse du bloc précédent. On peut régler ce problème en enrichissant encore un peu la structure de nos blocs, en répétant l'entête d'un bloc à la fin de celui-ci (*boundary tags*).

  On accède alors aux méta-données du bloc précédent en regardant deux mots avant l'adresse p.

La technique des *boundary tags* permet de meilleures fusions. Elle a néanmoins un coût : il faut sacrifier un bloc supplémentaire pour cette deuxième copie de l'entête. On peut cependant ajouter une nouvelle optimisation (fonction `free` v2.5) : ne répéter l'entête que dans le cas d'un bloc libre, car c'est seulement dans cette situation que l'on a besoin de la taille, et ajouter dans l'entête de chaque bloc un bit donnant le statut du bloc précédent.

Bilan : un bon allocateur de mémoire est un programme assez subtil, qui repose sur des structures de données qui sont cachées dans les interstices du tas. Il y a beaucoup de choix de stratégie à faire, et une abondance d'études empiriques sur l'efficacité des différentes stratégies. Dans Linux, la bibliothèque `malloc.c` *contient plus de 5000 lignes de code.*

10.3 Structures de données

Les langages de programmation permettent couramment la définition de structures de données composées de plusieurs champs, chaque champ étant identifié et accessible par un nom. On peut citer par exemple :

— les structures en C

```
struct point {  int x;  int y; };
```

— les enregistrements en caml (c'est essentiellement la même chose, sauf que par défaut les structures caml ne sont pas mutables.)

```
type point = { x: int; y: int; }
```

— les objets en java

```
class Point {
  public final int x;
  public final int y;
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, `point` désigne un nouveau type de données, `x` et `y` désignent des noms de champs de ce type de données, et les occurrences de `int` désignent le type du contenu associé aux champs `x` et `y`.

Manipulation de structures. On peut représenter une structure en mémoire par un bloc de plusieurs mots mémoire consécutifs, où les valeurs des champs sont stockées l'une après l'autre.

x	y	z	b
1	2	42	true

Dans un type de structure donné, on place systématiquement les champs dans le même ordre. Ainsi, chaque nom de champ peut être associé à une position, et le compilateur traduit chaque accès à un champ identifié par son nom en un accès à la position correspondante dans le bloc mémoire.

x ↦ 0 (décalage 0 octets)
y ↦ 1 (décalage 4 octets)
z ↦ 2 (décalage 8 octets)
b ↦ 3 (décalage 12 octets)

On crée une telle structure en réservant un bloc mémoire de taille suffisante, et en initialisant éventuellement les champs (le détail dépendant du langage). En particulier :

— En caml, on note

```
let p = { x=1; y=2; } in ...
```

la création d'une nouvelle structure `point` dont les champs `x` et `y` sont initialisés respectivement avec 1 et 2. Le bloc est créé dans le tas, ce qui implique une allocation de type `malloc`. La valeur de la variable `p` de type `point` est ensuite concrètement un pointeur vers le bloc alloué dans le tas. On accède au champ `x` de `p` avec la notation `p.x`.

— En java, la situation est similaire à celle de caml : on crée une structure (un objet) avec

```
Point p = new Point(1, 2);
```

Cette opération alloue un bloc sur le tas, et définit `p` comme un pointeur vers ce bloc. On accède au champ `x` de `p` de même avec `p.x`.

— En C, on alloue l'espace pour une structure sur le tas à l'aide de `malloc`.

```
point *p = malloc(sizeof(point));
```

L'adresse du bloc explicitement alloué par `malloc` est ensuite explicitement stockée dans une variable `p` de type `point*`, c'est-à-dire un pointeur vers une structure de type `point`. On accède au champ `x` de la structure pointée par `p` avec la notation `p->x`. On peut donc initialiser ensuite la structure avec les instructions supplémentaires

```
p->x = 1;
p->y = 2;
```

À noter, C admet également la notation utilisée en caml mais avec une signification légèrement différente : `point p = { x=1; y=2; };` place la structure sur la pile plutôt que sur le tas, et la variable `p` désigne directement cette structure et non un pointeur. Et dans ce cas, on accède au champ `x` avec la même notation `p.x`.

Autres formes de types définies par l'utilisateur. Voici quelques autres formes courantes de types définis par l'utilisateur. Une ***énumération*** est un type défini par un ensemble fini de symboles. On trouve le concept aussi bien en C avec le mot-clé `enum`

```
enum tete {
  valet; dame; roi;
}
```

qu'en caml avec un type algébrique où tous les constructeurs sont constants.

```
type tete =
  Valet | Dame | Roi
```

Pour traduire un programme utilisant des énumérations vers un langage de plus bas niveau, on numérote les symboles de chaque énumération.

$$\begin{aligned} \texttt{valet} &\mapsto 0 \\ \texttt{dame} &\mapsto 1 \\ \texttt{roi} &\mapsto 2 \end{aligned}$$

Il suffit alors de traduire dans le programme chaque occurrence d'un tel symbole par son numéro.

Un peu plus riche, un type ***union*** décrit pour un type de données une alternative entre plusieurs formats. En C par exemple, on déclare avec

```
union s {
  point p;
  int   n;
};
```

un type `s` désignant une donnée qui peut être soit un point soit un entier. On trouve un concept similaire en caml avec la définition de type algébrique

```
type s =
  | P of point
  | N of int
```

Nuance entre ces deux versions : en caml on introduit impérativement des *constructeurs*, ici les symboles `P` et `N`, qui permettent de distinguer les différents formats. L'opération de filtrage permet alors, partant d'une donnée concrète de type `s` en caml, de tester sa forme. En C en revanche, la représentation d'une donnée de type `s` ne permet pas par défaut de distinguer si à l'intérieur se trouve un point ou un entier. Il faut ajouter explicitement ces informations dans la structure, par exemple à l'aide d'une énumération des différents formes possibles.