Université de Montréal

Gestion manuelle et sécuritaire de la mémoire en Typer

par

Simon Génier

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

31 décembre 2022

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Gestion manuelle et sécuritaire de la mémoire en Typer

présenté par

Simon Génier

a été évalué par un jury composé des personnes suivantes :

Sébastien Roy
(président-rapporteur)

Stefan Monnier
(directeur de recherche)

Marc Feeley
(membre du jury)

Résumé

Dans ce mémoire, je présente une technique pour combiner du code de bas niveau à un langage purement fonctionnel avec types dépendants. Par *code de bas niveau*, je veux dire n'importe quel programme écrit dans un langage qui permet le contrôle direct des ressources de la machine. En particulier, ce texte s'intéresse à la gestion de la mémoire.

Plus concrètement, un programmeur C contrôle l'endroit et le moment où un bloc de mémoire est alloué, ainsi que la façon dont l'objet est initialisé. Par exemple, on peut allouer de l'espace sur la pile pour immédiatement l'initialiser avec un `memcpy`. Alternativement, on peut allouer un bloc sur le tas et l'initialiser champ par champ plus tard. Pour certaines applications où la mémoire est limitée ou la performance importante, ce choix est important.

Un tel niveau de contrôle n'est pas disponible dans les langages de haut niveau, sauf pour quelques exceptions. Les langages fonctionnels comme OCaml ou Haskell découragent ou même interdisent de modifier les champs d'un objet. C'est encore plus vrai pour les langages à types dépendants où la mutation est l'éléphant dans le magasin de porcelaine de la cohérence. C'est un choix de design intentionnel. Un programme pur est plus facile à comprendre et analyser, mais comment séparer l'initialisation de l'allocation quand un objet ne peut pas changer ?

Ce mémoire essaie de démontrer que ce n'est pas parce que c'est impossible, mais parce que ces langages ne sont pas habituellement utilisés dans un contexte où c'est nécessaire. Par contre, ce n'est pas facile non plus. Pour garantir la sécurité et la cohérence, il faut modéliser l'état d'un objet partiellement initialisé au niveau des types, ce que la plupart des langages ont de la peine à faire. La combinaison du manque de nécessité et de la lourdeur syntaxique et conceptuelle est la raison pour laquelle cette fonctionnalité est souvent absente.

Pour y parvenir, nous prenons un langage à types dépendants, Typer, et nous y ajoutons le nécessaire pour récupérer une partie du contrôle abandonné dans le design original. Nous permettons au programmeur d'allouer des blocs de mémoire et de les initialiser graduellement plus tard, sans compromettre les propriétés de sécurité du programme. Concrètement, nous utilisons les monades, un concept de la théorie des catégories déjà utilisé pour la modélisation d'effets de bord, pour limiter les mutations aux endroits sécuritaires.

Mots-clés : Types dépendants, Types alias, Sécurité mémoire

Abstract

In this thesis, I will demonstrate how to combine low-level code with dependent types. By *low-level code*, I mean any program authored in a language that allows direct control of computer resources. In particular, this text will focus on memory management.

Specifically, a C programmer has control over the location and time when a block of memory is allocated, as well as how it is initialized. For instance, it is possible to allocate stack space to immediately initialize it with an invocation of `memcpy`. Alternatively, one can allocate heap spate and initialize it field by field later. For some applications where memory is constrained or performance is important, this choice can matter.

This level of control is not available in high-level languages, barring a few exceptions. Functional languages such as OCaml or Haskell discourage or simply forbid the mutation of objects. This is especially the case in dependently typed languages where mutation is the bull in the china shop of consistency. This is a deliberate choice as a pure program is easier to understand and reason about. However, having allocation and initialization done in two distinct steps seems impossible in this situation.

This thesis shows that this is not impossible, it is simply done this way because these kinds of languages are seldom used in a context where such control is necessary. This does not mean though that adding this feature is easy. If we are to guarantee both safety and consistency, we need to keep track of the initialization state at the type level. Most languages struggle to do this. Language designers simply forgo this feature because it is not useful to them in addition to being difficult to use.

To achieve it, we start with a dependently typed language, Typer, and add back the mechanisms necessary to recover the control relinquished in the original design. We let the programmer allocate blocks of memory to initialize later, without compromising the safety properties of the program. Specifically, we use monads, a concept from category theory, a know technique to model side effects, to limit mutation to situations where it is safe.

Keyword: Dependent types, Alias types, Memory safety

Table des matières

Résumé 5

Abstract 7

Chapitre 1. Introduction 13

1.1. Sécurité de la mémoire 13

1.2. Deux approches à la sécurité de la mémoire 14

1.3. Vérification du C 15

1.3.1. Accès par un pointeur 15

1.3.2. Expressivité des conditions 16

1.4. Typer 17

1.4.1. Programmation fonctionnelle 18

1.4.2. Types dépendants 19

1.5. Vérification en Typer 21

1.5.1. Modélisation du tas 21

1.5.2. Triplets de Hoare 22

1.5.3. Opérations sur le tas 23

1.6. Composition 24

1.7. Modification du tas 25

1.8. Contributions 27

Chapitre 2. État de l'art 29

2.1. Langage d'assemblage typé 29

2.1.1. Passage de continuation 30

2.1.2. Allocations 30

2.1.3. Fuites de mémoire 32

2.1.4. Génération de code 33

2.1.5. Évaluation . . . 36
2.2. Types linéaires . . . 37
2.2.1. Types linéaires . . . 38
2.2.2. Compteur . . . 40
2.2.3. Évaluation . . . 41
2.3. Type alias . . . 41
2.3.1. Environnement linéaire . . . 42
2.3.2. Désallocation sécuritaire . . . 43
2.3.3. Évaluation . . . 44
Chapitre 3. Typer+Mem . . . 45
3.1. Valeurs sur le tas . . . 45
3.1.1. OCaml . . . 45
3.1.2. Pony . . . 46
3.2. Survol . . . 47
3.3. Modélisation de la mémoire . . . 51
3.3.1. Valeurs . . . 51
3.3.2. Tas . . . 52
3.3.3. Allocations . . . 53
3.4. Composition des effets . . . 56
3.4.1. Composition des opérations sur le tas . . . 60
3.5. Gestion d'erreurs . . . 61
3.5.1. Transtypage . . . 61
3.5.2. Lecture et écriture . . . 64
3.6. Exportation . . . 65
3.6.1. Réflexion des uplets . . . 65
3.6.2. Exportation des valeurs . . . 67
Chapitre 4. Implantation . . . 69
4.1. Compilateur vers Scheme . . . 69
4.2. Interaction avec le ramasse-miettes . . . 70
4.3. Implantation des primitives . . . 72

Chapitre 5. Conclusion 73

5.1. Évaluation 73

5.2. Travaux futurs 74

5.2.1. Performances à la compilation 74

5.2.2. Écriture de programmes en pratique 74

5.2.3. Automatisation des preuves 74

5.2.4. Modèles de mémoire et concurrence 74

5.3. Conclusion 75

Bibliographie 77

Chapitre 1

Introduction

1.1. Sécurité de la mémoire

Voici deux programmes similaires, un écrit en Java et l'autre en C, respectivement des langages de haut et bas niveaux.

```
1 class A {
2     public int x;
3
4
5     void setX() {
6         this.x = 7;
7     }
8 }
```

```
typedef struct {
    int x;
} A;


void setX(A* this) {
    this->x = 7;
}
```

Ces deux programme définissent chacun une structure `A` avec un champ entier `x`, puis une méthode `setX` qui assigne 7 à ce champ. Ce sont des programmes assez ennuyants si tout va bien, mais regardons comment un programmeur pourrait mal utiliser `setX`.

Les détails du modèle de mémoire diffèrent entre les deux langages, mais voici quelques propriétés nécessaires, mais pas suffisantes pour que l'assignation soit valide.

— La variable `this` doit contenir un pointeur non nul.

— Il doit pointer vers un bloc de mémoire assez grand pour contenir l'objet.

— Ce bloc doit avoir le même alignement que l'objet.

En Java toutes ces propriétés sont garanties.

— Le receveur `this` est nécessairement non nul à l'intérieur du corps d'une méthode. La machine virtuelle Java lance une exception avant d'y entrer s'il est nul. De plus, c'est syntaxiquement impossible de changer le receveur pendant l'exécution de la méthode.

— Les allocations ont la forme `new` C`(...)`, où C est une classe. Un bloc de mémoire de la bonne taille et du bon alignement est alloué et C est enregistrée dans ce bloc. Lors d'une assignation, la machine virtuelle vérifie que la variable a le type C ou est une de ses superclasses. Java garanti donc qu'un accès à une objet par une variable sera toujours valide.

En C, c'est le contraire. On peut appeler `setX` avec n'importe quoi, un pointeur nul ou même son nombre premier préféré transtypé en pointeur. Dans ce cas, le programme aura un comportement non-spécifié (*undefined behaviour*) : tout lui est permis. Si cette erreur peut être détectée statiquement, le compilateur pourrait appliquer des optimisations qui présupposent l'absence d'erreur de mémoire et générer du code machine inattendu. Sinon, à l'exécution, le programme pourrait corrompre sa propre mémoire pour planter ou faire fuir des données confidentielles à un attaquant.

La ligne 6 en Java [1] aura toujours un bon comportement, mais celui de la ligne correspondante en C dépendra du reste du programme. La propriété d'un langage d'empêcher les erreurs de mémoire se nomme sécurité de la mémoire (*memory safety*).

1.2. Deux approches à la sécurité de la mémoire

Les approches présentés à la section précédente sont fondamentalement différentes. Java a une approche locale, compositionnelle et automatique à la sécurité de la mémoire.

— Chaque expression ou énoncé est sécuritaire.

— La composition de n'importe quels deux sous-programmes sécuritaires est aussi sécuritaire.

La preuve complète pour un sous-ensemble pur de Java est disponible dans [6].

Dans le cas de C, c'est une approche globale, non-compositionnelle et manuelle.

— Une expression ou un énoncé n'est pas nécessairement sécuritaire.

— La sécurité n'est pas préservée par la composition.

Si l'on compare ce que ces langages offrent en surface, l'approche de C est clairement inférieure ! Pourtant, en pratique, la situation n'est pas si mauvaise. Les conditions de sécurité que le langage demande sont simplement vérifiés à l'« extérieur » du langage.

— Certains compilateurs offrent des diagnostiques en plus de ceux nécessaire à une implantation conforme.

— Il existe plusieurs outils d'analyse statique qui offrent une analyse encore plus sophistiquée.

1. Quand ce texte mentionne Java, il fait référence à Java sans `sun.misc.Unsafe`, une extension qui permet d'écrire du code non sécuritaire.

— Une équipe peut mettre en place des conventions qui garantissent certaines conditions de sécurité.
— Finalement, quand les approches précédentes ne sont pas suffisantes, le programmeur peut documenter ces conditions en prose dans un commentaire.

Même avec ces complications, il est désirable d'écrire du C dans certains cas. Prenons l'exemple de calculs scientifiques qui utilisent des grosses matrices. En C, on l'alloue un bloc de mémoire, puis on peut initialiser graduellement notre matrice. En Java, avec l'approche locale, ce n'est pas permis. Le programme doit être sécuritaire à chaque étape : il est impossible de séparer l'allocation de l'initialisation parce qu'on exposerait de la mémoire non-initialisée. Le système doit faire une première passe d'initialisation qui met la mémoire à zéro avant de la passer au programme.

Un compilateur peut détecter certaines de ces doubles initialisations et les éliminer. En particulier pour Java, où le compilateur le plus populaire, Hotspot, est extrêmement sophistiqué, elles peuvent presque complètement disparaitre pour offrir une excellente performance. Rappelons-nous par contre que le but du code de bas niveau n'est pas simplement la performance, mais le contrôle. Dans certains cas, par exemple dans un programme en temps réel, l'élimination de la double initialisation doit être garantie, ce que Hotspot ne peut simplement pas offrir.

1.3. Vérification du C

Les techniques hétéroclites de la section précédente sont en fait différentes façons d'énoncer, puis de vérifier les conditions de sécurités d'un programme. Cette section présente concrètement, mais informellement, ces conditions pour le langage C.

1.3.1. Accès par un pointeur

Voici une fonction.

```
void put_char_opt(int* x) {
  if (x) putchar(*x);
}
```

Elle prend un pointeur vers un entier et, si ce pointeur est non-nul, le déréférencie et imprime le caractère dont le code correspond à la valeur de l'entier. Décrivons les conditions de sécurité à partir de l'expressions la plus à l'intérieur de l'arbre syntaxique.

x. Nous supposons que notre programme est déjà valide, donc que x est une variable qui existe dans cet environnement. Il est toujours sécuritaire de lire la valeur d'une variable et notre condition est donc $\top$, la formule trivialement vraie.

`*`x. Cette expression déréférencie un pointeur. La variable `x` doit contenir un pointeur valide vers un entier. Nous utilisons la notation $\ell \mapsto \alpha$ pour indiquer qu'une adresse ℓ pointe vers une valeur de type α valide.

$$x \mapsto \texttt{int} \land \top$$

Le $\top$ vient de la condition intérieure pour l'utilisation de x. Comme elle est toujours vraie, nous pouvons simplifier la formule.

$$x \mapsto \texttt{int}$$

`putchar(*`x`);` Comme pour l'accès à une variable, les règles internes de C garantissent déjà que l'appel à cette fonction est sécuritaire. Il n'y a pas de condition supplémentaire.

`if`. Le `if` évalue son conséquent si sa condition est vraie. Il est donc sécuritaire dans les deux situations suivantes.

— La condition P du `if` est fausse.

— La condition de sécurité P_s de sa conséquente est respectée.

En d'autres mots, si la condition est fausse, le corps du `if` n'est pas évalué, ce qui est toujours sécuritaire.

$$\neg P \lor P_s$$

Dans notre cas, P est vrai si x n'est pas nul et P_s est la condition présentée plus tôt. Nous avons donc.

$$\neg(x \neq \texttt{nullptr}) \lor x \mapsto \texttt{int}$$

On encore,

$$x = \texttt{nullptr} \lor x \mapsto \texttt{int}$$

C'est aussi la condition de sécurité de la fonction elle-même.

1.3.2. Expressivité des conditions

Est-il possible de faire mieux ? Peut-on rendre cette fonction sécuritaire dans tous les cas ? Regardons les sous-formules, est-il possible d'ajouter une autre condition pour qu'une soit toujours vraie ? La première sous-formule est $x = \texttt{nullptr}$. Ce n'est pas très prometteur parce que forcer l'utilisateur de notre fonction à toujours passer un pointeur nul la rend

inutile. Nos espoirs reposent donc sur l'autre.

$$x \mapsto \texttt{int}$$

Le défi est donc d'écrire une expression C qui prend un pointeur x et retourne `true` si et seulement si x pointe vers un entier valide. À l'évaluation, un pointeur est une simple adresse, qui ne contient pas assez d'information pour faire cette détermination. On pourrait imaginer une extension au système d'allocation, où en plus de `malloc` et `free`, il y aurait la fonction suivante :

`is_valid(void *`p`, size_t` n`);`

Elle vérifirait dynamiquement que le bloc de n octets à l'adresse p est contenu dans bloc alloué plus tôt avec `malloc` et qui n'a pas déjà été libéré avec `free`. Malheureusement, cette approche souffre de plusieurs problèmes. En pratique, accéder aux métainformations de l'allocateur avant chaque accès à un pointeur entrainerait une dégradation des performances qui remetterait en question l'utilisation d'un langage comme C.

Au delà de l'aspect pratique, cette fonction n'est simplement pas possible à écrire. En effet, l'objectif de libérer la mémoire est de la réutiliser. Or comment savoir si un pointeur est bien dérivé d'un bloc présentement alloué et non d'un autre bloc désalloué plus tôt ? Le nouveau bloc pourrait contenir une valeur d'un autre type et donc l'accès mémoire retournera quelque chose, mais qui sera certainement incorrect. Pour cette raison, de tels accès accidentels à un bloc par un pointeur vers un bloc provenant d'un autre appel à `malloc` sont considérés comme des erreurs par une interprétation moderne du modèle de mémoire du langage C [10].

Il est possible d'ajouter encore plus de métainformations, comme associer à chaque pointeur un entier unique pour distinguer deux allocations à deux moments différents, mais à la même adresse. C'est que font les outils d'analyse dynamique comme Valgrind, mais ici la dégradation de performance devient inacceptable.

Bref, il n'est pas possible d'exprimer $x \mapsto$ `int` à l'intérieur du langage C. C'est au programmeur de raisonner sur son programme pour se convaincre que la condition tiendra à l'exécution.

1.4. Typer

Jusqu'à maintenant, j'ai donné mes exemples de code de haut niveau en Java, un langage qui devrait être familier ou du moins compréhensible. Cette section est une courte introduction à Typer, un langage moins connu, mais celui que j'utiliserai dans le reste de ce mémoire.

Un lecteur qui connait déjà un langage purement fonctionnel, surtout un avec des types dépendants, peut simplement la survoler. [11] présente Typer de façon plus complète.

1.4.1. Programmation fonctionnelle

Voici une définition possible d'une liste simplement chainée en Typer.

```
type List (α : Type)
| cons (head : α) (tail : List α)
| nil;
```

Comme Typer est un langage statiquement typé, nous définissons le constructeur de types `List`, paramétrisé par un autre type α, celui de ses éléments. D'un autre point de vue, `List` est une « fonction » qui prend un type comme paramètre et retourne le type d'une liste dont le type des éléments est fixe.

On crée des valeur de type `List` α avec l'un ou l'autre des constructeurs de valeurs `nil` et `cons`. Le premier ne prend pas de paramètres et le second en prend deux.

— Le champ `head`, du type α donné en paramètre, est un élément de la liste.

— Le champ `tail` est le reste de la liste.

Un type comme celui-ci, qui peut avoir plusieurs constructeurs dont chacun peut avoir plusieurs champs, possiblement récursifs, s'appelle *type algébrique*.

Voici comment créer une liste d'entiers.

```
1 x : List Int;
2 x = cons 1 (cons 2 (cons 3 nil);
```

La ligne 1 est une *déclaration* : on nomme une variable, x, et on lui assigne un type. La ligne 2 est une *définition* où l'on assigne une valeur à notre variable, la liste qui contient les entiers 1, 2, et 3. Comme les listes sont omniprésentes, nous utiliseront une notation plus légère dans ce mémoire, soit `[1, 2, 3]`.

Voyons maintenant comment utiliser ces valeurs. La fonction `map` prend une fonction et une liste, puis applique la fonction à chacun des éléments de la liste.

```
map : (?α -> ?β) -> List ?α -> List ?β;
```

La notation α `->` β est standard dans les langages fonctionnels et signifie une fonction qui prend un paramètre de type α et retourne une valeur de type β. Les variable préfixées par un `?` sont des *métavariables*. Elles indiquent des trous à être remplis par le compilateur. Dans ce cas, Typer généralisera notre fonctions avec deux nouveaux paramètres de type.

Voici une implantation simple de notre fonction.

```
map f x = case x
| cons head tail => cons (f head) (map f tail)
| nil => nil;
```

Le `case` sert à distinguer avec quel constructeur de valeurs `list` a été créé. Pour chaque constructeur, nous avons un motif qui contient le nom du constructeur et le nom des variables auxquelles les champs seront assignées, puis une expression à évaluer. En d'autres termes, on « ouvre » la liste pour accéder à ses champs.

1.4.2. Types dépendants

Voici les dépendances entre valeurs et types que l'on retrouve dans la plupart des langages statiquement typés.

- **valeur** ← **valeur** : une simple fonction.
- **valeur** ← **type** : une fonction polymorphique, c'est-à-dire une fonction qui a un ou plusieurs paramètres qui sont des types comme la fonction `map` présentée à la section précédente.
- **type** ← **type** : un constructeur de types comme `List`.

Un programmeur aura certainement remarqué, qu'avec deux variables qui peuvent prendre chacune deux valeurs, il existe une quatrième possibilité : un type qui dépend d'une valeur. C'est cette dépendance qui donne son nom aux type dépendants. Voyons comment l'utiliser avec un exemple.

La fonction `zip` est une fonction qui combine deux listes comme une fermeture éclair. Chaque élément du résultat est une paire qui contient les deux valeurs à la même position.

```
zip [1, 2, 3] ["a", "b", "c"] = [(1, "a"), (2, "b"), (3, "c")]
```

Contrairement à `map`, où il y a une implantation unique, il y a un choix à faire quand on implante `zip`. Que faire si les listes ne sont pas de la même taille ? La seule possibilité dans un langage pur est de tronquer le résultat à la longueur de la liste la plus courte. Ignorer des résultats silencieusement est propice à créer des bogues et donc des langages comme OCaml choisissent plutôt de lancer une exception. Les types dépendants permettent d'éviter le problème.

Définissons une variante de `List`, indexée sur sa longueur.

```
type Vec (α : Type) (n : ℕ)
| vec-cons (head : α) (tail : Vec α n) : Vec α (n + 1)
```

```
| vec-nil : Vec α 0;
```

`Vec` est un type algébrique généralisé (*generalized algebraic datatype* ou gadt). On peut raffiner le type de chaque constructeur en l'écrivant à la droite du deux-points. Le constructeur `vec-nil` crée un vecteur spécifiquement de longueur 0 et `vec-cons` en crée un de 1 supérieur à sa `tail`.

Voici le type de `zip` pour une `List`.

```
zip : List ?α -> List ?β -> List (?α, ?β);
```

Voici maintenant la version pour `Vec`.

```
vec-zip : Vec ?α ?n -> Vec ?β ?n -> Vec (?α, ?β) ?n;
```

Ici, nous utilisons une métavariable n qui n'est pas un type. C'est possible parce que, dans un langage à types dépendants, les variables de types ne sont pas spéciales. Pour comprendre, réécrivons cette déclaration comme le ferait le compilateur.

```
vec-zip :
  (α : Type) ≡> (β : Type) ≡> (n : ℕ) ≡>
  Vec α n -> Vec β n -> Vec (α, β) n;
```

Un fonction dépendante peut nommer ses arguments et les utiliser dans le type des arguments subséquents, que ce soit un type ou un terme comme n. La triple flèche `≡>` indique deux choses.

- Le paramètre est implicite. Le compilateur va essayer de le remplir automatiquement. C'est toujours possible pour le programmeur de le fournir manuellement pour aider le compilateur.
- Le paramètre est effaçable. Il est utilisé pour la vérification du programme, puis est jeté avant l'exécution.

La plupart des langages statiquement typés peuvent vérifier que le type des éléments est correct, mais ici nous garantissons en plus que les deux vecteurs ont la même longueur. Voici une définition possible.

```
vec-zip l r = case (l, r)
| (vec-nil, vec-nil) => vec-nil
| (vec-cons l-head l-tail, vec-cons r-head r-tail) =>
  vec-cons (l-head, r-head) (vec-zip l-tail r-tail)
```

Comment cet exemple peut-il compiler ? Ne manque-t-il pas deux motifs, soit celui pour `vec-nil` et `vec-cons` et vice-versa ? En fait, les types des deux vecteurs sont assez raffinés pour garantir que c'est impossible. Si l est `vec-nil`, la longueur n est nécessairement 0. La longueur d'un `vec-cons` est de un supérieur à la longueur de sa `tail`, donc toujours un entier plus grand que 0. Le vecteur r, qui a la même longueur, est donc aussi un `vec-nil`. [2]

Aussi, en voyant la définition, on comprend pourquoi il est possible d'effacer le n. La seule décision est faite en observant les constructeurs des vecteurs. En particulier, on ne fait pas de branchement sur la valeur du n.

1.5. Vérification en Typer

Dans la section 1.3, j'ai présenté des exemples de condition de sécurité pour un programme C qui n'étaient malheureusement pas exprimable à l'intérieur du langage lui-même. Dans la section 1.4, j'ai présenté un langage très expressif grâce aux types dépendants. Le lecteur a certainement déjà compris que ce n'est pas un hasard.

Cette section présente informellement comment exprimer et vérifier des conditions de sécurité en Typer, la contribution principale de ce mémoire. Le chapitre 3 en donne tous les détails.

1.5.1. Modélisation du tas

Il faut créer une représentation du tas qui contient assez d'information pour exprimer les conditions de sécurité d'un programme qui alloue dynamiquement de la mémoire. Un tableau associatif dont les clés sont les adresses de ces blocs est un choix naturel, mais à quoi sont associés les adresses ? Le type de la donnée stockée à cette adresse est un choix attirant par sa simplicité.

$$\{\ell_0 \mapsto \texttt{Int}; \ell_1 \mapsto \texttt{Int} \times \texttt{Int}\}$$

La fonctionnalité la plus importante de notre système est l'initialisation graduelle de structure de données et ce n'est pas possible avec cette représentation. Comment exprimer que le 2$^{\text{e}}$ champ du bloc vers lequel ℓ_1 pointe n'est toujours pas initialisé ?

Une représentation alternative pourrait être une liste de types optionnels.

$$\{\ell_0 \mapsto [\texttt{some Int}]; \ell_1 \mapsto [\texttt{some Int}, \texttt{none}]\}$$

C'est presque suffisant. Définissons notre propre type optionnel.

2. En fait, Typer n'est pas encore assez sophistiqué pour simplement omettre les motifs absurdes. Il faut écrire à la main les branches en exploitant la contradiction pour générer une valeur du bon type, mais le résultat est le même.

```
type Val
| val-↯                 % mot non-initialisé
| val-box (α : Type); % mot initialisé avec une valeur de type α
```

En plus de rendre la lecture plus facile, ce nouveau type nous permet d'ajouter des constructeurs pour gérer quelques complications que nous allons rencontrer plus tard.

$$\{\ell_0 \mapsto [\texttt{val-box Int}]; \ell_1 \mapsto [\texttt{val-box Int}, \texttt{val-}\lightning]\}$$

1.5.2. Triplets de Hoare

Une technique classique pour la vérification de programme impératifs est la logique de Hoare. L'idée est de décorer une opération C avec deux formules logiques. L'ensemble est noté $\{P\}C\{Q\}$.

- C est énoncé dans un programme impératif, par exemple une assignation à une variable mutable.
- P exprime les préconditions, c'est-à-dire que cette formule doit être vraie avant l'évaluation de C.
- Q exprime les postconditions : si P est vraie et on évalue C, alors Q doit nécessairement tenir.

La technique consiste à représenter un programme par un graphe dirigé. Chaque opération C_i est un nœud et chaque transfert de contrôle est une flèche entre deux opérations. Cette représentation est habituellement appliquée aux programmes de bas niveau et un transfert de contrôle est donc soit simplement l'instruction suivante ou encore un `goto`. Pour vérifier un programme, il suffit de prouver que, pour chaque flèche entre C_i et C_j, la précondition P_j est la conséquence de la postconditon Q_i.

Cette technique est attirante, mais n'est pas facilement applicable. Dans *Assigning Meanings to Programs* [4], un article phare de la vérification de programmes par graphes, Floyd réussi à vérifier par induction de simples boucles, mais ne mentionne pas les allocations dynamiques. Dans *Some Techniques for Proving Correctness of Programs which Alter Data Structures* [1], Burstall montre comment étendre l'approche au programmes qui manipulent certaines structures de données de forme répétitive, comme les listes chainées.

Une solution générale viendra avec le développement de la logique de séparation dans les années 2000 [15]. Aux connecteurs bien connus comme $\wedge$ et $\neg$, respectivement la conjonction et la négation, on ajoute des connecteurs qui prennent en compte le tas. Par exemple, la conjonction de séparation de deux formules $A * B$ est prouvable si et seulement si A et B

sont prouvables dans des tas disjoints. Ce type de conjonction, en garantissant que les sous-tas soient disjoints, garanti aussi l'absence d'alias entre les tas. Sans alias, un changement à un sous-tas est contenu dans ce sous-tas et les preuves de validités peuvent être composées sans danger. Cela rend les preuves tractables pour des programmes non triviaux.

1.5.3. Opérations sur le tas

En C, il existe deux primitives principales pour manipuler le tas.

```
void* malloc(size_t n);
void free(void*);
```

La première, `malloc`, alloue un bloc de n octets. La seconde, `free` libère un bloc de mémoire. Les conditions de sécurité sont les suivantes. [7, § 7.22.3]

— L'argument de `free` doit être un pointeur retourné par `malloc` (ou `calloc` *et al.*) ou un pointeur nul.

— Un pointeur vers un bloc de mémoire alloué par un appel à `malloc` doit être libéré au maximum une fois.

Cette dernière condition peut être surprenante pour un programmeur C. La règle habituelle est de libérer exactement une fois chaque bloc. Omettre de libérer un bloc n'est pourtant pas un problème du point de vue de la sécurité. Par exemple, c'est légitime d'allouer une variable globale qui restera jusqu'à la fin du programme. Par contre, une fuite de mémoire accidentelle peut causer d'autres problèmes. J'en prendrai donc compte dans la conception de mon système.

Voyons maintenant comment adapter ces fonctions en Typer. Notre analogue de `malloc` est la fonction suivante.

```
mem-alloc :
  (n : ℕ) ->
  Mem ?ℋ Addr (λ ℓ -> heap-add ℓ (replicate n val-↯) ?ℋ);
```

L'argument n de notre fonction n'est pas une quantité en octets, mais en nombre de champs. C'est une conséquence du choix de modéliser les bloc par des listes de valeurs. De plus, tandis que `malloc` retourne directement le nouveau pointeur, `mem-alloc` retourne un objet `Mem` inspiré d'un triplet de Hoare.

(1) La première composante est la précondition. Par contre, ce n'est pas une formule arbitraire, mais l'expression d'un tas comme présenté à la sous-section 1.5.1. Comme

une allocation peut être faite peu importe l'état du tas, nous avons à cet endroit une métavariable.

(2) La deuxième composante est le type de retour de l'opération. Ici, il s'agit de `Addr`, le type des adresses.

(3) La troisième composante est la forme finale du tas. Au tas initial $?\mathcal{H}$, nous ajoutons un bloc composés de n copie de `val-`$\lightning$.

Contrairement à la présentation habituelle des triplets, nous ne permettons pas des propositions logiques arbitraires. Ceci est pourtant fait sans perte de généralité : `Mem`, en tant que type dans un langage dépendant, peut être placé dans un type qui représente une proposition. La pleine puissance de la logique sous-jacente est déplacée à l'« extérieur » du type `Mem`.

```
Mem : Heap -> (α : Type) -> (α -> Heap) -> Type;
```

On remarque que le troisième argument est une fonction. Cela permet à la description du tas après l'opération de dépendre de son résultat. C'est nécessaire pour `mem-alloc`, où l'adresse du nouveau pointeur n'est pas connue d'avance.

Pour libérer un bloc de mémoire, nous avons la fonction `mem-dealloc`.

```
mem-dealloc :
  (ℓ : Addr) ->
  (proof : ℓ ∈ ?ℋ) ->
  Mem ?ℋ Unit (λ _ -> heap-remove ℓ ?ℋ proof);
```

Ici aussi, le résultat est une opération `Mem`. Par contre, nous avons un nouvel élément : contrairement à `mem-alloc`, cette fonction ne peut pas être évaluée sécuritairement dans tous les tas. Elle prend un second argument, une preuve que l'adresse à libérer ℓ est toujours dans ce tas, *i.e.* elle n'a pas déjà été libérée.

L'utilisation de Typer pour vérifier des opération mémoire n'est pas un hasard. L'expressivité d'un langage à types dépendants nous permet ce qui est impossible dans un langage comme C, soit directement encoder les conditions de sécurité dans un type. Si une de ces conditions n'est pas respectée, il sera impossible de créer un objet preuve et la fonction qui accède à la mémoire ne pourra simplement pas être appelée.

1.6. Composition

Voici une formulation alternative des fonctions de la section précédente.

```
mem-alloc :
  (n : ℕ) ->
  (ℓ : Addr)
  × (ℋ′ : Heap)
  × ℋ′ = heap-add ℓ (replicate n val-↯) ?ℋ;
mem-free :
  (ℓ : Addr) ->
  (proof : ℓ ∈ ?ℋ) ->
  (ℋ′ : Heap) × ℋ′ = heap-remove ℓ ?ℋ proof;
```

Plutôt que de retourner un objet de type `Mem`, elles sont exprimées dans un style direct où le tas d'entrée est passé en argument et le tas de sortie est retourné. Pourquoi ne pas utiliser ce style plus simple ?

La raison est qu'elle donne trop de latitude au programmeur. Voici un programme qui est non sécuritaire parce qu'il libère deux fois le même bloc.

```
exemple =
  let (ℓ, ℋ, p) = alloc {} int in
  let ℓ∈ℋ = ... in
  let _ = free ℓ ℓ∈ℋ in
  let _ = free ℓ ℓ∈ℋ in
  ...
```

Ici, on réutilise plusieurs fois la preuve $\ell \in \mathcal{H}$ que ℓ se trouve dans le tas. Une caractéristique critique pour la sécurité de notre système est de s'assurer que un modèle du tas $\mathcal{H}$ reflète bien la réalité. On peut considérer les valeur de type `Mem` comme un paquet qui combine une opération à un modèle du tas. Notre version en style direct « ouvre » le paquet et permet au programmeur de sauvegarder et d'utiliser un état du tas qui ne correspond plus à la réalité. Un type comme `Mem` nous empêche de composer des sous-programmes de façon incorrecte.

1.7. Modification du tas

Le type `Mem` et les fonction d'allocation et de désallocation sont un bon départ, mais comment écrire un programme utile ? Je terminerai cette introduction à Typer+Mem par un court exemple.

Supposons que nous voulions initialiser un doublet d'entiers.

```
run-mem $ do
  % {}
  block <- mem-alloc 2;
  % {block ↦ [val-↯, val-↯]}
  let field-0-storable = ...;
  mem-store block 0 11 field-0-storable;
  % {block ↦ [val-box Int, val-↯]}
  let field-1-storable = ...;
  mem-store block 1 22 field-1-storable;
  % {block ↦ [val-box Int, val-box Int]}
  let block-exportable = ...;
  mem-export block block-exportable;
  % {}
```

En plus de `mem-alloc` qui a déjà été présentée, nous avons de nouvelles fonctions. La première, `run-mem` évalue une action `Mem`.

```
run-mem : Mem {} ?α {} -> ?α;
```

Il est possible d'extraire la valeur d'une opération `Mem`, tant que les tas initial et final soient vide. Que le tas soit initialement vide va de soit : nous n'avons simplement pas alloué avant le premier `mem-alloc`. Par contre, la condition sur le tas final n'est peut-être pas nécessaire. En effet, un bloc qui resterait sur le tas serait « oublié » par le `run-mem` et serait une fuite de mémoire qui, comme mentionné plus tôt, n'est pas un problème de sécurité. Néanmoins, je garderai cette formulation dans ce mémoire parce qu'une telle fuite reste un problème grave.

La fonction `mem-store` écrit une valeur en mémoire, comme le ferait une affectation en C.

```
mem-store ℓ i v proof;          ℓ[i] = v;
```

Le paramètre supplémentaire `proof` est une preuve de la validité de l'écriture en mémoire, encore une fois impossible à exprimer en C.

```
mem-store :
  (ℓ : Addr) -> (i : ℕ) -> (α : Type) ->
  (proof : Mem-storable ℓ i α) ->
  Mem ?ℋ Unit (λ _ -> heap-store ℓ i α proof ?ℋ)
```

La dernière fonction, `mem-export` prend une adresse et une preuve que l'objet qui se trouve à l'adresse est pleinement initialisé et retourne l'objet lui-même comme une valeur pure.

La définition des types de la preuve, ainsi que les termes de preuve qui se trouveraient aux lignes 5, 8 et 11 on été omis pour limiter la taille de l'exemple. Finalement, le lecteur a peut-être remarqué que les types des primitives ne fonctionnent pas tout à fait. La plupart des erreurs sont intentionnelles, pour alléger la présentation. Les vraies définitions et tous les détails se retrouvent au chapitre 3.

1.8. Contributions

Ce mémoire offre les contributions suivantes.

— J'ai créé une bibliothèque Typer qui permet la manipulation de bas niveau de la mémoire. Plus précisément, j'ai adapté l'approche des types-alias, en les implantant à l'aide d'une monade.

— J'ai étendu cette approche pour supporter les uplets (doublets, triplets, …) dépendants.

— Je l'ai aussi étendue pour permettre d'exporter les valeurs du langage de bas niveau vers celui de haut niveau. Cela permet de manipuler les valeurs plus facilement dès que le contrôle accru n'est plus nécessaire. Les conditions de sécurité pour cette opération sont non-triviales, en particulier, il faut prévoir les intéractions avec le ramasse-miettes et la traduction de types de bas niveau vers les types algébriques du langage hôte. Je considère qu'il s'agit de la contribution principale de ce mémoire.

— J'ai ajouté un générateur de code Gambit Scheme pour le compilateur Typer. En fait, avant cette contribution, Typer était un langage exclusivement interprété. Je considère que la compilation vers du code natif, possible grâce à Gambit, est essentielle pour valider mon design.

Mon extension n'utilise que les fonctionnalités de base des langages à types dépendants. Elle n'a pas nécessité de changements au langage Typer ou son système de types et elle devrait être facilement adaptable à d'autres langages comme Idris, Agda, Coq, et même Haskell qui a maintenant des extensions pour supporter les types dépendants.

Chapitre 2

État de l'art

Ce chapitre présente les travaux qui traitent de la vérification de programmes impératifs et qui ont inspiré le design de mon extension.

2.1. Langage d'assemblage typé

Un compilateur ne transforme pas directement un langage de haut niveau en langage machine. Le travail est divisé en phases, où chacune émet un langage intermédiaire que consommera la prochaine. Cela permet de modulariser le compilateur : on peut travailler par exemple sur la vérification de types sans se soucier des optimisations. Pour que ce soit le cas, nous devons vérifier que le nouveau code émit reste valide et quel mécanisme accompli cette tâche mieux qu'un système de type ? Cela a aussi l'avantage qu'une phase peut être décrite comme une transformation de type, qui peut être spécifiée formellement et vérifiée.

En pratique, les compilateurs utilisent souvent des représentations typées pour les niveaux les plus hauts, mais abandonnent cette technique pour les dernières phases. Dans « *From System F to Typed Assembly Language* » [13], Greg Morissett *et al* présentent une technique pour conserver les types jusqu'à la génération de code assembleur. Le processus entier de compilation est donc garanti, incluant les optimisations de bas niveau.

L'article présente les phases suivantes.

(1) Conversion en passage de continuation (*CPS conversion*).

(2) Conversion de fermetures (*closure conversion*).

(3) *Hoisting*.

(4) Rendre explicite les allocations.

(5) Génération de code d'assemblage.

Comme ce mémoire s'intéresse au code de bas niveau, je me concentrerai sur les deux dernière phases par souci d'espace. Par contre, pour comprendre le langage d'entrée de la phase d'allocation, je commencerai quand même par discuter brièvement des précédentes

2.1.1. Passage de continuation

Le langage d'entrée du compilateur de l'article est un langage fonctionnel de style ML. Nous y retrouvons les caractéristiques de Système F, une variante du λ-calcul avec types statiques et polymorphisme, en plus de quelques extensions pour la programmation pratique comme les nombres naturels, les uplets, et la récursion au niveau des termes.

Comme tout langage de haut niveau, les termes ont une structure en arbre. Notre langage cible, le code d'assembleur, a plutôt une structure « plate », c'est-à-dire qu'un programme est une séquence linéaire d'instructions avec des sauts. L'objectif des phases 1 à 3 est donc de convertir l'arbre d'expressions en code séquentiel.

Le résultat est λ^H. Un programme dans ce langage est constitué d'une série de blocs. Un bloc est une fonction qui ne retourne pas, mais plutôt qui en appelle une autre, sa continuation. Par exemple, une expression qui applique une fonction f à un argument x, puis le résultat à g s'écrit habituellement de la façon suivante.

$g\ (f\ x)$

Dans un style en passage de continuation, f ne retourne jamais. On passe donc g comme second argument et c'est f qui doit l'invoquer.

$f\ x\ g$

En général, écrire un programme dans un style purement CPS est laborieux. C'est pourquoi, comme dans ce cas, un tel langage est utilisé comme représentation intermédiaire. Comme discuté plus haut, nous avons une représentation séquentielle du programme qui se prête bien à être converti en `goto` ou instruction équivalente. Par contre, contrairement à un `goto` où les arguments sont implicitement passé par registres, pile, ou variable globale, nous avons le passage explicite et typé statiquement par des paramètres.

2.1.2. Allocations

En λ^H, on crée un n-uplet avec la forme syntaxique suivante.

$$\langle v_0, v_1, \ldots, v_{n-1} \rangle$$

Comme nous sommes dans une phase qui suit la conversion CPS, les éléments des uplets ne peuvent pas être des expressions arbitraires. Ce sont des valeurs, c'est-à-dire des valeurs littérales, des variables, ou autres termes qui ne nécessite pas d'appel ou de branchement.

Cette opération est relativement simple, mais pas suffisament pour être réprésenté par une seule instruction machine et il faut donc la décomposer en sous-opérations plus primitives.

En particulier, il faut séparer l'allocation de l'initialisation et considérer l'initialisation de chaque champ séparément.

Pour l'allocation, on introduit une primitive `malloc`, qui prend une séquence de types.

$$\texttt{malloc}\ [\tau_0, \tau_1, \ldots, \tau_{n-1}]$$

Quel devrait être le type de cette primitive ? Clairement, c'est un uplet de n éléments, mais quel est leur type ? Par exemple, on pourrait être tenté d'affirmer

$$\texttt{malloc}\ [\texttt{word}, \texttt{word}] : \langle \texttt{word}, \texttt{word} \rangle$$

Ce serait incorrect. On s'attend à pouvoir projeter les champs d'un uplet comme $\langle \mathbb{N}, \mathbb{N} \rangle$, mais c'est à éviter ici parce que ces champs ne sont pas encore initialisés. La solution proposée dans l'article est d'ajouter un drapeau φ à chaque champ du type d'un uplet.

$$\langle \tau_0^{\varphi_0}, \tau_1^{\varphi_1}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$$

Ce drapeau prend la valeur 0 pour un champs non-initialisé. Nous pouvons donc assigner un type à un terme `malloc`.

$$\texttt{malloc}\ [\texttt{word}, \texttt{word}] : \langle \texttt{word}^0, \texttt{word}^0 \rangle$$

Lorsqu'on assigne une valeur à un champ, la valeur du drapeau passe à 1. Formellement, cela peut être exprimé par la règle suivante.

$$\frac{\begin{array}{c}\Gamma \vdash v_0 : \left\langle \tau_0^{\varphi_0}, \tau_1^{\varphi_1}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \right\rangle \qquad \Gamma \vdash v_1 : \tau_i^{\varphi_i} \\ \Gamma, x : \left\langle \tau_0^{\varphi_0}, \tau_1^{\varphi_1}, \ldots, \tau_i^{1}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \right\rangle \vdash e \\ x \notin \Gamma \qquad 1 \leq i \leq n\end{array}}{\Gamma \vdash \texttt{let}\ x = v_0[i] \leftarrow v_1\ \texttt{in}\ e}$$

Cette syntaxe est souvent utilisée en théorie des types, mais elle mérite des explications pour le lecteur qui la rencontre pour la première fois. La ligne horizontale sépare des prémisses, en haut, de la conclusion, en bas. En autres mots, si on peut prouver que chacune des expressions du haut est vraie, on peut déduire que celle du bas l'est aussi. Cette disposition est populaire parce que les règles peuvent être « connectées » pour présenter une preuve sous forme d'arbre.

Les prémisses et les conclusions sont des jugements. Des exemples de jugements sont les assertions bien connues en mathématiques comme l'égalité $a = b$ ou ici le fait qu'un élément ne se trouve pas dans un ensemble $a \notin B$. En théorie des types, on rencontre souvent $\Gamma \vdash e : \tau$, qui indique que e a le type τ, à un endroit dans le programme où une liste Γ de

variables est disponible, nommé environnement. Finalement, nous utilisons aussi le jugement $\Gamma \vdash e$ qui indique qui e est un programme bien formé dans Γ. Ce jugement est spécifique aux programmes CPS, qui ne retournent pas de valeur. En général, un auteur peut inventer les jugements qu'il veut pour décrire son langage et, à part quelques-uns qui sont communs, il faut lire l'article pour en comprendre le sens.

Pour revenir à notre règle, la valeur initiale du drapeau du champ i n'est pas vérifiée. Cela permet la réaffectation. Cela peut être désirable, mais la raison pour laquelle c'est permis ici est qu'il n'est pas possible de l'empêcher. La référence originale vers v_0 qui présente le champ i est toujours accessible dans e. Il peut donc toujours être réinitialisé, mais pas lu par cette référence originale. C'est sécuritaire parce que l'initialisation est monotone, c'est-à-dire qu'on ne peut pas déinitialiser un champ. Tenir une référence dont le type indique qu'un champ est initialisé est une preuve suffisante que la lecture est sécuritaire, peu importe le type des autres alias.

$$\frac{\Gamma, v : \left\langle \tau_0^{\varphi_0}, \tau_1^{\varphi_1}, \ldots, \tau_i^1, \ldots, \tau_{n-1}^{\varphi_{n-1}} \right\rangle \vdash e \qquad \Gamma, x : \tau_i \vdash e \qquad x \notin \Gamma \qquad 1 \leq i \leq n}{\Gamma \vdash \texttt{let}\ x = \pi_i(v)\ \texttt{in}\ e}$$

Il est important de noter que cette technique n'ajoute pas de calculs accessoires lors de l'évaluation : ces drapeaux n'existent qu'au niveau des types et est effacé. Voici un court exemple qui introduit, puis élimine une paire d'entiers.

```
let p : ⟨word⁰, word⁰⟩ = malloc [word, word] in
let p′ : ⟨word¹, word⁰⟩ = p[0] <- 4 in
let a : word = π₀(p′) in
let p″ : ⟨word¹, word¹⟩ = p′[0] <- 6 in
let b : word = π₁(p″) in
k(a + b)
```

2.1.3. Fuites de mémoire

Ce système permet une forme de changement de type (*strong update*), soit de τ^0 vers τ^1. C'est sécuritaire parce qu'il est à sens unique et que le deuxième état, τ^1, permet strictement plus d'opérations que τ^0. Des alias vers les deux types de champs peuvent coexister, il est simplement impossible de projeter un champs des alias créés avant l'assignation. Par contre, il reste une opération à traiter : la désallocation. Une solution serait d'ajouter un troisième

état, $\varphi = 2$ pour indiquer qu'une cellule est libérée, mais cela ne fait qu'introduire des incohérences. En effet, ce nouvel état enlève la permissions de faire une opération sur les champs. Les différents types d'alias ne pourraient plus coexister, mais le langage n'offre pas de mécanisme pour les éviter cette situation.

Une autre solution est de laisser fuir la mémoire. À première vue, c'est inacceptable, mais il existe une solution générale pour attraper les fuites à l'extérieur du langage : réclamer automatiquement les blocs avec un ramasse-miettes. Comme le but du langage d'assemblage typé est de vérifier la compilation de langages de haut niveau et que ces langages sont généralement conçus pour la gestion de mémoire automatique, c'est un compromis acceptable. Par contre, cela rend cette approche inutilisable comme telle pour les langages de bas niveau.

2.1.4. Génération de code

La génération de code comme telle est assez ennuyante. Comme discuté dans la section précédente, nous avons à cette phase de la compilation que du code séquentiel et nous avons éliminé la dernière opération complexe, soit allocation et l'initialisation de uplets. Par exemple, un énoncé comme `let` z `=` x `+` y `in` est converti en une instruction d'addition et un `malloc` est converti en une instruction qui appelle une procédure dans le sous-système de gestion de mémoire.

La partie qui nous intéresse est la façon dont les types sont préservés au niveau du code d'assembleur. Les termes de forme CPS peuvent surprendre le lecteur, mais leur sémantique statique reste assez semblable à ce qu'on retrouve dans un langage fonctionnel moyen, avec des fonctions qui opèrent sur d'autres fonctions, des uplets, ou des types primitifs et avec des paramètres et des variables introduites par des `let`s.

Ce n'est pas le cas dans du code d'assembleur, où il n'y a pas de variable, mais des registres. Ce sont deux « conteneur » de valeur, mais il y a une différence importante. Il n'y a pas de limite au nombre de variable et une variable a un type fixe. C'est le contraire pour les registres, dont le nombre est limité et où le même registre peut contenir à un moment un simple entier et plus tard l'adresse d'une procédure. Notre système de type doit cela.

La technique que présente l'article est de maintenir un modèle de l'ensemble des registres et du tas. Nous remplaçons l'environnement Γ, qui est une liste qui associe un nom à un type, par un triplet Ψ, Δ, Γ. Δ est l'environnement qui contient les variables de type. Le lecteur peut être surpris d'apprendre que de telles variables existent encore à cette phase, mais elles jouent un rôle important qui sera expliqué plus tard. Les deux autres parties de l'environnement, Ψ et Γ, peuvent faire référence à des variables contenues dans Δ.

Ψ modélise le tas. C'est une liste qui associe chaque n adresses ℓ à un type τ.

$$\Psi = \{\ell_0 \mapsto \tau_0; \ell_1 \mapsto \tau_1; \ldots; \ell_{n-1} \mapsto \tau_{n-1}\}$$

Les adresses ici sont des étiquettes abstraites. En particulier, ce système ne supporte pas l'arithmétique de pointeur.

Finalement, Γ modélise les registres.

$$\Gamma = \{r_0 \mapsto \tau_0; r_1 \mapsto \tau_1; \ldots; r_{n-1} \mapsto \tau_{n-1}\}$$

Les r_n sont les noms des registres et les τ_n sont aussi des types. Il n'est pas nécessaire que tous les registres soient présents.

Je vais illustrer l'utilisation de ces environnements par la sémantique statique de deux instructions, `ld` et `jmp`. Voici d'abord la règle de typage pour `ld`, qui charge (*load*) le i^{e} champ d'un uplet sur le tas.

$$\frac{\Psi, \Delta, \Gamma \vdash r_s : \langle \tau_0^{\varphi_0}; \ldots; \tau_i^1; \ldots; \tau_n^{\varphi_n} \rangle \qquad \Psi, \Delta, \Gamma\{r_d \mapsto \tau_i\} \vdash I}{\Psi, \Delta, \Gamma \vdash \texttt{ld}\, r_d, r_s[i]; I}$$

Les registres r_d et r_s sont respectivement les registres de destination et de source. Les deux prémisses vérifient respectivement que

— r_s contient bien l'adresse d'un uplet dont le champ i a un type τ_i et est initialisé ;

— La séquence d'instruction I est bien typée dans un environnement comme l'environnement initial sauf que le registre r_d contient maintenant une valeur de type τ_i.

La conclusion indique que la séquence d'instruction à laquelle notre `ld` est préfixé est aussi valide.

La deuxième instruction est `jmp`, qui effectue un saut inconditionnel.

$$\frac{\Psi, \Delta, \Gamma \vdash v : \forall[].\, \Gamma' \qquad \Delta \vdash \Gamma \leq \Gamma'}{\Psi, \Delta, \Gamma \vdash \texttt{jmp}\, v} \text{ S-JMP}$$

La partie intéressante est le type du bloc, soit $\forall[].\, \Gamma'$.

Premièrement, le bloc de destination s'attend à ce que les registres soient conformes à Γ'. Le jugement $\Delta \vdash \Gamma \leq \Gamma'$ indique que les registres courants Γ ne doivent pas nécessairement être égaux, mais peuvent aussi être un sous-type des registres de destination Γ'. Le système de types de TAL ne supporte qu'une version limitée du sous-typage qui permet

— d'oublier un registre r_n, ou

— d'oublier qu'un champ d'un uplet est initialisé.

Deuxièmement, la liste [] d'argument de type du bloc doit être vide. Cela peut paraitre curieux parce que nous avons insisté plus tôt sur le fait que TAL supporte le polymorphisme.

En fait, c'est bien le cas, un bloc peut offrir une interface avec des variables de types, mais l'appeleur doit absolument les instancier avant l'appel. La règle TAPP-VAL nous permet de le faire librement.

$$\frac{\Delta \vdash \tau \qquad \Psi, \Delta, \Gamma \vdash v : \forall[\alpha_0, \alpha_1, \ldots, \alpha_{n-1}].\, \Gamma'}{\Psi, \Delta, \Gamma \vdash v[\tau] : \forall[\alpha_1, \ldots, \alpha_{n-1}].\, \Gamma'[\tau/\alpha_0]}\ \text{TAPP-VAL}$$

Voici un exemple qui explique pourquoi les auteurs ont choisi ces règles. Commençant à l'étiquette ℓ_0, ce programme charge des constantes numériques dans les registres r_0, r_1 et r_3, ainsi qu'une continuation dans r_2, avant de sauter vers ℓ_1. Ce bloc double l'argument dans r_0 avant de continuer vers l'adresse fournie dans r_2. Pour alléger la lecture de cet extrait, j'ai omis les environnements vides.

```
1  ℓ0: code[]{}.
2    mov r0, 11
3    mov r1, 22
4    mov r2, ℓ2
5    mov r3, 33
6    jmp ℓ1
7
8  ℓ1: code[α]{r0 : int, r1 : α, r2 : ∀[]. {r0 : int, r1: α}}
9    add r0, r0
10   jmp r2
11
12 ℓ2: code[]{r0 : int, r1 : int}.
13   add r0, r1
14   halt
```

Le `jmp` de la ligne 6 est bel et bien valide.

$$\dfrac{\dfrac{\bullet \vdash \texttt{int} \qquad \bullet, \bullet, \Gamma \vdash \ell_1 : \forall[\alpha].\, \Gamma''}{\bullet, \bullet, \Gamma \vdash \ell_1 : \forall[].\, \Gamma'}\ \text{TAPP-VAL} \qquad \bullet \vdash \Gamma \leq \Gamma'}{\bullet, \bullet, \Gamma \vdash \texttt{jmp}\, \ell_1}\ \text{S-JMP}$$

où $\bullet$ indique un environnement vide et

$$\Gamma = \{r_0 : \texttt{int}, r_1 : \texttt{int}, r_2 : \forall[].\{r_0 : \texttt{int}, r_1 : \texttt{int}\}, r_3 : \texttt{int}\}$$
$$\Gamma' = \Gamma''[\texttt{int}/\alpha] = \{r_0 : \texttt{int}, r_1 : \texttt{int}, r_2 : \forall[].\{r_0 : \texttt{int}, r_1 : \texttt{int}\}\}$$
$$\Gamma'' = \{r_0 : \texttt{int}, r_1 : \alpha, r_2 : \forall[].\{r_0 : \texttt{int}, r_1 : \alpha\}\}$$

En faisant le saut, on observe les deux phénomènes suivants, du point de vue du système de types.

— On oublie le registre r_3, c'est-à-dire qu'il est présent dans Γ, mais absent de Γ'.

— La type du registre r_1 reste abstrait dans le bloc ℓ_1, mais on récupère le type concret dans la continuation ℓ_2.

Si on reformule ces observations du point de vue du code de bas niveau.

— La valeur du registre r_3 est perdue. Si le bloc qui fait l'appel veut la conserver, il doit la sauvegarder sur la pile.

— La valeur du registre r_1 est préservée. Si le bloc destination veut utiliser r_1, il doit sauvegarder la valeur et la restaurer avant d'appeler la continuation.

Notre système est donc assez expressif pour décrire et vérifier les conventions d'appel. Par exemple, l'interface binaire System V est une interface populaire pour l'architecture `x86_64`. Elle spécifie qu'un registre comme `%rax` fait partie de la première catégorie (*caller saved*), tandis que `%rbx` fait partie de la seconde (*callee saved*).

2.1.5. Évaluation

Le système de type de TAL est très intéressant pour la gestion de la mémoire. En particulier, je m'inspire des aspects suivants.

— Modéliser le tas par une liste d'association entre adresses abstraites et types.

— Maintenir l'état initialisé ou non des données au niveau des types et avec la granularité d'un champ.

— Utiliser des concepts de haut niveau comme le polymorphisme pour modéliser des concepts de bas niveau comme les conventions d'appel.

Par contre, TAL reste un système assez limité. Il n'est pas possible de libérer la mémoire et un programme TAL doit donc être évalué dans un environnement avec un ramasse-miettes, ce qui n'est pas toujours possible ou désirable, surtout pour du code de bas niveau. De plus, les environnements du tas et des registres ne sont pas des constructions de première classe. Ils sont « rigides » : il doivent être exprimés par des listes littérales. Par exemple, on pourrait

vouloir utiliser une expression conditionnelle pour exprimer qu'un bloc n'existe que si un pointeur n'est pas nul, mais c'est syntaxiquement impossible.

2.2. Types linéaires

Dans une présentation plus récente de TAL [12], Morrisett suggère une version alternative de `malloc`. Plutôt que d'annoter chaque champs d'un uplet avec un drapeau, l'allocateur ne peut retourner que des blocs d'entiers initialisés avec une valeur arbitraire. On ne permet que les entiers parce que les autres types possible, notamment les pointeurs, n'ont pas de valeur arbitraire valide. Nous évitons la complexité de maintenir les drapeau, mais nous perdons la possibilité d'avoir des blocs qui contiennent autres choses que des entiers.

On peut récupérer cette possibilité en permettant les changements de type (*strong update*). Dans la sous-section 2.1.2, nous avons discuté que ces changements de types sont incompatibles avec les alias. En effet, on mettrait à jour un alias, mais les autres indiqueraient encore le vieux type, Voici un programme qui démontre comment exploiter les alias pour créer une incohérence.

```
1 let b0 : ⟨⟩ = malloc 0 in
2 let b1 : ⟨word⟩ = malloc 1 in
3 let b1' : ⟨ptr ⟨⟩⟩ = b1[0] <- b0 in
4 let _ : ⟨word⟩ = b1[0] <- 0xcafe in
5 let p : ptr word = π0(b1') in
6 k(p)
```

— À la ligne 2, on alloue un bloc d'un élément, initialement un entier arbitraire.
— On assigne le pointeur crée à la ligne 1 à l'unique champ de notre bloc.
— En passant par l'alias original, qui est toujours un uplet d'un entier, on assigne un entier arbitraire.
— On peut obtenir un pointeur invalide en projetant le champ par le second alias.

Introduire les changements de type a donc la répercussion de nous forcer à interdire les alias. Morrisett propose de séparer les pointeurs en deux classes : uniques et partagés. Comme le nom l'indique, les règles de typages garantissent que les pointeurs uniques ne peuvent pas être copiés. Une fois bloc pleinement initialisé, l'opération `commit` nous permet de le convertir en pointeur partagé. Concrètement, si le résultat du `malloc` de la ligne 2 est linéaire, il est consommé à la ligne 3 et ne peut simplement pas utilisé à la ligne 4.

2.2.1. Types linéaires

La logique linéaire a été développée par Jean-Yves Girard. Son objectif était d'avancer l'état des logiques constructives, en allant plus loin que la logique intuitionniste. En particulier il présente des connectives plus primitives, avec lesquelles on peut implanter celles bien connues comme $\to$ ou $\times$. Girard voyait dès le début des applications en informatique, dans les programmes concurrents par exemple, où la structure des preuves linéaires peut être répliquée pour modéliser des processus indépendants. [5]

C'est plus tard, dans « *Linear types can change the world!* » [17], que Philip Wadler utilise cette notion pour permettre la mutabilité dans les programmes purs. Cette section présente cette utilisation des types linéaires, mais avec une syntaxe plus légère, inspirée de [19].

L'idée est d'ajouter la sorte des qualificateurs q, qui peut être soit `unr` pour persistant (*unrestricted*) ou `lin` pour linéaire.

$$q ::= \texttt{unr} \mid \texttt{lin}$$

Les types bien connus comme $T \to U$ ou $\mathbb{N}$ deviennent des prétypes. Les types comme tels sont des prétypes annotés avec un qualificateur. Par exemple,

$$\texttt{lin}\ (\texttt{unr}\ \mathbb{N}) \to (\texttt{unr}\ \mathbb{N})$$

est le type des fonctions linéaires qui prennent un entier persistant et retournent un entier persistant.

La logique linéaire fait partie d'une plus grande classe de logiques appelées sous-structurelles. Ce terme vient du fait que ces logiques délaissent une ou plusieurs des règles structurelles traditionnelles.

— La règle de l'échange permet de réordonner les variables d'un environnement.

$$\frac{\Gamma_0, x_0 : T_0, x_1 : T_1, \Gamma_1 \vdash x : T}{\Gamma_0, x_1 : T_1, x_0 : T_0, \Gamma_1 \vdash x : T}$$

— La règle d'affaiblissement permet d'ajouter une supposition.

$$\frac{\Gamma \vdash x : T}{\Gamma, x_0 : T_0 \vdash x : T}$$

— La règle de contraction permet de fusionner deux supposition identiques.

$$\frac{\Gamma_0, x_0 : T_0, x_1 : T_0 \vdash x : T}{\Gamma_0, x_2 : T_0 \vdash (x : T)[x_2, x_2/x_0, x_1]}$$

Un langage linéaire interdit l'affaiblissement et la contraction.

— L'affaiblissement « oublie » une valeur et permet la fuite d'une ressource, un bloc de mémoire qui n'est pas libéré ou un fichier qui n'est pas fermé.
— La contraction « duplique » une valeur. Cela permet la réutilisation qui est problématique comme démontré à la section précédente.

D'autres combinaisons sont utiles. Par exemple, Rust n'interdit que la contraction, pour donner un système qu'on appelle *affine* plutôt que linéaire. Les duplications de ressources, une chaine de caractère qui contient un bloc de mémoire alloué sur le tas par exemple, doivent être explicites. En pratique, le programmeur crée une nouvelle variable à laquelle il assigne le résultat d'une fonction qui copie la ressource. De l'autre côté, l'affaiblissement est permis. On peut « oublier » une variable et le compilateur insérera automatiquement un appel au destructeur. C'est un choix de design avant tout : les concepteurs du langage ont jugé qu'il était bénéfique de montrer les allocations dans la source, mais que la verbosité des destructeurs l'aurait rendu plus difficile à lire.

Simplement supprimer les règles donne un langage trop restrictif. Les langages sous-structuraux réintègrent une forme limitée de manipulation d'environnement dans les règles non-structurelles. Les opérations permises sont choisies avec soin pour ne pas réintroduire par accident la pleine puissance des règles originales.

Le reste de cette section montre comment implanter l'application d'une fonction linéaire. Le lecteur intéressé peut consulter [19] pour une présentation complète. L'application est traditionnellement définie comme ceci.

$$\frac{\Gamma \vdash t_f : T \to U \qquad \Gamma \vdash t_a : T}{\Gamma \vdash t_f\ t_a : U}$$

Cette définition ne peut pas être utilisée avec des valeurs linéaire : elle permettrait d'en réutiliser une dans le terme de fonction t_f et en même temps dans le terme d'argument t_a. Il faut une notion $\Gamma = \Gamma_0 \circ \Gamma_1$ et qui indique que l'environnement Γ peut être séparé en Γ_0 et Γ_1.

— L'environnement vide $\bullet$ peut être triviallement séparé en deux environnements vides.

$$\bullet = \bullet \circ \bullet$$

— Une variable contenant une valeur d'un type persistant est dupliquée.

$$\Gamma, x : \texttt{unr}\ T = \Gamma_0, x : \texttt{unr}\ T \circ \Gamma_1, x : \texttt{unr}\ T$$

— Une variable contenant une valeur d'un type linéaire est soit envoyée à « gauche »

$$\Gamma, x : \texttt{lin}\ T = \Gamma_0, x : \texttt{lin}\ T \circ \Gamma_1$$

ou à « droite ».

$$\Gamma, x : \texttt{lin}\ T = \Gamma_0 \circ \Gamma_1, x : \texttt{lin}\ T$$

Finalement, l'application dans un système linéaire est comme l'application traditionnelle, mais l'environnement est séparé plutôt que dupliqué.

$$\frac{\Gamma_f \vdash t_f : T \to U \qquad \Gamma_a \vdash t_a : T}{\Gamma_f \circ \Gamma_a \vdash t_f\ t_a : U}$$

Les règles de typage des autres formes syntaxiques suivent le même raisonnement : l'environnement est séparé plutôt que dupliqué pour la vérification des sous-termes.

2.2.2. Compteur

Voici une implantation de notre compteur qui tire parti des types linéaires pour éviter une réutilisation accidentelle.

```
type lin State = state (n : unr ℕ);

make-state : unr (unr Unit -> lin State);
make-state unit = state 0;

drop-state : unr (lin State -> unr Unit);
drop-state (state _) = unit;

inc-state : unr (lin State -> lin (unr ℕ × lin State));
inc-state (state n) = (n, state (n + 1));
```

Notez que même si l'argument `State` est linéaire, le champ n de son constructeur est persistant, ce qui nous permet de le dupliquer dans `inc-state` pour à la fois retourner la valeur du compteur et le nouvel état du compteur.

Le lecteur peut s'inquiéter avec raison qu'`inc-state` fasse une allocation par appel. Heureusement, les types linéaires rendent certaines optimisations plus faciles, dont la réutilisation de mémoire. Dans tous les cas, le compilateur peut rechercher dans une fonction des paires de déallocation puis d'allocation et les éliminer. Avec des valeurs linéaires, l'utilisation est aussi une déallocation. Il y a donc beaucoup d'occasions de faire ces associations, comme dans la fonction `inc-state`, sans avoir besoin de ce fameux compilateur suffisamment sophistiqué.

Et voici un exemple de code qui doit être rejeté. La variable qui contient une valeur linéaire est utilisée à la ligne 2 et réutilisée à la 3.

```
1 let s0 = make-state unit in
2 let (n, s1) = inc-state s0 in
3 let (m, s2) = int-state s0 in
4 let () = drop-state s2 in
5 n + m
```

Sous quel prétexte peut-on rejeter ce programme ? Après la ligne 1, l'environnement est

$$\Gamma_1 = s_0 : \texttt{lin State}$$

Comme tous les `let`, le `let` de la ligne 2 contient deux termes.

— L'expression à évaluer et lier à la variable.

— Le corps du `let`.

Il faudra donc séparer notre environnement, dans ce cas c'est d'envoyer la variable s_0 dans l'une ou l'autre de ces expressions. Comme elle est présente des deux côtés, nous aurons nécessairement une branche où la variable manquera.

2.2.3. Évaluation

J'ai choisi cet article parce que le système linéaire qu'il présente est assez approchable, mais il reste plutôt limité. La raison est que les alias sont très communs dans un programme. Par exemple, si un programmeur insère un pointeur dans un tableau, il s'attend à pouvoir y accéder plus tard. Avec ces types linéaires, il faut invalider la copie dans le tableau pour permettre à une copie d'exister à l'extérieur, puis il faudra, en pur gaspillage, finalement recopier le pointeur pour le remettre dans le tableau.

En pratique, les systèmes basées sur les types linéaires ou affines permettent la création d'alias temporaires sous certaines conditions. Par exemple, Rust nomme ces alias temporaires « emprunts » (*borrows*), qui peuvent être formalisés par un concept nommé *stacked borrows* [8].

Dans un langage qui offre déjà des types linéaires, il n'est pas nécessaire d'introduire d'autres mécanismes pour la gestion de ressource comme la mémoire. Typer+Mem est une solution alternative pour les autres, utilisable sans changements aux système de type, chose qui peut être délicate en surtout en présence de types dépendants.

2.3. Type alias

Le système de type de TAL et les types linéaires sont deux approches qui permettent les changements de types, mais chacune avec leurs faiblesses.

— TAL ne permet que d'initialiser un espace mémoire. En particulier, la désallocation n'est pas possible à cause de la présence d'alias qui pointeraient vers un bloc de mémoire devenu invalide (voir 1.3.2).
— Les types linéraires permettent les changements arbitraires dont les déallocations, mais en abandonnant la possibilité de créer des alias.

Ces choix imposent des restrictions sévères. Comme discuté plus tôt, TAL ne peut qu'être évalué dans un environemment où soit un ramasse-miettes est présent, soit le processus est éphémère. Dans un système à types linéaires, il est difficile d'exprimer certaines structures de données.

Malgré l'approche différente de ces deux systèmes, c'est possible de les combiner pour contourner leurs limitations. L'article « *Alias Types* » [16] de Smith, Walker et Morrisett nous montre comment.

2.3.1. Environnement linéaire

La technique utilisé dans l'article est de conserver l'aspect de TAL ou la forme du tas est décrite au niveau des types. On y introduit de la linéarité, non pas pour les variables, mais pour le type du tas. Plus précisément, l'environnement Γ, qui associe un nom de variable à un type, est équipé des règles structurelles standard comme l'affaiblissement et la contraction. C'est plutôt l'environnement C, qui décrit le tas en associant une adresse à un type, qui n'admet pas ces deux règles.

Cette formulation introduit en fait un niveau d'indirection. En TAL, l'environnement Γ associe directement un registre au type auquel il pointe.

$$\Gamma = \{r_0 \mapsto \langle \texttt{int}, \texttt{int} \rangle\}$$

Avec les types alias l'environnement Γ indique seulement qu'une variable est un pointeur et C en donne le type.

$$\Gamma = \{x \mapsto \texttt{ptr}(\ell)\}$$
$$C = \{\ell \mapsto \langle \texttt{int}, \texttt{int} \rangle\}$$

Le type $\texttt{ptr}(\ell)$ est un type unitaire (*singleton type*), c'est-à-dire un type avec une seule valeur, ℓ. Dans les langages non-dépendants comme celui de l'article, c'est une façon de faire le lien entre un pointeur dans Γ et une adresse dans C. En effet, parce que c'est impossible de faire référence à une valeur dans un type, les types unitaires permettent d'inverser la dépendance en fixant la valeur à partir du type. Cette technique n'est pas nécessaire en Typer.

Comme les variables sont persistantes, on peut les copier.

```
let y = x in e
```

Si ce programme est évalué dans l'environnement donné en exemple plus haut, alors e sera évalué dans l'environnement suivant.

$$\Gamma' = \{x \mapsto \texttt{ptr}(\ell); y \mapsto \texttt{ptr}(\ell)\}$$
$$C = \{\ell \mapsto \langle \texttt{int}, \texttt{int} \rangle\}$$

C'est parfaitement légal d'avoir deux alias du même pointeur. Si on fait dans e un changement de type, par exemple en désallouant le bloc par la variable x et on change par le fait même aussi y. En introduisant une forme d'indirection, les types de tous les alias sont mis à jour.

2.3.2. Désallocation sécuritaire

Pour comprendre comment ce système peut être sécuritaire, je vais présenter la règle pour `free`, l'opération qui libère la mémoire.

$$\frac{\Delta, \Gamma \vdash p : \texttt{ptr}(\eta) \qquad \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_0; \ldots; \tau_{n-1} \rangle\} \qquad \Delta, C' \oplus \{\eta \mapsto \lightning\}, \Gamma \vdash_\iota \iota}{\Delta, C, \Gamma \vdash_\iota \texttt{free}\ v; \iota}$$

Cette règle utilise quelques concepts qui n'ont pas encore été présentés.

— La métavariable η peut être soit une adresse littérale ℓ ou une variable d'adresse ρ.

— L'environnement Δ est la liste des variables de types. Comme les adresses sont levés à ce niveau pour construire les types unitaires des pointeurs, il contiendra donc aussi les variables d'adresse ρ.

— La métavariable ι est une séquence d'instruction, comme en TAL. L'expression $\texttt{free}\ v; \iota$ est ι à laquelle l'instruction $\texttt{free}\ v$ est préfixée.

— Le jugement $\Delta \vdash C = C'$ indique que les deux tas C est C' sont égaux dans l'environnement Δ.

— Le jugement $\Delta, C, \Gamma \vdash_\iota \iota$ indique que la séquence d'instruction ι est bien typée.

— Le type $\lightning$ est le type d'un bloc non-alloué. Il est important de le distinguer du type $\langle \lightning; \ldots; \lightning \rangle$, qui est le type d'un bloc non-initialisé.

Cette règle permet de conclure qu'un `free` est valide sur un pointeur $v : \texttt{ptr}(\eta)$ si

— le bloc pointé est alloué ($\eta \mapsto \langle \tau_0; \ldots; \tau_{n-1} \rangle$), et

— le reste du programme, ι, considère que ce pointeur ne pointe nulle part ($\eta \mapsto \lightning$).

La première vérifie que le `free` lui-même est sécuritaire, c'est-à-dire que le bloc existe et n'a pas déjà été alloué. La seconde vérifie que le reste du programme n'accède jamais au pointeur. C'est cette dernière règle qui montre la force de cette approche : il n'est pas nécessaire de supprimer les variable dans l'environnement qui contiennent un pointeur qui est libéré comme il faudrait le faire avec les types linéaires de la section précédente.

Une autre façon de comprendre le tas C est qu'il s'agit d'une liste de permissions. Il n'est pas suffisant de posséder un pointeur $v : \texttt{ptr}(\eta)$ pour accéder au tas, il faut aussi prouver que ce η se trouve dans C et pointe vers un bloc du bon type. L'instruction `free` supprime la permission en la remplaçant par $\lightning$, une sorte de permission nulle qui ne donne droit à aucune opération.

2.3.3. Évaluation

Malgré leur flexibilité, les types alias souffrent comme les autres systèmes d'un manque d'expressivité. En particulier, la version présenté dans cet article ne supporte pas les types récursifs et il n'est donc pas possible d'exprimer des structures de données comme les listes chainées de longueur arbitraire.

Malgré tout, l'approche utilisée, un tas linéaire et des variables persistantes, est à la base de Typer+Mem. Les types dépendants sont ce qui permet de récupérer la flexibilité des langages non sécuritaires puisque le tas lui-même peut utiliser toute l'expressivité du langage des termes.

Chapitre 3

Typer+Mem

Typer+Mem, une bibliothèque pour manipuler directement la mémoire en Typer, est la principale contribution de ce mémoire. Ce chapitre présente les choix faits pour sa conception.

3.1. Valeurs sur le tas

Même si Typer+Mem est conçu comme une bibliothèque Typer, un de mes objectifs principal est qu'il soit portable. Nous devons décrire les objets en tant que séquence de bits, mais à un niveau suffisamment haut pour être portable entre plusieurs environnements. Cette sections présente un échantillon de la façon dont ces objets sont représentés.

3.1.1. OCaml

Les données sur le tas OCaml ont une représentation uniforme, appelée **bloc** [9]. Tout bloc commence par un mot d'en-tête qui contient

— la taille du bloc en mots ;

— des méta-informations pour le ramasse-miettes ;

— une étiquette qui en décrit le contenu.

L'étiquette indique le format du reste des données.

0 Un n-uplet, c'est à dire une séquence de n **valeurs**.

252 Une chaine de caractères.

253 Un nombre à virgule flottante de 64 bits.

254 Une séquence de nombres à virgule flottante de 64 bits.

Une valeur est soit un pointeur vers un autre bloc, soit un entier encodé par un décalage de un bit vers la gauche avec le bit le moins significatif remplacé par 1. Cela implique donc que les blocs doivent avoir un alignement d'au moins 2 octets pour pouvoir distinguer un pointeur vers ce bloc d'un entier.

Les variantes d'une somme ont deux représentations possibles.

— Un simple entier pour les constructeurs sans paramètres.

— Un bloc pour ceux avec paramètres. Dans l'en-tête d'un bloc, il y a de l'espace pour un petit entier qui agit comme discriminant entre les différents constructeurs.

Prenons cette définitions.

```
type t =
  | A
  | B
  | C of string
  | D of int * int
```

Le constructeur `A` est représenté de la même façon que l'entier 0, `B` comme 1, `C` par un bloc de taille 1 avec 0 comme discriminant, et `D` par un bloc de taille 2, avec 1 comme discriminant.

Cette approche a un avantage important. Avec l'encodage des entiers et les métadonnées contenue dans l'en-tête d'un bloc, les valeurs sur le tas sont autodescriptives. Du moins pour le ramasse-miettes, il n'est pas nécessaire de maintenir d'autres métadonnées. Cela simplifie son implantation et c'est en fait si avantageux que beaucoup d'autres système adoptent la même approche.

— Gambit est une implantation de Scheme. Lorsque la cible de compilation est C, une valeur est soit un entier encodé ou un pointeur vers un bloc de mémoire. Un tel bloc est composé d'un discriminant qui indique son type dynamique, suivi d'une séquence de références vers des objets Scheme, d'entiers de 8, 16, 32 ou 64 bits, ou de nombres à virgule flottante de 32 ou 64 bits selon ce type dynamique. [3]

— MRI est l'implantation principale du langage Ruby. Une valeur en Ruby, appelé `VALUE`, est aussi soit un pointeur vers un objet alloué sur le tas, soit une valeur immédiate encodée. Une caractéristique particulière à Ruby est que la plupart des objets ont la même taille dans le tas géré par le ramasse-miettes. Si un objet a besoin de plus de mémoire, par exemple une instance avec plus qu'un petit nombre de champs ou chaine de plus que quelques caractères, un second bloc est alloué avec `malloc` et sera libéré lors de la finalisation du premier bloc.

3.1.2. Pony

À l'opposé d'OCaml, Pony supporte les objets hétérogènes [2]. Un objet peut non seulement contenir des champs de différentes tailles, mais aussi un mélange de pointeurs et de valeurs immédiates. Comment le ramasse-miettes peut-il faire son travail dans de telles conditions ? À la compilation, pour chaque type, le compilateur synthétise une fonction de traçage.

Par exemple, prenons le type

$$T = a : A \times n : \text{Int64} \times b : B$$

c'est-à-dire une structure qui contient 3 champs, soit a une instance d'une classe A, n un entier signé de 64 bits, et b une instance de B. Le compilateur émet une fonction qui ressemble à [1] :

```
void T_trace(Context ctx, T obj) {
  trace(ctx, T.a, A_trace);
  // pas besoin de tracer n, un entier
  trace(ctx, T.b, B_trace);
}
```

Une autre implantation qui utilise cette technique est CPython, l'implantation la plus populaire de Python. [14] Chaque objet contient dans son en-tête un pointeur vers sa classe. Elle contient un champ, `tp_traverse`, qui joue un rôle similaire aux fonctions de traçage en Pony.

3.2. Survol

Avant de présenter les détails du design de Typer+Mem, je vais présenter un exemple, pour donner une vue d'ensemble des ses différents composants. Dans la section 1.4, j'ai présenté la fonction `map`, qui crée une nouvelle liste en appliquant une fonction f sur chacun des éléments d'une liste d'entrée $\overline{x}$.

```
map : (?α -> ?β) -> List ?α -> List ?β;
map f x̄ = case x̄
| cons head tail => cons (f head) (map f tail)
| nil => nil;
```

Cette implantation a le bon comportement, mais souffre d'un problème. Elle fait un appel récursif qui n'est pas en position terminale et consommera donc une quantité d'espace sur la pile proportionnelle à la taille de la liste. Cela peut être indésirable dans certains environnement où l'espace disponible pour la pile est limité. Par exemple, la valeur par défaut pour un processus qui utilise la bibliothèque C MUSL est de 80 kO, rendant impossible d'appeler cette version de `map` sur une liste de quelques milliers d'éléments.

1. La vraie fonction est un peu plus compliquée pour tenir compte des objets qui peuvent se trouver dans d'autres acteurs, une contrainte qui sort des limites de notre système.

La solution classique dans un langage pur est d'utiliser `map-reverse`, qui a une implantation possible avec un appel terminal en utilisant un accumulateur.

```
map-reverse f x̄ =
  let
    loop : ?;
    loop x̄ acc = case x̄
    | nil => acc
    | cons head tail => loop tail (cons (f head) acc);
  in
    loop x̄ nil
```

On peut maintenant implanter `map` en inversant le résultat de `map-reverse`.

```
map f = map-reverse id ∘ map-reverse f
```

où ∘ est la composition de fonction et `id` la fonction identité.

Cette version n'épuisera pas la taille de la pile, mais crée inutilement une copie inverse de la liste sur le tas. En fait, si on considère les activations de la version sans appel terminal de `map` comme une liste, nous avons simplement déplacé les données temporaire de la pile au tas, où l'espace n'est plus limité, mais où la désallocation est beaucoup plus chère.

Dans un langage qui permet la mutation, il est possible de créer la liste dans le bon ordre dès le début et donc d'éviter cette inefficacité. Voici une version en Typer+Mem, d'abord sans vérification.

```
1  map f x̄ = case x̄
2  | nil = nil
3  | cons head tail =
4    let
5      loop ℓ tail = match tail
6      | nil => mem-store ℓ 2 nil
7      | cons head tail => do {
8        (ℓ′, _) <- mem-alloc 3;
9        mem-store ℓ′ 0 (mem-header List cons);
10       mem-store ℓ′ 1 (f head);
11       mem-store ℓ 2 ℓ′;
12       loop ℓ′ tail
```

```
13      }
14    in
15      run-mem $ do {
16        (ℓ, _) <- mem-alloc 3;
17        mem-store ℓ 0 (mem-header List cons);
18        mem-store ℓ 1 (f head);
19        loop ℓ tail;
20        ℓ
21      };
```

Si la liste n'est pas vide, on alloue une première cellule à la ligne 16 et on initialise son `head` à la ligne 18. Ces opération préparent la récursion, où c'est la responsabilité de chaque étape d'initialiser le `tail` de la cellule précédente, soit à `nil` (ligne 6) ou à une nouvelle cellule (ligne 11).

Les opérations `mem-alloc`, `mem-store` et `run-mem` sont décrites dans le chapitre 1. Par contre, c'est la première fois que nous rencontrons `mem-header`. Cette primitive prend le nom d'un type inductif et le nom d'un constructeur de valeur de ce type et retourne la valeur d'une en-tête bien formée pour ce constructeur. On stocke la cette valeur à la cellule 0 des blocs alloués pour qu'ils puissent devenir des objets `cons` valides une fois les champs initialisés.

Voici la version complète de `map`. Les commentaires (débutant par `%`) indiquent la forme du tas.

```
1  map : (?α -> ?β) -> List ?α -> List ?β;
2  map f x̄ = case x̄
3  | nil = nil
4  | cons head tail =
5    let
6      loop :
7        (ℓ : Addr) ->
8        (tail-storable : Heap-storable ?ℋ ℓ 2 (List β)) ->
9        List α ->
10       Mem {ℓ ↦ [val-box* 1; val-box β; val-↯]; ?ℋ}
11           (List β @ ℓ)
12           ?ℋ;
```

```
13      loop ℓ tail-storable nil = do {
14        % {ℓ ↦ [val-box* 1; val-box β; val-↯]; ?ℋ}
15        mem-store ℓ 2 nil tail-storable;
16        % {ℓ ↦ [val-box* 1; val-box β; val-box (List β)]; ?ℋ}
17        mem-export ℓ ...
18        % ?ℋ
19      };
20      loop ℓ tail-storable (cons head tail) = do {
21        % {ℓ ↦ [val-box* 1; val-box β; val-↯]; ?ℋ}
22        (ℓ′, _) <- mem-alloc 3;
23        % { ℓ ↦ [val-box* 1; val-box β; val-↯]
24        % ; ℓ′ ↦ [val-↯; val-↯; val-↯]; ?ℋ }
25        mem-store ℓ′ 0 (mem-header List cons) ...;
26        % { ℓ ↦ [val-box* 1; val-box β; val-↯]
27        % ; ℓ′ ↦ [val-box* Int 1; val-↯; val-↯]; ?ℋ }
28        mem-store ℓ′ 1 (f head) ...;
29        % { ℓ ↦ [val-box* 1; val-box β; val-↯]
30        % ; ℓ′ ↦ [val-box* 1; val-box β; val-↯]; ?ℋ }
31        mem-store ℓ 2 ℓ′ ...;
32        % { ℓ ↦ [val-box* 1; val-box β; val-addr ℓ′]
33        % ; ℓ′ ↦ [val-box* 1; val-box β; val-↯]; ?ℋ }
34        proof <- loop ℓ′ tail;
35        % { ℓ ↦ [val-box* 1; val-box β; val-addr ℓ′]; ?ℋ }
36        mem-update proof;
37        % { ℓ ↦ [val-box* 1; val-box β; val-box (List β)]; ?ℋ }
38        mem-export ℓ ...;
39        % ?ℋ
40      };
41    in
42      do run-mem {
43        (ℓ, _) <- mem-alloc 3;
44        mem-store ℓ 0 (mem-header List cons) ...;
45        mem-store ℓ 1 (f head) ...;
```

```
46        proof <- loop ℓ tail;
47        mem-cast ℓ proof
48      };
```

Les termes de preuves ont été omis pour garder l'exemple relativement court.

L'introduction d'appels à `mem-export` et `mem-update` après l'appel récursif semble empêcher l'optimisation des appels terminaux. Ce n'est pas le cas parce que ces deux fonctions sont dynamiquement des non-opération. Elles ne font que modifier la forme statique du tas.

Cet exemple introduit plusieurs nouveaux concepts, mais avant de les présenter, regardons la nouvelle formulation de `mem-export`. Dans le chapitre 1, `mem-export` retourne directement la valeur avec son nouveau type. C'est adéquat quand nous voulons manipuler directement la valeur dans le monde pur, mais ce n'est pas le cas ici. Pour conserver les appels terminaux, nous devons l'écrire sur le tas *avant* l'appel récursif. Nous ne voulons pas transtyper une adresse en une valeur, mais bien transtyper une valeur dans le tas. Les valeurs de type α `@` ℓ

— sont utilisées avec `mem-cast` pour transtyper une valeur de pointeur, ou

— sont utilisées avec `mem-update` pour mettre à jour une valeur sur le tas.

Finalement, `val-box*` est une version de `val-box` qui spécifie non seulement le type d'un valeur sur le tas, mais aussi sa valeur exacte. C'est nécessaire ici parce que si l'on veut exporter une somme, il faut s'assurer que le mot d'en-tête n'est pas simplement n'importe quel entier, mais un entier spécifique.

3.3. Modélisation de la mémoire

Cette section présente les détails du modèle que nous utiliserons pour décrire l'état du tas. Les définitions sont données dans le langage Typer [11], mais sont conçues pour être facilement portable à un autre langage à types dépendants.

3.3.1. Valeurs

Plusieurs implantations présentées à la section 3.1 dont OCaml et Gambit, fonctionnent selon le principe qu'une valeur peut être immédiate ou une référence, mais encodée de façon à ce que la distinction soit accessible à l'évaluation. De plus, dans le cas d'OCaml, ces données sont toutes de la taille d'un mot machine. C'est le plus petit dénominateur commun et un bon point de départ. Reprenons la définition des valeurs données à la section 1.5.1.

```
type Val
| val-↯
```

```
| val-box (α : Type)
```

Un `val-`$\lightning$ représente une valeur non-initialisée et un `val-box` représente une valeur de type τ, soit immédiate ou derrière un pointeur selon son encodage.

Nous avons vu à la section précédente qu'il est parfois nécessaire d'initialiser un champ à une valeur qui n'est pas pleinement initialisée, donc à une adresse spécifique.

```
| val-addr (ℓ : Addr)
```

Il est important de noter que `val-addr` indique l'adresse spécifique du champ. C'est important pour pouvoir faire un `mem-update` plus tard. Il est aussi nécessaire de modéliser des valeurs spécifiques d'un type arbitraire pour supporter les étiquettes des sommes et, comme nous verrons plus tard, l'exportation de types dépendants.

```
| val-box* (α ::: Type) (x : α)
```

Le résultat d'une allocation contigüe est une séquence de valeurs que nous appelons un bloc.

```
Block = List Val;
```

3.3.2. Tas

Le tas est une collection qui associe une adresse ℓ à un bloc unique $\boldsymbol{v}$. Un tableau associatif, implanté par un arbre ou par hachage, offre les bonne propriétés, mais nous ne pouvons pas en utiliser ici. Le tas est vérifié à la compilation, quand nous ne savons pas la valeur exacte des adresses, ce qui nous empêche de normaliser notre structure.

Pour comprendre le problème, prenons comme exemple l'insertion d'une adresse dans un arbre binaire. Pour trouver le site où insérer notre valeur, il faut parcourir l'arbre de la racine vers la bonne feuille, en faisant une comparaison à chaque branche. Comme la valeur numérique de l'adresse est inconnue à ce moment, les `if` ne sont pas normalisés. Cela ne donne pas simplement l'arbre qui associe les adresses à un bloc mais, en interprétant ces `if` comme des nœuds, l'arbre de toutes les permutations de ces adresses. C'est possible de les utiliser pour faire des preuves, mais il faut explicitement gérer ces différentes possibilités à chaque fois qu'une induction sur un tas est faite.

Repartons du début, cette fois avec un type inductif qui a la structure d'une bonne vielle liste chainée.

```
type Heap
| heap-nil
```

```
| heap-cons (ℓ : Addr) (v : Block) (tail : Heap) (p ::: P)
```

On reconnait le `nil` et le `cons`. La tête/`car` est composé de deux champs, ℓ et $\boldsymbol{v}$. On ajoute aussi une proposition P qui encodera la propriété d'unicité des adresses.

Quelle sera cette proposition P ? Une adresse est unique si elle n'est pas présente dans le reste de la liste. Voici une façon de l'exprimer en Typer

```
1 type Heap-fresh (ℋ : Heap) (ℓ : Addr) =
2 | heap-fresh-empty
3   : Heap-fresh heap-nil ℓ
4 | heap-fresh-here
5   (ℓ′ : Addr)
6   (ℓ-and-ℓ′-do-not-alias : ¬(ℓ = ℓ′))
7   : Heap-fresh (heap-cons ℓ′ ?block ?ℋ′) ℓ
```

La ligne 1 définit le constructeur de notre proposition, une relation entre ses deux arguments $\mathcal{H}$ et ℓ, qui est prouvable si et seulement si ℓ n'apparait pas dans $\mathcal{H}$. Le premier constructeur de valeur (ligne 2) est le cas de base, soit une preuve qu'une adresse est toujours fraiche dans un tas vide. Le second constructeur (ligne 4), est le cas inductif. En mots, si l'adresse ℓ est fraiche dans un tas $\mathcal{H}'$, elle le sera aussi dans le tas où l'on ajoute une autre association, tant que l'adresse ℓ' de la nouvelle association est distincte de ℓ.

Nous utiliserons la notation suivante pour faciliter la lecture.

$$\{\ell_0 \mapsto t_0; \ldots; \ell_n \mapsto t_n\} \equiv (\texttt{heap-cons}\ \ell_0\ t_0\ (\ldots\ (\texttt{heap-cons}\ \ell_n\ t_n\ \texttt{heap-nil}))) \quad (1)$$

$$\{\ell_0 \mapsto t_0; \ldots; \ell_n \mapsto t_n; \mathcal{H}\} \equiv (\texttt{heap-cons}\ \ell_0\ t_0\ (\ldots\ (\texttt{heap-cons}\ \ell_n\ t_n\ \mathcal{H}))) \quad (2)$$

3.3.3. Allocations

Nous avons développé un modèle statique du tas, mais ce n'est pas très utile sans opérations dynamiques. Sans surprise, nous avons `mem-alloc` et `mem-dealloc`, mais il nous manque un concept important avant de les présenter.

Typer, comme la plupart des langages basés sur types dépendants, sont des langages purs. La pureté n'est pas nécessaire, mais rend possible d'effacer les preuves, essentiel pour avoir de bonnes performances. C'est une erreur d'effacer une expression qui a un effet de

bord. Parce que ces preuves peuvent être des expressions arbitraires, la façon la plus directe de garantir qu'une telle expression ne soit jamais effacée est de simplement l'interdire. [2]

Interdire les effets peut sembler être fatal pour une extension qui en introduit comme la notre. Pourtant, c'est une problème beaucoup plus facile à régler que de faire des preuves dans un langage impur ! Il existe plusieurs techniques, mais celle que nous utiliserons est basée sur les monades. Elle est utilisée dans des langages populaires comme Haskell. Elle consiste à voir les effets d'un autre point de vue : plutôt qu'une fonction qui effectue directement les effets, elle retourne un objet qui décrit ces effets.

Dans notre cas, cet objet a le type `Mem`.

```
Mem :
  (ℋ : Heap) ->
  (α : Type) ->
  (h : α -> Heap) ->
  Type
```

— $\mathcal{H}$ est la forme que le tas doit avoir *avant* d'exécuter l'effet.
— α est le type de retour de l'effet.
— h est la forme du tas *après* l'effet.

La partie intéressante est que h dépend de la valeur de retour. Pour comprendre pourquoi est-ce nécessaire, regardons le type de notre première primitive, `mem-alloc`.

```
mem-alloc : (n : ℕ) -> Mem ?ℋ r h
  where
    r = (ℓ : Addr) × Heap-fresh ℓ ?ℋ
    h (ℓ, p) = heap-cons ℓ b ?ℋ p
    b = replicate n val-↯
    replicate 0 _ = []
    replicate (S n) x = x :: replicate n x
```

Il y a plusieurs morceaux qui méritent une explication.
— n est la taille de l'allocation. Il faut faire attention parce que cette taille est donnée en mots et non en octets comme dans le cas de `malloc`.

2. Une raison plus importante de garantir la pureté est que l'interprétation logique d'un effet de bord peut être assez louche. Une fonction correspond à une implication en logique, mais quelle est la contrepartie d'allouer un bloc de mémoire ?

— r est le type de la valeur de retour. C'est une paire dépendante. Sans surprise, la première composante est l'adresse du nouveau bloc. La deuxième est une preuve que cette adresse est fraiche.
— h est la forme du tas finale. Il serait impossible d'exprimer la forme du tas finale sans avoir accès autant à l'adresse de retour qu'à sa preuve de fraicheur, nécessaire pour construire le nouveau tas.
— b est la valeur du bloc, qui contient n fois `val-`$\lightning$.

Introduisons les définitions suivantes pour alléger la notation.

```
FreshAddr : Heap -> Type
FrashAddr ℋ = (ℓ : Addr) × Heap-fresh ℓ ℋ)


heap-add :
  (ℋ : Heap) -> Block -> FreshAddr ℋ -> Heap
heap-add ℋ 𝒗 (ℓ, p) = Heap-cons ℓ 𝒗 ℋ p
```

— `FreshAddr` est un alias pour le paquet existentiel d'une adresse et de sa preuve de fraicheur.
— `heap-add` construit un `heap-add` d'une `FreshAddr`. L'ordre de ses arguments et différent pour être plus pratique pour l'application partielle.

Voici le type de `heap-alloc` réécrit de façon plus concis.

```
mem-alloc :
  (n : ℕ) -> Mem ?ℋ FreshAddr (heap-add ?ℋ b)
  where
    b = replicate n val-↯
```

Finalement, voici la primitive qui libère un bloc.

```
mem-dealloc :
  (ℓ : Addr) -> (p : Heap-∈ ?ℋ ℓ) ⇒ Mem ?ℋ ⊤ h
  where
    h _ = heap-remove ?ℋ ℓ
```

Comme `free`, elle prend l'adresse du bloc, mais notre version prend en plus une preuve qu'elle est toujours dans le tas. C'est suffisant pour éliminer complètement les *double-free*, une classe de bogues commune en C. Comme le nom l'indique, ce type de bogue survient lorsqu'on appelle deux fois `free` sur le même pointeur. Un lecteur naïf pourrait croire que

l'allocateur peut simplement ignorer le `free` redondant, mais ce n'est pas possible parce que les blocs de mémoire sont réutilisés. Voici un exemple qui démontre le contraire.

(1) Un bloc accessible par un pointeur ℓ_0 est alloué, puis libéré.

(2) Un nouveau bloc est alloué. Ce bloc est accessible par le pointeur ℓ_1 qui a accessoirement la même valeur numérique que ℓ_0.

(3) Le bloc à ℓ_0 est libéré une deuxième fois.

(4) La mémoire est réutilisée pour une troisième allocation, accessible par ℓ_2.

Les pointeurs ℓ_1 et ℓ_2 sont maintenant accidentellement des alias. Ils peuvent facilement briser les règles de mémoire et mener à du *undefined behaviour* si, par exemple, les objets n'ont pas des types compatibles [7, § 6.5 ¶ 7] ou si un des alias est `restrict` [7, § 6.7.3 ¶ 8]. Si, par « chance », aucune règle n'est brisée, il est facile d'imaginer que le programme aura un comportement incorrect si une valeur change quand une autre est modifiée.

3.4. Composition des effets

Dans la section précédente, nous avons présenté une façon de réifier les effet, c'est-à-dire de les représenter par un objet `Mem`. Ce n'est qu'une partie de l'approche monadique. L'autre morceau important est la façon dont se *composent* ces objets. S'ils peuvent respecter quelques règles, nous pourrons réutiliser les fonctions génériques qui opèrent sur ces monades et qui sont déjà familières aux programmeurs fonctionnels. De plus, plusieurs langages fonctionnels offrent une syntaxe spéciale, la notation `do`, qui est parfois plus facile à lire.

Pour présenter une solution possible, je vais prendre un exemple plus simple. Voici comment implanter un compteur, de façon purement fonctionnelle.

```
1 type Counter | counter (n : Int);
2
3 make-counter : Int -> Counter;
4 make-counter init = counter init;
5
6 count : Counter -> (Int, Counter);
7 count (counter n) = (n, counter (n + 1))
```

— La ligne 1 définit un type avec un seul constructeur qui contient un entier.
— Les lignes 3 et 4 définissent une fonction qui crée un nouveau compteur.
— Les lignes 6 et 7 définissent une fonction qui retourne le prochain entier du compteur.

Typer est un langage pur et la fonction `count` doit retourner un nouveau compteur plutôt que de muter celui passé en argument.

Voici une expression qui utilise le compteur.

```
let counter = make-counter unit in
let (n, counter') = count counter in
let (m, counter'') = count counter in
(n, m)
```

Le résultat est `(0, 0)`, parce que nous comptons deux fois à partir de l'état initial. Pour l'exemple, imaginons que c'est indésirable, par exemple pour implanter le compteur de façon optimisée en utilisant un espace de mémoire mutable.

Comme discuté plus tôt, la racine du problème est que le programmeur est responsable de passer le bon objet `Counter` à la bonne fonction. L'astuce est d'enlever le contrôle direct des ces objets de l'interface.

— Le sous-programme devient une composition d'une série d'opérations impures comme `count` qui opèrent sur notre compteur et d'autres fonctions.

— Une fois le sous-programme qui inclut l'entièreté de la porté du compteur est créé, on le passe à une fonction qui l'évalue.

Quel est l'équivalent d'une fonction pour une opération impure ? Prenons comme point de départ le type de `count`.

```
count : Counter -> (Int, Counter);
```

Notre opération de base retourne un entier, mais en général pourrait avoir n'importe quel type. Par exemple, un terme qui appelle le compteur puis transforme l'entier en sa représentation textuelle aurait le type `Counter -> (Text, Counter)`.

```
op : Counter -> (?β, Counter)
```

Comme une fonction, nous avons une sortie α, mais nous n'avons pas d'entrée. L'argument `Counter` n'en est pas une, c'est un argument additionnel qui nous permet de passer l'état. Ajoutons un paramètre générique.

```
op' : ?α -> Counter -> (?β, Counter);
```

Pour éviter la répétition, définissons un alias.

```
Count : Type -> Type;
Count β = Counter -> (β, Counter);
```

Ce qui nous donne une expression plus courte qui rappelle la forme d'une fonction pure.

```
fn : ?α ->        ?β;
op : ?α -> Count ?β;
```

`Count` « décore » le type de retour pour y ajouter notre état implicite.

Il est important de noter qu'il n'y a pas de distinction fondamentale entre une fonction pure et une opération sur un compteur : cette dernière est implanté par deux fonctions. Il s'agit plutôt d'un modèle de conception (*design pattern*). Même si les deux sont ultimement de simples fonctions, c'est utile de faire la distinction pour décrire la structure d'un programme.

Voyons maintenant comment utiliser ces opérations. L'opérateur `&` fait l'application inverse d'un argument à une fonction pure.

```
(&) : ?α -> (?α -> ?β) -> ?β;
x & f = f x;
```

Quel serait l'équivalent pour les opérations sur les compteurs ? Il faut faire deux changements, soit remplacer notre fonction pure `?α -> ?β` par une opération sur les compteurs et remplacer la valeur appliquée par le résultat d'une opération.

```
(&) :          ?α -> (?α ->        ?β) ->        ?β;
(>>=) : Count ?α -> (?α -> Count ?β) -> Count ?β;
```

L'implantation utilise simplement le fait que `Count ?β` est lui-même une fonction.

```
x >>= f = lambda s -> f x s;
```

Voici un exemple qui additionne les deux prochaines valeurs d'un compteur.

```
add-next-2 : Count Int;
add-next-2 =
  count >>= lambda x ->
  count >>= lambda y ->
  pure (x + y);
```

Comme `>>=` ne permet que de séquencer des opérations sur les compteurs, nous utilisons `pure`, qui transforme n'importe quelle valeur en résultat d'opération.

```
pure : ?α -> Count ?α;
pure x = lambda s -> (s, x);
```

La définition d'`add-next-2` montre la force de cette approche. Malgré sa définition simple, `>>=` permet de séparer la partie spécifique de notre fonction, qui prend les deux prochaine valeurs et les additionne, de la partie générique qui passe l'état du compteur au prochain. Le programmeur n'a pas accès explicitement à l'état et ne peut donc pas le malmener. En fait, un type comme `Count` est souvent défini comme un type algébrique plutôt qu'un alias.

```
data Count ?α | count (k : Counter -> (?α, Counter);
```

Cela permet de n'exposer que le constructeur de type sans le constructeur de valeur, pour éviter qu'un programmeur qui connait l'implantation puisse extraire l'état et mal l'utiliser.

Pour obtenir un résultat, on évalue un calcul avec `run-count`.

```
eval-count : Int -> Count ?α -> ?α;
eval-count init (count op) =
  let (x, _) = op init in
  x;
```

Le premier argument est la valeur initiale du compteur.

Il reste une remarque importante. Le lecteur a peut-être remarqué qu'il est possible de dupliquer un `Count` α. N'observerons-nous pas le même problème ? La réponse est non et voici un exemple qui illustre la façon dont nous l'évitons.

```
let next-2 =
  count >>= lambda x ->
  count >>= lambda y ->
  pure (x, y)
in
let add-next-2 = next-2 >>= lambda (x, y) -> x + y in
let mul-next-2 = next-2 >>= lambda (x, y) -> x * y in
(eval-count 0 add-next-2, eval-count 0 mul-next-2)
```

C'est possible de réutiliser `next-2`. Par contre, parce que `next-2` représente un calcul sur un état et non un état lui-même, cela ne peut pas mener à une incohérence. Lors des deux invocations de `eval-count`, `next-2` est réévalué à chaque fois.

`Count` est un exemple d'une monade. Cette structure algébrique est utile pour modéliser toutes sortes d'effets impératifs dans un langage pur [18]. L'opérateur `>>=` (prononcé *bind*) et la fonction `pure` sont des noms standards, utilisé notamment en Haskell.

3.4.1. Composition des opérations sur le tas

L'exemple de `Count` nous offre une technique utile pour séquencer les effets de bord. Comment l'appliquer pour maintenir une modèle du tas qui reflète les modifications qu'on lui apporte ? La première étape est d'abstraire le compteur pour permettre n'importe quel type d'état `S`.

```
data Count    ?α | count (k : Counter -> (?α, Counter));
data State ?S ?α | state (k : ?S      -> (?α, ?S));
```

On peut être tenté d'assembler directement `State` et `Heap` en fixant `S = Heap` pour donner une constructeur de valeur de type

```
state : (Heap -> (?α, Heap)) -> State Heap ?α;
```

Malheureusement, ce n'est pas correct. Le tas est une phase trop tard. Dans cette formulation, les valeurs de type `Heap` existent à l'évaluation du programme. Ce serait le bon choix si nous voulions valider les accès mémoire durant cette phase, mais notre objectif est de vérifier à la compilation. Reformulons `State` pour rendre explicite le type `S`.

```
data State (S : Type) ?α | state (k : S -> (?α, S));
```

Nous avons,

```
% exécution | compilation
  h          : Heap : Type
```

où `h` est un tas. Nous voulons ramener `h` à la compilation.

```
% exécution | compilation
               h : Heap : Type
```

On remplace le premier argument de `State`.

```
data Mem (h : Heap) ?α | mem (k : Unit -> ?α);
```

Nous avons maintenant une valeur spécifique de tas encodée directement au niveau des types, qui pourra être utilisée pour vérifier un programme à la compilation. Par exemple, `Mem {`$\ell \mapsto$ `Int}` est le type d'une opération qui agit sur un tas qui contient un seul entier à l'adresse ℓ. De plus, le modèle du tas est complètement effacé à l'exécution : le constructeur de valeur `mem` contient une fonction qui ne prend ni ne retourne un tas.

C'est cette étape qui nécessite un langage à types dépendants comme Typer. En effet, il n'est pas possible de ramener le tas au niveau des types dans un langage comme OCaml.

3.5. Gestion d'erreurs

Un aspect difficile dans le design d'une interface à un allocateur est la gestion des erreurs. Un logiciel écrit pour s'exécuter sur une machine à mémoire virtuelle, surtout une de 64 bits, ne verra jamais une allocation échouer. D'un autre côté, un logiciel embarqué peut très bien fonctionner en mode réel et manquer de ressource. Pour certain, il n'est même pas nécessaire de savoir qu'une allocation peut échouer, pour d'autre, gérer cette éventualité n'est pas optionnel. Une interface ne doit pas ajouter un poids inutile au premier, mais doit exposer les erreurs au second, ainsi qu'une façon de les gérer.

Le reste de ce texte suppose que nous somme dans le premier cas : un système à mémoire virtuelle de taille suffisante. Dans le cas improbable où l'allocation échouerait, par exemple en essayant d'allouer plus de 4 GO sur un système 32 bits, nous laissons l'exécution diverger.

Il est possible de modifier `alloc` pour permettre à l'utilisateur de gérer les erreurs, par exemple en retournant le résultat dans un `Option`.

```
mem-alloc-safe : (n : ℕ) -> Mem ?ℋ r h
  where
    r = Option ((ℓ : Addr) × Heap-fresh ℓ ?ℋ)
    h = maybe ?ℋ (λ (ℓ, p) -> heap-cons ℓ b ?ℋ p)
    b = replicate n val-↯
    maybe n _ none = n
    maybe _ f (some x) = f x
```

Cette version est utile lorsque la mémoire est limité, mais nous ne l'utiliserons pas dans ce texte parce qu'elle rend les programmes encore plus lourds qu'il le sont déjà.

3.5.1. Transtypage

La définition de `Heap` de la section précédente a l'avantage d'être simple, mais souffre d'un problème. En encodant nos tas par une liste chainée, l'égalité dépend accessoirement de l'ordre des adresses. Par exemple,

$$\{\ell_0 \mapsto [\mathbb{N}; \mathbb{N}]; \ell_1 \mapsto [\mathbb{N}]\} \neq \{\ell_1 \mapsto [\mathbb{N}]; \ell_0 \mapsto [\mathbb{N}; \mathbb{N}]\}$$

Ces deux expressions ne sont pas propositionnellement égales, mais représentent le même tas, c'est-à-dire qu'un programme évalué dans l'un ou l'autre de ces tas ne pourrait pas les distinguer. Est-ce que c'est simplement inélégant, ou cela peut nous empêcher d'écrire certain programmes ?

Prenons un exemple qui lit des entiers de deux sockets différents, avec comme optimisation de vérifier avec un `wait-for-multiple-objects` (aussi appelé `poll` ou `select`) quel socket est lisible en premier. Notez que `Mem+Nw` est une monade qui permet d'exécuter les effets sur les sockets en plus de nos effets sur la mémoire.

```
exemple :                                                                    1
  Socket ->                                                                  2
  Socket ->                                                                  3
  Mem+Nw ?ℋ (ℓ : FreshAddr × FreshAddr) h                                    4
exemple s₀ s₁ = do                                                           5
  i <- wait-for-multiple-objects [s₀, s₁]                                    6
  match i                                                                    7
  | 0 => do                                                                  8
      ℓ₀ <- receive-a s₀                                                     9
      ℓ₁ <- receive-b s₁                                                     10
      (ℓ₀, ℓ₁)                                                               11
  | 1 => do                                                                  12
      ℓ₁ <- receive-b s₁                                                     13
      ℓ₀ <- receive-a s₀                                                     14
      (ℓ₀, ℓ₁)                                                               15
where                                                                        16
  receive-a :                                                                17
    Socket ->                                                                18
    Mem+Nw ?ℋ FreshAddr (Heap-add ?ℋ A)                                      19
                                                                             20
  receive-b :                                                                21
    Socket ->                                                                22
    Mem+Nw ?ℋ FreshAddr (Heap-add ?ℋ B)                                      23
                                                                             24
  wait-for-multiple-objects :                                                25
    Vec ?n Socket -> Mem+Nw (Fin ?n)                                         26
```

Le type de `exemple` est le même que celui du `match`, qui est le type commun entre ses branches. Dans les deux cas, c'est un `Mem` dont la valeur de retour est la paire d'adresses des

nouveaux blocs. Le problème apparait quand on essaie de trouver la forme finale du tas h. Dans le premier cas, on ajoute d'abord le A, puis ensuite le B.

```
h (ℓ₀, ℓ₁) ≡
  Heap-cons-fresh (Heap-cons-fresh ?ℋ A ℓ₀) B ℓ₁
```

L'ordre est inversé dans le second cas.

```
h (ℓ₀, ℓ₁) ≡
  Heap-cons-fresh (Heap-cons-fresh ?ℋ B ℓ₁) A ℓ₀
```

Si $A \neq B$, les types des deux branches du `match` ne sont pas définitionellement égaux et Typer doit rejeter ce programme, qui a pourtant une sémantique dynamique bien définie et utile. Il devrait être accepté parce que les tas sont extensionnellement égaux, c'est-à-dire qu'ils associent le même type à la même adresse.

La solution a deux parties.

— Définir une relation d'équivalence entre deux tas dont les éléments ne sont différents que par l'ordre.

— Offrir une primitive qui permet de convertir librement entre les tas équivalents.

Voici la définition de la relation.

```
Heap-≃ : Heap -> Heap -> Addr -> Type
Heap-≃ (ℋ : Heap) (ℐ : Heap) =
  (ℓ : Addr) -> heap-get ℋ ℓ = heap-get ℐ ℓ
```

Si on lit la flèche dépendante comme un $\forall$, cette proposition signifie que deux tas sont équivalents si, *pour toute adresse* ℓ, on retrouve le même bloc ou la même absence de bloc.

Voici le type de l'opération de conversion, une primitive.

```
heap-cast :
  ((x : ?τ) -> Heap-≃ (?hᵢ x) (?h_f x)) ->
  Mem ?ℋ ?τ ?hᵢ ->
  Mem ?ℋ ?τ ?h_f
```

Elle permet de `heap-cast` permet de transformer le tas à la sortie d'un effet `Mem`. Or, la forme de ce tas peut dépendre du résultat. On ne connait cette valeur qu'à l'exécution du programme, mais on vérifie la conversion statiquement, d'où la nécessité de prouver que les tas sont équivalents dans tous les cas.

Une formulation équivalente est la suivante.

```
heap-cast' : Heap-≃ ?ℋ_i ?ℋ_f -> Mem ?ℋ Unit (λ unit -> ?ℋ_f)
```

Cette version prend la preuve et retourne un `Mem` prêt à être inséré dans une séquence de `>>=`. J'utiliserais la première dans le reste du texte parce qu'elle transforme un objet de type `Mem` en le prenant en argument, ce qui rend plus clair que c'est une fonction pure et non une opération qui aura des effets à l'exécution comme pourrait le suggérer la seconde version.

3.5.2. Lecture et écriture

La lecture d'une valeur en mémoire se fait en passant une preuve que notre pointeur fait bien référence au type attendu.

```
mem-load :
  (ℓ : Addr) ->
  (i : ℕ) ->
  Mem-loadable ?ℋ ℓ i ?τ ->
  Mem ?ℋ ?τ (λ _ -> ?ℋ)
```

Comme avec les types alias (section 2.3) où la présence d'une entrée dans le modèle du tas est une permission pour lire cet emplacement, une valeur de type `Mem-loadable` nous donne la permission de lire une valeur dans `Mem`. Voici sa définition.

```
gadt Mem-loadable
  (ℋ : Heap) (ℓ : Addr) (i : ℕ) (τ : Type)
| mem-loadable-box
  (ℋ : Heap) (ℓ : Addr) (b : Block) (i : ℕ) (v : Val)
  (p_ℋ ::: Heap-get ℋ ℓ = some b)
  (p_b ::: List-get b i = some (val-box τ))
: Heap-loadable ℋ ℓ i τ
;
```

La primitive pour l'écriture, `mem-store`, prend aussi comme paramètre une preuve, mais qui est plus compliquée parce qu'elle met en relation les tas initial et final.

```
mem-store :
  (ℓ : Addr) -> (i : ℕ) -> ?τ ->
  Mem-storable ?ℋ_i ?ℋ_f ℓ i ?τ ->
  Mem ?ℋ_i Unit (λ _ -> ?ℋ_f)
```

```
gadt Mem-storable
  (ℋ_i : Heap) (ℋ_f : Heap) (ℓ : Addr) (i : ℕ) (τ : Type)
| Mem-storable-box
  (ℋ_i : Heap) (ℋ_f : Heap) (ℓ : Addr)
  (b_i : Block) (b_f : Block) (i : ℕ)
  (τ : Type)
  (p_b ::: List-replace b_i b_f i (val-box τ))
  (p_ℋi ::: Heap-get ℋ_i ℓ = some b_i)
  (p_bi ::: (v_i : Val × List-get b_i i = some v_i))
  (p_ℋf ::: Heap-get ℋ_f ℓ = some b_f)
  (p_bf ::: (List-get b_f i = some (val-box τ))
: Mem-storable ℋ_i ℋ_f ℓ i τ
;
```

3.6. Exportation

Cette section présente la façon dont Typer+Mem interopère avec du code pur. Le mécanisme est basé sur `mem-export`, qui permet l'exportation de blocs sur le tas en une valeur Typer normale. Comme le reste de notre système, nous devons exprimer les conditions de sécurités par une proposition encodée dans un type. Essentiellement, il s'agit d'une relation qui tient si un uplet peut bel et bien être représenté par un bloc. Il y a malheureusement une petite complication pour exprimer cette relation en Typer : le type d'un uplet n'est pas une valeur de première classe. Il faut introduire un mécanisme de réflexion où nous créons une valeur qui est l'image de la structure du uplet. Nous pourrons alors faire nos preuves inductives sur cet objet.

3.6.1. Réflexion des uplets

Un uplet est une séquence linéaires de valeurs. Comme point de départ pour notre type inductif, que nommerons Reflect−Σ, imitons la structure d'une liste de types.

```
type Reflect-Σ =
  | reflect-Σ-cons Type Reflect-Σ
  | reflect-Σ-nil
```

Le constructeur de valeur reflect−Σ−nil représente $\top$, qui peut être aussi vu comme le tuple vide. Pour un tuple non-vide, on ajoute un reflect−Σ−cons pour chaque élément. Voyons qu'est-ce que le `?` de ce `cons` devra contenir. Pour un simple tuple sans dépendance, le type du champ est suffisant. Par exemple, le terme suivant reflète un tuple qui contient un naturel, un entier, et du texte, qu'on note $\mathbb{N} \times \mathbb{Z} \times \text{Text}$.

```
reflect-Σ-cons
  ℕ
  (reflect-Σ-cons
    ℤ
    (reflect-Σ-cons
      Text
      reflect-Σ-nil))
```

Qu'est-ce qu'il manque pour exprimer une dépendance ? Prenons une paire $n : \mathbb{N} \times$ `Fin` n, où `Fin` n est l'ensemble (*fini*) des entiers inférieurs à n. Essayons naïvement d'utiliser les constructeurs définis plus haut.

```
reflect-Σ-cons
  ℕ
  (reflect-Σ-cons
    (Fin n)
    reflect-Σ-nil)
```

Le problème est que n est une variable libre dans cette expression. Elle devrait plutôt être introduite pour permettre de faire un lien avec le naturel du premier reflect−Σ−cons. Plutôt que chaque cellule de notre liste contienne directement la suivante, laissons les contenir une fonction qui donne la suivante. Dans notre cas, ce serait

```
reflect-Σ-cons
  ℕ
  (λ n → reflect-Σ-cons
    (Fin n)
    (λ _ → reflect-Σ-nil))
```

Voici le type inductif complet.

```
type Reflect-Σ =
```

```
  | reflect-Σ-cons (α : Type) (α -> Reflect-Σ)
  | reflect-Σ-nil
```

3.6.2. Exportation des valeurs

La fonction export suit le même design que les autres fonctions.

```
mem-export :
  (ℓ : Addr) ->
  (proof : Mem-exportable ℓ ?τ ?ℋ) ->
  Mem ?ℋ (?τ @ ℓ) (λ _ -> heap-remove ℓ ?ℋ)
```

La proposition qui affirme la sécurité de cette opération est `Mem-exportable`. Sa preuve est une preuve inductive sur les champs du bloc.

— Un bloc vide est exportable en un uplet vide.

— Un bloc précédé une valeur v est exportable en un uplet précédé d'un type τ si le bloc est exportable en ce uplet et v est exportable en τ.

Finalment, on complète la preuve en montrant que le bloc commence par une valeur additionnelle, qui est la réprésentation de l'en-tête du uplet.

Pour les types dépendants, il faut conserver un liste d'obligations à décharger. Quand on passe sur un champ dépendant, on ne peut pas directement montrer que la valeur correspond au type parce que le type contient des variables libres. À la place, il faut montrer qu'il existe des valeurs pour lesquelles la correspondance tient et ajouter à notre liste d'obligations les preuves d'égalités pour les champs desquels le type du champ courant dépend. Quand nous continuons notre preuve inductive, nous devons décharger nos obligations pour compléter la preuve.

Chapitre 4

Implantation

4.1. Compilateur vers Scheme

Le compilateur Typer est constitué des phases suivantes.

— Deux phases d'analyse lexicale et une phase d'analyse syntaxique, qui transforment une source Typer en `sexp`s (*syntactic expressions*).

— Une phase d'élaboration des `sexp`s en `lexp`s (*lambda expressions*), la représentation intermédiaire du compilateur Typer. Cela inclu notamment l'expansion de macros et la vérification de types.

— Une phase d'effaçage, qui supprime l'information de type des `lexp`s pour donner des `elexp`s (*erased lambda expressions*).

La compilation se termine à ce moment, où l'interprète de `elexp`s prend le relai. Comme Typer est avant tout un projet de recherche sur la métaprogrammation et les systèmes de types, un iterprète simple et facile à maintenir est le choix évident. Par contre, tester du code de bas niveau est plus difficile dans ce cas.

La solution directe est d'écrire un simulateur pour le tas et donner une sémantique opération aux primitives comme mutations sur ce tas. Cela n'est pas satisfaisant, parce que la fonctionalité la plus intéressante de mon système est l'exportation d'objets vers le monde pur, il est nécessaire de créer de vrais objets et non des simulacres.

Deux choix s'offrent à nous.

(1) Exposer le modèle de mémoire du langage hôte de l'interprète, dans ce cas OCaml, ainsi que la représentation des objets Typer en OCaml.

(2) Écrire une phase de plus pour transformer les `elexp`s en code natif.

La première solution est la plus facile, mais n'est pas très satisfaisante. Les objets sont doublement encodés, d'abord de Typer en OCaml, puis de OCaml en code natif. En particulier, une conséquence est que chaque objet a deux en-têtes, ce qui rend encore plus long et laborieux des programes qui le sont déjà.

Pour la seconde solution, est-ce vraiment difficile de compiler les `elexps` ? Si une `lexp` est un terme dans une variante du calcul des constructions, une `elexp`, où les types sont effacés, est donc un terme de λ-calcul non-typé. Il existe plusieurs langages inspirés de ce calcul, dont Scheme. Il a l'avantage de permettre une traduction presque directe parce qu'il a une syntaxe simple et reste très proche de la sémantique du λ-calcul. Il a aussi l'avantage d'avoir plusieurs implantations qui génèrent du code natif. Je tiens à noter que ce n'est pas une approche nouvelle : Idris, un autre langage à types dépendant, utilise un compilateur Scheme pour les dernières phases de la compilation.

Gambit Scheme [3] est une implantation mature de Scheme.

— Il génére du code natif en passant par du C portable, qui a un modèle de mémoire bien connu.

— Le format des objets en mémoire est celui décrit à la section 3.1 et Typer+Mem peut donc être adapté sans changements.

L'intégration s'est faite facilement. Une `elexp` correspond presque exactement à une expression Scheme, sauf pour les primitives Typer où l'on insère un fragment de Scheme prédéterminé.

4.2. Interaction avec le ramasse-miettes

Un question importante est l'interation de bloc partiellement initialisés et du ramasse-miette. Gambit utilise comme beaucoup d'autre environnement une représentation uniforme des données, où les entiers et les autres valeurs atomiques sont encodés pour être distincts des pointeurs. Les objets en mémoire sont donc autodescriptifs et le ramasse-miettes peut simplement traverser le tas à partir des racines sans avoir besoin de métainformations.

Il faut donc implanter Typer+Mem avec soin pour éviter une catastrophe. En effet, quand on dit qu'un bloc de mémoire fraichement alloué n'est pas initialisé, ce n'est pas qu'à un niveau conceptuel : sa représentation en mémoire contient des bits arbitraires. Ces valeurs peuvent contenir des motifs de bits qui ressemblent assez à un pointeur pour tromper le ramasse-miette et l'envoyer corrompre une autre partie du tas.

Les systèmes à représentation uniforme ont souvent un ou plusieurs types de données explicitement concu pour éviter le ramasse-miette. Souvent, c'est pour implanter les chaines de caractères sans avoir à encoder chacun des caractères. Gambit supporte cette struture de données, mais offre aussi des vecteurs homogènes. Le choix de structure de données pour les bloc de Typer+Mem est donc `u64vector` pour une architecture à 64 bits et `u32vector` pour une architecture à 32 bits.

Cela règle le problème décrit plus haut, mais en introduit un autre. Une autre façon de voir notre solution est qu'elle cache les données non-initialisés au ramasse-miette. Malheureusement, par le fait même, elle lui cache aussi les donnés valides. Pour les valeurs immédiate comme les entiers, c'est tout à fait correct. Pour les pointeurs vers d'autres objets sur le tas, cela peut s'avérer un problème grave. Si une passe de collection est faite quand la seule référence à un objet est dans notre bloc, il sera libéré et la référence deviendra invalide. Plusieurs alternatives s'offrent à nous.

(1) Initialiser tous les champs à une valeur immédiate, 0 par exemple, et permettre au ramasse-miettes d'inspecter les blocs.

(2) Désactiver le ramasse-miettes lors de l'évaluation d'opérations dans `Mem`.

(3) Maintenir une liste de racines.

La première solution peut paraitre contrintuitive. Pourquoi concevoir un système assez compliqué pour permettre l'initialisation graduelle si l'environnement est pour quand même l'initialiser d'un coup, en pure perte ? Oui, il y a un peu d'effort qui est gaspillé, mais rappellons nous l'exemple de `map` à la section 3.2, où l'on évite complètement la création d'une liste intermédiaire. Mettre une région de mémoire à 0 est extrêmement rapide sur une machine moderne. Nous payons un petit prix, mais conservons notre capacité à faire des mutation dans un langage pur. La version plus efficace de `map` reste exprimable.

La deuxième solution est acceptable dans une des deux conditions suivante.

— S'il manque de mémoire, on peut en ajouter plutôt que de libérer des objets.

— On demande au programmeur de spécifier la quantité de mémoire libre nécessaire avant de désactiver le ramasse-miette.

Cette solution est peut-être la moins élégante, mais elle a l'avantage d'être la plus simple. Elle fonctionne le mieux quand les `Mem`s sont courts à évaluer.

La troisième solution consiste à conserver une table inaccessible au programmeur. Elle a structure est similaire à `Mem`, c'est-à-dire un tableau associatif dont les clés sont les adresses des blocs. Par contre, les valeurs ne sont pas un modèle du tas, mais une liste de racines. À chaque évaluation de `mem-store`, si la valeur écrite est un pointeur, on l'ajoute à cette liste. Si l'adresse est exportée ou libérée, on supprime la liste de racine associée. L'important est que ce modèle du tas soit un objet normal, que le ramasse-miettes pourra inspecter. Cette solution est avantageuse si les données à initialiser graduellement sont surtout composées de valeurs immédiates. Par exemple, pour une grande matrice de nombres à virgule flottante, il n'y aura aucune racine à conserver et aucune perte de performance.

Voyons comment se comporte notre extension dans des environnements avec des designs moins courants. Pour un système comme Pony où chaque objet a une fonction de traçage, on peut mettre à jour la fonction de traçage à mesure que l'objet est initialisé. Si on peut garantir de ne pas être interrompu par le ramasse-miettes, il est possible de regrouper plusieurs `mem-store`.

Je crois que le meilleur environnement pour Typer+Mem en est un où la mémoire est collectée par compte de référence, comme Swift. En effet, il n'est pas nécessaire de cacher des données au ramasse-miette parce qu'il n'y en a simplement pas ! La version de `mem-dealloc` présenté plus tôt peut causer des fuites de mémoire. On peut la corriger en reforçant la précondition en exigeant que le bloc à libérer ne contienne pas de pointeurs.

4.3. Implantation des primitives

Sachant que notre choix pour la représentation en mémoire des bloc est un vecteur uniforme comme `u64vector`, l'implantation des primitives est assez directe. En effet, elles ont une sémantique statique riche pour permettre d'exprimer les conditions de sécurité, mais leur sémantique dynamique est assez simple.

```
mem-alloc                 make-u64vector
mem-dealloc               (lambda (x) nil)
mem-export                (lambda (x) x)
mem-load                  (lambda (ℓ)
                            (lambda (i)
                              (u64vector-ref ℓ i))))
mem-store                 (lambda (ℓ)
                            (lambda (i)
                              (lambda (v)
                                (u64vector-set ℓ i v))))
```

Comme nous traduisons directement les `elexp`s et que les fonctions dans un terme `elexp` sont currifiées, il faut aussi currifier nos primitives si nécessaire.

Chapitre 5

Conclusion

5.1. Évaluation

L'objectif de Typer+Mem est d'écrire du code de bas niveau vérifié. Nous avons adapté l'approche des types alias dans Typer, un langage à types dépendants. La puissance du langage permet d'exprimer directement les conditions de sécurités du programme. De ce point de vue, je considère que Typer+Mem est un succès.

Par contre, mon approche diverge des types alias d'un point de vue important : ce dernier est explicitement conçu pour être généré par un programme, et non écrit directement comme Typer+Mem. Un problème que les auteurs des types alias pouvaient ignorer était donc la difficulté d'écrire de tels programmes. Comme les exemples le témoignent, Typer+Mem est extrêmement verbeux et assembler les termes de preuve est laborieux. Ce problème est mitigé par ce que je considère la force de mon design, l'approche à la carte. Le programmeur n'a à subir la complexité de la vérification que lorsqu'il ou elle ne souhaite pas manipuler directement la mémoire. Les parties qui ne font pas d'initialisation graduelle peuvent même consommer les objets crées par Typer+Mem grâce à `mem-export`.

Les preuves consistent à montrer qu'une adresse est dans un tas où que tel type est à telle adresse. Ce ne sont pas des preuves compliquées et je crois qu'elles peuvent être presque complètement automatisées : un objectif de Typer est d'être métaprogrammable. Alternativement, on pourrait implanter `Mem` sur la base d'un autre langage. En Coq+Mem, il serait possible de concevoir une tactique pour aider la génération par des métaprogrammes en Ltac.

5.2. Travaux futurs

5.2.1. Performances à la compilation

Un question non-résolue est l'impact de l'évaluation de ces métaprogramme sur le temps de compilation. Serait-il possible d'utiliser Typer+Mem pour un « vrai » programme. Deux aspects du système mitigent les risques.

— Notre système est conçu en utilisant directement les mécanismes de preuve du langage hôte. Nous profitons donc des optimisations générales apportées au compilateur.

— Encore ici, l'approche à la carte nous permet de contourner le problème si on rencontre un cas pathologique. En effet, si une preuve est trop longue à générer, on peut simplement remplacer l'invocation du métaprogramme par son résultat.

5.2.2. Écriture de programmes en pratique

Au delà de la performance, une autre question est de développer des techniques pour faciliter l'écriture de programme de bas niveau. Dans ce mémoire, je me suis penché sur le choix de primitives et leur implantation. Il reste à déterminer quoi construire sur la base de ces primitives. Un projet intéressant est par exemple la conception d'une bilbiothèque de manipulation de vecteurs et matrices.

5.2.3. Automatisation des preuves

J'ai mentionné plus tôt que la faille principale de Typer+Mem est la lourdeur des preuves, pour offrir comme solution de fournir des métaprogrammes pour automatiser ce processus. En réalité, ce n'est pas si simple. Ces métaprogrammes comme tels ne devraient pas particulièrement être difficiles, de complexité similaire à une tactique en Ltac. Le problème est que Typer n'offre pas, à l'écriture de ce mémoire, les fonctionnalités nécessaires. En particulier, les seuls types de métaprogrammes disponibles sont des macros au niveau de la syntaxe. Il est nécessaire de pouvoir accéder à l'élaborateur, ce qui est possible, mais non trivial à ajouter.

5.2.4. Modèles de mémoire et concurrence

La version de Typer+Mem présenté dans ce texte se base sur un modèle de mémoire simple, où elle est découpée en blocs de mots. Ce modèle a l'avantage d'être portable, en ignorant complètement détails de la cache du microprocesseur. Il est donc inadéquat pour vérifier des programme ou plusieurs fils d'exécution partagent des plages de mémoire. Comme

les types dépendants nous permettent d'exprimer des propriétés arbitraires, il est possible d'étendre Typer+Mem pour prendre en compte les accès à la mémoire partagée.

Avoir un modèle de mémoire précis permettrait aussi une traduction certifiée d'un programme C. Cela permettrait de porter graduellement un programme C en un langage de haut niveau, en évitant les risque d'une réécriture.

5.3. Conclusion

On considère généralement *bas* et *haut niveau* comme les pôles opposés d'une échelle. On retrouverait C près du premier pôle et Haskell, près du second, avec Java entre les deux. Je propose plutôt un modèle où ce sont deux axes indépendants, où un langage peut avoir autant les caractéristiques des deux pôles. Typer+Mem combine de cette façon les accès directs à la mémoire aux types dépendants pour offrir à la fois un contrôle fin des ressources de l'ordinateurs et des programmes certifiés sans erreur de mémoire.

Bibliographie

[1] R. M. Burstall. “Some Techniques for Proving Correctness of Programs which Alter Data Structures”. In: *Machine Intelligence 7*. New York: American Elsevier, 1972, pp. 23–50.

[2] Sylvan Clebsch et al. “Orca: GC and Type System Co-Design for Actor Languages”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133896.

[3] Marc Feeley. *Gambit, a portable implementation of Scheme*. 2019. URL: https://www.iro.umontreal.ca/~gambit/doc/gambit.html.

[4] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science* 19 (1967), pp. 19–31.

[5] Jean-Yves Girard. “Linear logic”. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90045-4.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: A Minimal Core Calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505.

[7] ISO. *ISO/IEC 9899:2018: Information technology — Programming languages — C*. 2018. URL: https://www.iso.org/standard/74528.html.

[8] Ralf Jung et al. “Stacked Borrows: An Aliasing Model for Rust”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371109.

[9] Xavier Leroy, Damien Doligez, and INRIA Rocquencourt. *OCaml*. 2020. URL: https://github.com/ocaml/ocaml/tree/4.11.

[10] Kayvan Memarian et al. “Exploring C Semantics and Pointer Provenance”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290380. URL: https://doi.org/10.1145/3290380.

[11] Stefan Monnier. “Typer: ML boosted with type theory and Scheme”. In: *Journées Francophones des Langages Applicatifs* (2019), pp. 193–208.

[12] Greg Morrisett. “Typed Assembly Language”. In: *Advanced topics in types and programming languages*. Ed. by Benjamin C. Pierce. The MIT Press Cambridge, 2005. Chap. 1, pp. 141–175.

[13] Greg Morrisett et al. “From System F to Typed Assembly Language”. In: *ACM Trans. Program. Lang. Syst.* 21.3 (May 1999), pp. 527–568. ISSN: 0164-0925. DOI: 10.1145/319301.319345.

[14] Python Software Foundation. *Python Developer’s Guide: Garbage Collector Design*. 2022. URL: https://devguide.python.org/internals/garbage-collector/.

[15] J.C. Reynolds. “Separation logic: a logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[16] Frederick Smith, David Walker, and Greg Morrisett. “Alias Types”. In: *Programming Languages and Systems*. Ed. by Gert Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 366–381. ISBN: 978-3-540-46425-9. DOI: 10.1007/3-540-46425-5_24.

[17] Philip Wadler. “Linear types can change the world!” In: *Programming concepts and methods*. Vol. 3. 4. 1990, p. 5.

[18] Philip Wadler. “Monads for functional programming”. In: *Advanced Functional Programming*. Ed. by Johan Jeuring and Erik Meijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 24–52. ISBN: 978-3-540-49270-2. DOI: 10.1007/3-540-59451-5_2.

[19] David Walker. “Substructural type systems”. In: *Advanced topics in types and programming languages*. Ed. by Benjamin C. Pierce. The MIT Press Cambridge, 2005. Chap. 1, pp. 3–44.