Université Laval

Projet de fin d'études

MAT-3600 Hiver 2015

Catégories et métaprogrammation en C++

Auteur :
Louis Dionne

Superviseur :
Danny Dubé

26 avril 2015

Table des matières

1 Introduction

Certains langages de programmation utilisent des concepts tirés des mathématiques pour faciliter le raisonnement à propos des programmes. C'est notamment le cas de certains langages fonctionnels, qui, avec Haskell en tête, utilisent des notions de théorie des catégories pour définir des interfaces génériques encapsulant des phénomènes omniprésents en programmation comme le non-déterminisme et la propagation d'erreurs. Il en résulte des programmes qui sont généralement plus simples et surtout plus souvent corrects.

À l'opposé du spectre des langages se trouve le C++, langage typiquement associé à la programmation de bas niveau et peu reconnu pour ses qualités d'expressivité, encore moins d'esthétisme mathématique. De plus, le coeur du système de typage du C++ renferme une bête indomptée qui fut découverte Turing complète par accident peu après sa création. Il s'agit du système des *templates*, dont les créateurs n'avaient initialement pas soupçonné la puissance. Ce système forme un langage purement fonctionnel interprété par le compilateur et qui a donné naissance à la métaprogrammation statique, outil devenu omniprésent dans le C++ moderne de par ce qu'il permet d'accomplir.

L'utilisation des *templates* comme modèle de calcul étant désagréable au mieux, certaines bibliothèques comme la MPL [5] et Fusion [3] sont apparues il y a quelques années, dans le but de rendre la métaprogrammation plus accessible. Cependant, le langage C++ ayant beaucoup évolué dans les récentes années, le domaine du possible s'est vu exploser et plusieurs bibliothèques de métaprogrammation ont vu le jour, avec différents niveaux de succès.

Parmi ces bibliothèques se trouve Hana [4], une bibliothèque qui fusionne la programmation statique et la programmation dynamique en fournissant une syntaxe unifiée pour exprimer les calculs, ce qui n'avait jamais été fait auparavant. Les outils fournis par cette bibliothèque sont spécifiés dans un langage mathématique semi-formel dont les idées sont tirées du Haskell, de la théorie des catégories et parfois carrément nouvelles. Ce texte se veut à la fois une introduction à la métaprogrammation moderne en C++ et une formalisation de celle-ci à l'aide de la théorie des catégories, en utilisant la bibliothèque Hana comme exemple concret de cette interaction.

2 La métaprogrammation en C++

2.1 Le système de typage et les templates

Cette section introduit brièvement les concepts de base du système de typage du C++, qui seront nécessaires par la suite. Le lecteur qui est familier avec le C++11 et le C++14 est invité à sauter cette section.

Le C++ est un langage statiquement typé. Ceci veut dire que chaque objet possède un (et un seul) type, et que celui-ci est déterminé au moment de la compilation. De plus, en C++, le type d'un objet est directement relié à son mode de représentation. Par exemple, pour représenter une valeur entière, on doit d'abord décider d'un mode de représentation, disons `int`. Or, le mode de représentation choisit ne permet généralement pas de représenter le concept voulu dans toute sa généralité. En effet, `int` n'est capable de représenter que des valeurs entre -2,147,483,647 et 2,147,483,647, ce qui est très loin du bien plus général $\mathbb{Z}$. La différence entre le concept abstrait représenté par un type et sa représentation concrète est un problème auquel les programmeurs sont constamment confrontés. Un bon programmeur saura utiliser la bonne représentation au bon moment, et ce dans le but d'optimiser des choses qui dépendent de la situation particulière ; performance, expressivité, testabilité ou même productivité du programmeur.

Lorsqu'on souhaite créer un objet pour lui donner une valeur, on doit donc dire au compilateur quel est le mode de représentation que l'on souhaite utiliser. Par exemple, pour créer un entier représenté par le type `int` avec une valeur de 10, on écrira

```
int i = 10;
```

De manière similaire, le domaine et le codomaine d'une fonction sont des types qui doivent être connus au moment de la compilation. Ici aussi, il est nécessaire de spécifier le mode de représentation exact que l'on souhaite utiliser. Par exemple, pour déclarer une fonction dont le domaine est l'ensemble des chaînes de caractères et le codomaine est l'ensemble des entiers non-négatifs, on devra d'abord décider d'une représentation concrète pour chacun de ces ensembles. Supposons que l'on choisisse le type `std::string` pour représenter l'ensemble des chaînes de caractères et le type `unsigned int` pour représenter l'ensemble des entiers non-négatifs. Une fonction `f` : `std::string` → `unsigned int` s'écrira alors

```
unsigned int f(std::string);
```

Les créateurs du langage C++ se sont vite rendu compte que ce système, bien que plutôt simple à gérer pour le compilateur, limitait grandement l'expressivité du côté du programmeur. Par exemple, supposons que l'on ait une famille de types représentant des séquences d'objets, et que ces types aient tous une interface commune :

```
struct int_sequence {
    bool is_empty() const;
```

```
    int head() const;
    int_sequence tail() const;

    // implémentation ignorée pour simplifier
};

struct float_sequence {
    bool is_empty() const;
    float head() const;
    float_sequence tail() const;

    // implémentation ignorée pour simplifier
};

// possiblement d'autres types de séquences
```

Dans cet exemple, `is_empty` retourne une valeur booléenne qui est `true` si la séquence est vide, et `false` sinon. `head` retourne la première valeur d'une séquence non-vide et `tail` retourne une nouvelle séquence qui contient toutes les valeurs sauf la première. Il est possible de faire plusieurs choses utiles à partir de cette interface minimale. Par exemple, on peut aller chercher la n-ème valeur dans une séquence de la manière suivante :

```
int_sequence int_sequence_nth_element(unsigned int n, int_sequence s) {
    if (n == 0) return s.head();
    else        return int_sequence_nth_element(n - 1, s.tail());
}
```

Malheureusement, cette fonction ne fonctionne que sur des objets de type `int_sequence`. Il faudrait donc écrire une autre fonction pour les objets de type `float_sequence`

```
float_sequence float_sequence_nth_element(unsigned int n, float_sequence s) {
    if (n == 0) return s.head();
    else        return float_sequence_nth_element(n - 1, s.tail());
}
```

Évidemment, l'implémentation de la fonction est exactement la même ; la seule différence est le type des objets qui sont manipulés. Ceci est fâcheux puisqu'il s'agit d'une duplication d'efforts qui doit être effectuée à chaque fois qu'une nouvelle représentation pour une séquence d'un type différent est ajoutée. Les *templates* servent à éliminer ce problème. Au lieu de spécifier exactement la représentation du domaine d'une fonction, on peut utiliser un type fictif de la manière suivante :

```
template <typename Sequence>
Sequence nth_element(unsigned int n, Sequence s) {
```

```
    if (n == 0) return s.head();
    else        return nth_element(n - 1, s.tail());
}
```

Ensuite, le compilateur s'occupera lui-même de substituer les bons types puis de générer la fonction appropriée. Par exemple, lorsque le compilateur voit le code suivant

```
int_sequence s = ...;
int i = nth_element(10, s);
```

il commence par déterminer quel est le type qui devrait être substitué à la place de `Sequence` dans le template écrit plus haut. Il détermine que le type à substituer est `int_sequence`, puis il génère une fonction équivalente à celle que nous avions écrite pour le type `int_sequence`. Finalement, l'appel à `nth_element` sera en réalité un appel à la fonction générée pour le type `int_sequence`. L'idée est donc d'utiliser le compilateur comme un système de réécriture pour nous éviter des tâches répétitives dans certains cas. Cependant, il reste un cas très évident de répétition dans l'exemple que nous venons de donner ; les séquences sont une famille de types qui possèdent tous une interface similaire et dont l'implémentation pourrait être similaire, mais nous avons quand même répété exactement le même code pour chacun de ces types. Heureusement, les templates peuvent aussi être utilisés pour créer des canevas de types :

```
template <typename T>
struct sequence {
    bool is_empty() const;
    T head() const;
    sequence<T> tail() const;

    // implémentation ignorée pour simplifier
};
```

Ensuite, on aura qu'à écrire `sequence<int>` pour faire référence à la représentation d'une séquence contenant des `int`, `sequence<float>` pour une séquence de `float`, et ainsi de suite. Par exemple, lorsque le compilateur voit

```
sequence<float> s = ...;
```

il génère un type équivalent à `float_sequence` en substituant `T` par `float` dans le canevas écrit plus haut, et c'est à ce type généré que l'écriture `sequence<float>` fait référence.

Dans la réalité, il arrive souvent que l'on souhaite écrire une fonction template qui ait une implémentation bien spécifique pour certains types d'arguments, ou encore un type template qui ait une représentation spéciale dans certains cas. Ceci est particulièrement utile à des fins d'optimisation. Par exemple, supposons que nous voulions représenter une séquence de booléens. Nous pouvons utiliser le type `sequence<bool>`. Cependant, en supposant que

`sequence` stocke ses valeurs dans un tableau d'éléments contigus en mémoire, on peut remarquer qu'une représentation différente nous ferait sauver de l'espace. En effet, le type `bool` prend généralement 1 octet en mémoire parce qu'il s'agit de la plus petite unité adressable, mais nous n'avons en réalité besoin que d'un seul bit pour le représenter. Si on voulait stocker une séquence de booléens, on pourrait utiliser un tableau d'octets et n'utiliser qu'un seul bit par booléen. Le C++ rend ce genre d'optimisation très facile à implémenter en utilisant une technique appelée *spécialisation* :

```
template <>
struct sequence<bool> {
    bool is_empty() const;
    bool head() const;
    sequence<bool> tail() const;

    // on pourrait spécifier ici une représentation plus efficace
};
```

Avec une telle spécialisation, le type `sequence<T>` fait référence à l'implémentation pour le cas général présenté au début de cette section, mais le type `sequence<bool>` fait référence à la représentation plus efficace qu'on vient de donner. De plus, tant qu'on ne change pas l'interface publique du type `sequence<bool>` en le spécialisant, toutes les fonctions qui n'utilisent que cette interface (comme `nth_element`) fonctionneront encore. Cependant, il arrive fréquemment qu'il est possible d'implémenter une fonction de manière plus efficace si l'on tire avantage de la représentation d'un type. Le C++ permet donc aussi de spécifier une implémentation différente pour une fonction à l'aide d'une technique appelée *overloading*. Par exemple, on pourrait spécifier l'implémentation de la fonction `nth_element` pour le type `sequence<bool>` comme suit :

```
sequence<bool> nth_element(unsigned int n, sequence<bool> s) {
    // implémentation plus efficace
}
```

Lorsque le compilateur voit un appel à la fonction `nth_element`, il effectue un procédé plutôt complexe appelé *overload resolution*, qui consiste essentiellement à trouver la version de `nth_element` qui est la plus spécifique étant donné le type des arguments avec lesquels la fonction est appelée.

Un autre élément de base dont nous aurons besoin pour la suite est la déduction automatique du type d'une expression, qui a été introduite dans le langage en 2011. En C++, toute expression possède un type unique qui est connu par le compilateur. En principe, le compilateur devrait donc être capable de dire ce type au programmeur. C'est en effet le cas ; il suffit d'utiliser le mot clé `decltype`. Par exemple, pour savoir le type de l'expression `1 + 1`, on peut écrire `decltype(1 + 1)` :

```
// équivalent à `int i = 0`, parce que `decltype(1 + 1) = int`
decltype(1 + 1) i = 0;
```

De plus, il est possible de demander au compilateur de donner à une variable le type inféré d'une expression qu'on lui assigne. Pour ce faire, on utilise le mot clé `auto` :

```
auto i = 1 + 1;
```

Ici, le compilateur détermine que le type de `1 + 1` est `int`, et le type de `i` est donc `int`, exactement comme si on avait écrit `int i = 1 + 1`. Ce principe s'étend aussi au type de retour des fonctions. En effet, le compilateur peut inférer le type de retour d'une fonction à partir du type de l'expression qui est retournée. Par exemple, pour prendre un exemple un peu trivial,

```
auto add(int i, int j) {
    return i + j;
}
```

Ici, le compilateur sait que `i` et `j` sont de type `int`, puisque c'est comme cela qu'ils ont été déclarés comme paramètres de la fonction `add`. De plus, il sait que l'addition de deux `int` donne un `int` ; il peut donc inférer le type de retour de la fonction `add` comme étant le type `int`.

Bien que ces mots clés n'en aient pas l'air, ils sont terriblement utiles pour faire de la programmation générique. Par exemple, si on changeait la définition de `add` pour

```
template <typename T, typename U>
auto add(T t, U u) {
    return t + u;
}
```

le type de retour serait bien moins évident. Par exemple, si on appelle la fonction `add` avec une `std::string` et un `char const*`, il n'est pas du tout évident que le type de retour doit être `std::string`.[1] En général, il sera très difficile ou même impossible pour un programmeur d'écrire explicitement le type d'une expression lorsque celui-ci dépend de plusieurs autres expressions dont les types sont soit complexes ou inconnus du programmeur.

Le dernier élément de base qui sera nécessaire pour la suite est la notion d'*expression constante*. Une expression constante est, intuitivement, une expression dont la valeur est connue au moment de la compilation. Pour dénoter une expression constante, on utilise le mot clé `constexpr` :

```
constexpr int i = 0;
```

1. En effet, le fichier `<string>` fournit une fonction `operator+` ayant une signature de `std::string(std::string, char const*)`. C'est cet opérateur qui sera utilisé ici.

Les règles exactes de ce qui peut apparaître dans une expression constante sont un peu techniques, mais l'essentiel est qu'une expression constante ne doit contenir que des sous-expressions qui sont elles-aussi des expressions constantes, et que les litéraux comme `1`, `'x'` et `true` sont des expressions constantes. De plus, on peut utiliser certaines expressions constantes comme argument à des templates. C'est notamment le cas pour celles d'un type intégral (au sens du standard) comme `int` ou `bool`. Les expressions constantes sont un outil très important, comme nous le verrons dans ce qui suit.

2.2 La base de la métaprogrammation

Maintenant que ces notions de base sont écartées, nous pouvons nous pencher sur la vraie question : qu'est-ce que la métaprogrammation en C++ ? Lorsque nous avons présenté l'*overloading* et la spécialisation dans la section précédente, nous aurions pu remarquer qu'il s'agit en fait d'une manière d'introduire un branchement dans le compilateur. En effet, dépendamment d'une condition exprimée sous la forme d'un type, on peut demander au compilateur de compiler une fonction ou bien une autre en faisant en sorte que la branche que l'on souhaite soit choisie par l'*overload resolution*. Par exemple, considérons le type template `_bool`, qui représentera une condition connue au moment de la compilation. On peut alors obtenir un branchement équivalent à `condition ? x : y`, mais effectué au moment de la compilation, de la manière suivante :

```
template <bool condition>
struct _bool { };

template <typename Then, typename Else>
Then if_(_bool<true>, Then t, Else e) { return t; }

template <typename Then, typename Else>
Else if_(_bool<false>, Then t, Else e) { return e; }
```

L'expression `if_(condition, x, y)` est équivalente à `condition ? x : y`, à la différence que les types de `x` et `y` ne sont pas obligés d'être compatibles, et que la condition doit être un objet de type `_bool<...>` au lieu d'être un `bool`. En effet, lorsqu'on écrit

```
if_(_bool<true>{}, 1, std::string{"xyz"})
```

le compilateur détermine que la version de `if_` à choisir est la première et c'est donc le premier argument qui est retourné. Si on avait écrit

```
if_(_bool<false>{}, 1, std::string{"xyz"})
```

alors la deuxième fonction aurait été appelée et la `std::string` aurait été retournée. Tout l'intéret de cette technique est qu'elle permet aux objets dans les deux branches d'avoir des types incompatibles, ce qui n'est pas le cas avec l'opérateur `? :` usuel. En effet, avec cet

opérateur, les deux alternatives doivent avoir un type en commun, puisque le compilateur doit être en mesure de donner un type à cette expression au moment de la compilation. Or, il est possible que la condition ne soit connue qu'à l'exécution du programme. Le type de l'expression ne peut donc pas dépendre de la *valeur* de la condition. Avec notre technique, il est nécessaire que la condition soit une expression constante (puisqu'elle est utilisée comme argument dans le template `_bool`), mais on gagne alors la possibilité d'avoir des objets de types complètement différents dans chaque branche.

Le branchement à la compilation n'est pas la seule technique que nous pouvons exploiter à des fins de métaprogrammation. En effet, il est aussi possible d'écrire des fonctions complexes utilisant la récursion à l'intérieur même du compilateur. Pour illustrer ce fait, nous étudierons la fonction factorielle. Mais avant de regarder la factorielle elle-même, commençons par se créer un type qui représentera un entier connu au moment de la compilation ; ce sera ces objets que la factorielle manipulera :

```
template <int n>
struct _int { };
```

Exactement comme `_bool` un peu plus haut, `_int` ne sert qu'à représenter une valeur connue au moment de la compilation à l'aide d'un type. Étant donné un objet de type `_int<i>`, la seule information qui nous intéresse est le type de cet objet, puisqu'il contient l'entier que nous souhaitons représenter. C'est d'ailleurs la seule information qui nous est donnée par un tel objet, puisque le type `_int<i>` est un type singleton, c'est-à-dire qu'il ne possède qu'une seule valeur possible. Les raisons qui nous poussent à représenter les entiers connus au moment de la compilation par des objets de types singletons deviendront claires plus tard. Nous pouvons maintenant regarder l'implémentation de la fonction factorielle :

```
template <int n, int acc>
auto fact_impl(_int<n>, _int<acc>)
{ return fact_impl(_int<n - 1>{}, _int<acc * n>{}); }

template <int acc>
_int<acc> fact_impl(_int<0>, _int<acc>)
{ return {}; }

template <typename N>
auto fact(N n)
{ return fact_impl(n, _int<1>{}); }
```

Voici ce qui arrive lorsque l'on appelle `fact(_int<3>{})`. D'abord, `fact` appelle directement `fact_impl` en lui fournissant un accumulateur à 1 sous la forme d'un `_int`. Il s'agit là d'une simple astuce pour ne pas être obligé de fournir l'accumulateur soi-même lorsqu'on utilise `fact`. Ensuite, le compilateur détermine que la version de `fact_impl` à appeler est celle dont la signature est `?? fact_impl(_int<3>, _int<1>)`. Cependant, pour connaître

le type de retour de la fonction, on force le compilateur à aller voir le corps de la fonction, où on se trouve à appeler `fact_impl(_int<n - 1>{}, _int<acc * n>{})`. C'est là que la récursion arrive ; le compilateur est forcé de recommencer le même processus avec la fonction dont la signature est `fact_impl(_int<2>, _int<1 * 3>)`, ce qui le mènera éventuellement à considérer la fonction `fact_impl(_int<0>, _int<1 * 3 * 2 * 1>)`. À cette étape, la deuxième implémentation de `fact_impl` sera choisie, puisqu'elle est plus spécifique. Il s'agit du cas de base de la récursion. Le type de retour de cette fonction étant `_int<acc>` (`_int<1 * 3 * 2 * 1>` dans notre cas), le compilateur n'a pas besoin d'aller plus loin et il "remonte" la chaîne de fonctions qu'il était en train de compiler. Ainsi, le type de l'expression `fact(_int<3>{})` est évalué à `_int<1 * 3 * 2 * 1>`, et nous venons effectivement de forcer le compilateur à calculer la fonction factorielle. Cet exemple n'est pas très utile ; en effet, il existe de meilleures manières de calculer une factorielle au moment de la compilation en utilisant `constexpr`, et de toute façon la factorielle n'est pas si intéressante que ça. Cependant, il est intéressant de remarquer que la compilation des templates en C++ constitue un langage Turing-complet, et il est donc possible d'exprimer des algorithmes très complexes de cette manière, même si cela n'est généralement pas facile à faire en pratique.

2.3 L'arithmétique à la compilation

Le but de la bibliothèque Hana est précisément de rendre cette tâche plus facile et plus plaisante en fournissant des outils de base pour exprimer des algorithmes complexes au moment de la compilation. D'abord, la bibliothèque fournit des outils pour faire de l'arithmétique au moment de la compilation. Bien que nous n'en parlerons pas, il est à noter que ces outils sont basés sur une hiérarchie de concepts venant de l'algèbre abstraite ; on y trouve les concepts de monoïde, de groupe, d'anneau et de domaine intègre. Concrètement, ceci se matérialise via le type

```
template <typename T, T v>
struct _integral_constant { /* ... */ };
```

`_integral_constant` est un peu comme `_int` introduit plus haut, mais il est possible de spécifier une valeur de n'importe quel type intégral. De plus, on fournit des *variable templates* qui servent d'alias pour créer des objets de ces types avec un poid syntaxique minimal :

```
template <typename T, T v>
constexpr _integral_constant<T, v> integral_constant{};

template <int i>
constexpr auto int_ = integral_constant<int, i>;
```

```
// On peut ensuite écrire
auto three = int_<3>;
```

De plus, `_integral_constant` fournit les opérateurs arithmétiques usuels, ce qui permet de faire de l'arithmétique avec une syntaxe raisonable :

```
template <typename T, T t, typename U, U u>
auto operator+(_integral_constant<T, t>, _integral_constant<U, u>)
{ return _integral_constant<decltype(t + u), t + u>{}; }

template <typename T, T t, typename U, U u>
auto operator*(_integral_constant<T, t>, _integral_constant<U, u>)
{ return _integral_constant<decltype(t * u), t * u>{}; }

// etc...

auto seven = int_<3> + (int_<2> * int_<2>);
// `seven` est maintenant un objet de type `_integral_constant<int, 7>`
```

Il est bien important de se rendre compte de ce qui se passe ici. Ces opérateurs n'ont rien à voir avec les opérateurs arithmétiques usuels. En effet, ils n'utilisent pas du tout la *valeur* de leurs arguments, mais uniquement leur type. De plus, le type de retour dépend entièrement du type des arguments. La valeur de retour, quant à elle, est complètement inutile puisque `_integral_constant<...>` ne possède qu'une seule valeur, c'est à dire `_integral_constant<...>{}`. L'idée derrière ceci est en fait très puissante ; nous utilisons des objets de types singletons (n'ayant qu'une valeur possible) pour représenter des expressions constantes et faire des calculs au moment de la compilation.

Il est aussi intéressant de noter qu'il ne s'agit jusqu'ici que d'une différence syntaxique avec les approches précédentes comme la bibliothèque MPL, puisque la donnée d'une valeur d'un type singleton fournit exactement la même quantité d'information que la donnée de ce type seulement. Cependant, manipuler des valeurs est bien plus commode que manipuler des types directement. Bien entendu, cette correspondance nécessite la capacité de demander le type d'une expression au compilateur, ce qui est justement possible grâce à `decltype` et `auto`.

Avant de continuer, penchons-nous un peu plus sur la nature d'`_integral_constant`. Étant donné une expression constante `v` d'un type intégral `T`, on est capable de créer un objet du type singleton `_integral_constant<T, v>`, ce qui nous permet effectivement de représenter cette expression constante par un objet. Cependant, étant donné un tel objet, sommes-nous capable de retrouver l'expression constante qu'il représente ? Présentement, c'est impossible.[2] Or, il est crucial d'être capable de le faire. De plus, il est très important

2. En fait, c'est possible, mais comprendre pourquoi ça fonctionne dans tous les cas demande une excellente compréhension de `constexpr`. Les détails sont expliqués dans Hana.

que cette valeur soit considérée par le compilateur comme une expression constante, et ce peu importe que l'objet dont elle est issue soit une expression constante ou non. Pour que ce soit possible, il nous faut changer la définition d'`_integral_constant` de la manière suivante :

```
template <typename T, T v>
struct _integral_constant {
    static constexpr T value = v;
};
```

Maintenant, étant donné un objet de type `_integral_constant<...>`, on peut retrouver l'expression constante qu'il représente, et ce même si cet objet n'en est pas une. En effet, il suffit d'utiliser `decltype` :

```
// `n` n'est clairement pas une expression constante, puisque volatile.
volatile auto n = int_<3>;
constexpr bool v = decltype(n)::value;
```

On commence donc par déduire le type de `n`, qui est de la forme `_integral_constant<...>`. Ensuite, on va chercher l'expression constante qu'il représente et qui est stockée dans le membre de classe `value`. Bien que cela puisse sembler anodin, nous venons de faire quelque chose d'important. Nous étions en mesure de passer d'une expression constante vers un objet d'un type singleton. Maintenant, nous sommes aussi capables de parcourir le chemin en sens inverse et de passer d'un objet d'un type singleton vers une expression constante. Cette correspondance entre les objets de types singletons et ce qu'ils représentent est une partie importante de la bibliothèque Hana, et nous verrons d'autres correspondances similaires plus loin.

2.4 Le calcul sur les types

Être capable de faire de l'arithmétique à la compilation est intéressant, mais il y a plusieurs autres choses que l'on souhaiterait faire. Par exemple, il est souvent utile de manipuler des types comme s'ils étaient des objets, pour effectuer des transformations sur ceux-ci. Par exemple, on aimerait écrire une fonction `f` qui prend un type `T`, et qui renvoie le type "pointeur vers T", qu'on écrit `T*`. Attention, ici `T` n'est pas le domaine de la fonction `f`. En fait, mathématiquement, on pourrait écrire

$$f : \{\text{tous les types C++}\} \to \{\text{tous les types C++}\}$$
$$\texttt{T} \mapsto \texttt{T*}$$

Comment pourrait-on écrire une telle fonction en C++ ? Une option est d'utiliser le concept de *métafonction*, tel qu'introduit par la bibliothèque MPL. Une métafonction au sens MPL est un template contenant un alias nommé `type` qui contient la "valeur de retour" de la métafonction :

```
template <typename T>
struct f {
    using type = T*;
};
```

De plus, on se donne la convention qu'appliquer une métafonction `f` à un type `T` se fait de la manière suivante :

```
using U = f<T>::type;
// ‘U‘ est maintenant un alias pour ‘f<T>::type‘,
// qui est lui-même un alias pour ‘T*‘
```

Cette approche fonctionne assez bien, mis à part que la syntaxe est peu intuitive et qu'écrire des métafonctions complexes est assez difficile. Pour cette raison, Hana utilise une approche mixte qui consiste à écrire les transformations de base (précisément celles de `<type_traits>`) comme des métafonctions au sens MPL, puis de les manipuler en tant qu'objets pour construire des métafonctions plus complexes. Voici comment cela fonctionne. D'abord, on définit une famille de types singletons qui représenteront des types C++, exactement comme on a utilisé des types singletons pour représenter des entiers auparavant. Ceci est la clé pour être capable de manipuler des types comme s'ils étaient des objets :

```
template <typename T>
struct _type { /* ... */ };

// On se dote d'un alias
template <typename T>
constexpr _type<T> type{};

auto t = type<int>;
// ‘t‘ est maintenant un objet de type ‘_type<int>‘, qui représente
// le type ‘int‘ dans notre monde
```

Étant donné un type `T`, on est donc maintenant capable de fournir un objet qui représente ce type `T`. Cependant, étant donné un objet qui représente un type `T`, on est pas (pour l'instant) capable de retrouver le type `T`. Pour être capable de faire la correspondance entre le monde des types et celui des valeurs dans les deux sens, on fait la chose suivante. D'abord, on fournit un alias qui fait en sorte que `_type<T>` devienne une métafonction au sens MPL :

```
template <typename T>
struct _type {
    using type = T;
};
```

Ensuite, étant donné un objet qui représente un type `T`, on est capable de retrouver `T` comme suit :

```
auto t = ...; // un `type<T>`
using T = decltype(t)::type;
```

On utilise donc l'inférence de type fournie par le compilateur avec `decltype` pour d'abord obtenir `_type<T>` à partir de `t`, puis on utilise le fait que `_type<T>` est une métafonction MPL pour obtenir un alias au `T` initial. D'ailleurs, il est très éclairant de constater que ce procédé est analogue à ce que nous avons fait pour `_integral_constant`, où nous avions la variable de classe `value` qui jouait le rôle de `type` et qui nous permettait de traverser le pont entre les expressions constantes et les objets singletons dans le sens inverse.

Pour être en mesure de faire des calculs sur des types, il nous manque cependant un bloc important. Comment pouvons-nous encoder des fonctions qui prennent des types en argument et qui retournent des types ? Encore une fois, il s'agit de définir une famille de types singletons qui représenteront des métafonctions :

```
template <template <typename ...> class F>
struct _metafunction {
    template <typename ...T>
    auto operator()(_type<T>...) const {
        using Result = F<T...>::type;
        return type<Result>;
    }
};

// On se dote d'un alias
template <template <typename ...> class F>
constexpr _metafunction<F> metafunction{};

auto f = metafunction<std::add_pointer>;
// `f` est maintenant un objet de type `_metafunction<std::add_pointer>`,
// qui représente la métafonction `std::add_pointer` dans notre monde

auto p = f(type<int>);
// `p` est maintenant un objet de type `_type<int*>`, qui représente le
// type `int*` dans notre monde!
```

Voyons comment cela fonctionne. D'abord, nous dotons `_metafunction<F>` d'un opérateur d'appel. Cet opérateur assume que `_metafunction<F>` ne sera appelé qu'avec des objets de type `_type<T>`. Ensuite, dans le corps de la fonction, on commence par appliquer la métafonction `F` aux types `T` cachés dans les paramètres `_type<T>...`, puis on stocke le résultat dans l'alias `Result` pour faciliter la lecture. On retourne finalement un objet

de type `_type<Result>` en utilisant le raccourci `type<Result>`. C'est donc de la manière suivante qu'il faut comprendre l'expression `f(type<int>)` :

```
f(type<int>) == metafunction<std::add_pointer>(type<int>)
             == type<std::add_pointer<int>::type>
             == type<int*>
```

De cette manière, on se trouve à manipuler des fonctions plutôt que des métafonctions, et il devient possible d'effectuer des calculs sur des types avec la même syntaxe qu'on utilise pour la programmation usuelle.

2.5 Les types généralisés

En programmation conventionnelle, les types sont utilisés pour classifier des valeurs qui ont la même représentation afin, par exemple, d'écrire des fonctions qui sont capables de manipuler n'importe quelle valeur d'un type donné. L'importance de classifier les valeurs selon leur type vient entre autres du fait que cela nous permet de raisonner de manière générale à propos de toutes les valeurs de ce type en même temps.

Mais jusqu'ici, nous avons utilisé les types pour un usage tout à fait différent. En effet, Hana utilise sans réserve les types pour encoder des valeurs connues au moment de la compilation. Les types deviennent donc un véhicule par lequel il est possible de faire faire des calculs au compilateur, ce qui nous arrange bien. Cependant, on y a perdu une chose fondamentale ; il s'agit de la capacité de raisonner de manière formelle à propos de notre code. En particulier, il nous est pour l'instant difficile de définir le "domaine" d'une fonction manipulant des objets de types hétérogènes, puisque nous utilisons les types – qui seraient normalement utilisés pour représenter le domaine de la fonction – comme un véhicule pour les valeurs que nous manipulons. Par exemple, considérons la fonction suivante, que nous avons déjà étudiée lorsque nous avons introduit les calculs sur des types :

```
template <typename T>
auto add_pointer(_type<T>) {
    return type<T*>;
}
```

Quel est le domaine de cette fonction ? Mécaniquement, on peut dire que le domaine est

$$\bigcup_{\text{tous les types } \texttt{T}} \texttt{\_type<T>}$$

Cependant, sous cette forme, ce domaine ne nous est pas vraiment utile. Pour pouvoir mettre un nom sur cet ensemble, Hana introduit le concept de type généralisé. En essence, un type généralisé est une étiquette qui rassemble une famille de types (comme `_type<...>`) sous une même bannière. Ceci nous permet de manipuler des objets de n'importe quel type faisant partie de cette famille comme étant un représentant de la bannière.

Par exemple, pour représenter la famille de types `_type<...>`, on utilisera la bannière `Type`. D'une certaine manière, on peut donc dire que

$$\texttt{Type} = \bigcup_{\text{tous les types } \texttt{T}} \texttt{\_type<T>}$$

Concrètement, les types généralisés sont implémentés comme des types conventionnels qui ne servent que d'étiquette à une famille de types conventionnels. La métafonction `datatype` de Hana permet d'obtenir le type généralisé d'un type conventionnel. Étant donné un type `T` quelconque, le type généralisé de `T` est `datatype<T>::type` :

```
using GeneralizedT = datatype<T>::type;
```

D'ailleurs, pour ceux qui sont familiers avec des bibliothèques comme Boost.Fusion et Boost.MPL, la notion de type généralisé coïncide avec la notion de *tag* dans ces bibliothèques. Cependant, Hana pousse le principe beaucoup plus loin. Le changement décisif qui est introduit dans Hana est l'utilisation de ces étiquettes comme étant le "type" des objets. Ainsi, dans Hana, les objets sont classifiés par leur type généralisé et les fonctions spécifient leur domaine comme un type généralisé plutôt qu'un type conventionnel. Par exemple, la définition rigoureuse de notre fonction `add_pointer` devient

$$\texttt{add\_pointer} : \texttt{Type} \to \texttt{Type}$$

ce qui veut dire que `add_pointer` accepte n'importe quel objet dont le type généralisé est `Type`, et ce peut importe son type réel (qui sera de la forme `_type<T>`). Ceci permet de faire abstraction de la représentation des objets, qui n'est généralement qu'un détail d'implémentation dans le contexte de la métaprogrammation. De plus, on peut se rendre compte que la notion de type généralisé est strictement plus générale que celle de type conventionnel, d'où le nom qui a été choisi. En effet, il suffirait de poser le type généralisé de tout type conventionnel comme étant ce type lui-même pour retomber sur la notion habituelle de type. C'est d'ailleurs ce qui est fait dans Hana. Plus précisément, le type généralisé d'un type est ce type lui-même par défaut, mais il est possible de spécifier le type généralisé d'une famille de types si on le désire. Pour se faire, il suffit d'utiliser la spécialisation des *templates* pour la métafonction `datatype`. Par exemple, le type généralisé de `int` est `int`, mais celui de n'importe quel `_type<T>` est `Type`, puisque `datatype` est spécialisé de la manière suivante :

```
struct Type; // représente le type généralisé Type
template <typename T>
struct datatype<_type<T>> {
    using type = Type;
};
```

Mais ce n'est pas tout. Le C++ possède la notion de *template*, qui est dans l'essence un type paramétré, ou encore une fonction sur les types. Cette notion de type paramétré est fondamentale si l'on veut exprimer des types moindrement complexes, comme par exemple `std::vector<T>`. En bonne généralisation de la notion de type, Hana introduit la notion de type généralisé paramétré. Formellement, on peut voir ceci comme une fonction sur les types généralisés, exactement comme les *templates* sont des fonctions sur les types.

Sans s'en rendre compte, nous avons déjà vu une manifestation des types généralisés paramétrés lorsque que nous voulions manipuler des entiers au moment de la compilation. Au départ, nous avions commencé avec `_int<i>`, puis nous avions étendu le principe à `_integral_constant<T, i>` pour être capable de manipuler des entiers de n'importe quel type intégral `T`. En réalité, étant donné un type intégral `T`, le type généralisé de `_integral_constant<T, i>` est `IntegralConstant(T)`, et ce peut-importe l'entier `i` que `_integral_constant<T, i>` représente. Vu de cette manière, `IntegralConstant` est donc un type généralisé paramétré par le type généralisé `T` de la valeur entière qu'il représente. Cependant, étant donné que `IntegralConstant` ne peut que représenter des valeurs de types intégraux (`int`, `long`, etc...), les types généralisés auxquels il fait du sens d'appliquer `IntegralConstant` coïncident tous avec des types standards du C++.

Au passage, on notera que nous utilisons la notation `IntegralConstant(T)` plutôt que `IntegralConstant<T>` pour dénoter l'application de `IntegralConstant` à `T`. Nous faisons ceci principalement pour deux raisons. D'abord, cela permet de mettre en évidence la différence entre les types généralisés et les *templates*. De plus, la plupart des types généralisés paramétrés de Hana ne sont pas implémentés comme des *templates* pour des raisons techniques ; il serait donc mal choisi d'utiliser une notation associée aux *templates*.

Bien que la notion de type généralisé puisse sembler un peu vide de sens à ce stade, les prochaines sections mettrons en évidence l'importance des types généralisés comme brique de base de la bibliothèque et de sa formalisation présentée ici.

3 Les catégories : premier contact

L'intérêt de se tourner vers la théorie des catégories est de guider le design de la bibliothèque pour obtenir des interfaces très générales et spécifiées clairement, formellement. La généralité de la théorie permet d'avoir des interfaces qui sont très abstraites et qui en couvrent donc très large, ce qui réduit la complexité globale de la bibliothèque. La capacité de formaliser ces interfaces permet d'automatiser une bonne partie des tests et de raisonner à propos des programmes écrits avec Hana de manière très précise, quasiment mathématique.

Définition (Catégorie)**.** Une catégorie $\mathcal{C}$ est composée de trois choses :

1. Une collection d'objets, habituellement notée $\mathrm{ob}(\mathcal{C})$.
2. Une collection de flèches (aussi appelées morphismes) entre ces objets, habituellement notée $\hom(\mathcal{C})$.
3. Une opération binaire, habituellement notée $\circ$, permettant de composer n'importe quelle paire de flèches dont les sources et destinations sont compatibles pour en obtenir une troisième.

Pour que $\mathcal{C}$ soit une catégorie, ces "choses" doivent aussi respecter certaines propriétés, qui seront expliquées un peu plus bas. Avant de continuer, il est important de préciser quelques détails et d'introduire un peu de notation qui rendra notre tâche plus facile pour la suite. D'abord, pour deux objets $X, Y \in \mathrm{ob}(\mathcal{C})$, on note la collection des flèches allant de X vers Y par $\hom(X, Y)$, ou encore $\hom_{\mathcal{C}}(X, Y)$ lorsque la catégorie $\mathcal{C}$ n'est pas évidente à partir du contexte. On utilise aussi la notation $f : X \to Y$ pour dire que f est une flèche allant de X vers Y, c'est-à-dire que $f \in \hom(X, Y)$. On dit alors que f est de source X et de destination Y, noté $\mathrm{dom}(f)$ et $\mathrm{range}(f)$, respectivement.

Ensuite, il est important de bien comprendre qu'on ne parle pas d'une seule opération binaire, mais en réalité d'une famille d'opérations binaires $\circ$ qui associe à chaque paire de hom-classes $\hom(Y, Z)$ et $\hom(X, Y)$ une opération

$$\begin{aligned} \circ_{X,Y,Z} : \hom(Y, Z) \times \hom(X, Y) &\to \hom(X, Z) \\ g, f &\mapsto g \circ_{X,Y,Z} f \end{aligned}$$

Pour éviter de s'enfarger dans les fleurs du tapis, nous utiliserons simplement la notation $\circ$, comme il est d'usage dans la littérature. Nous sommes maintenant prêts à passer aux propriétés qui doivent être satisfaites pour qu'on ait bien une catégorie. D'abord, pour tout objet $X \in \mathrm{ob}(\mathcal{C})$, il doit exister une flèche de X vers lui-même, qu'on appelle généralement identité (sur X) :

$$\mathrm{id}_X : X \to X$$

Ensuite, la composition $\circ$ définie plus haut doit être associative, c'est-à-dire que pour toutes flèches $f : A \to B$, $g : B \to C$ et $h : C \to D$, on doit avoir

$$(h \circ g) \circ f = h \circ (g \circ f)$$

Finalement, on doit avoir que les identités sont des éléments neutres de la composition, c'est-à-dire que pour toute flèche $f : A \to B$,

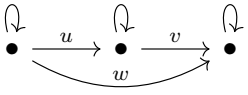
$$f = \mathrm{id}_B \circ f = f \circ \mathrm{id}_A$$

Il est intéressant d'observer que ces deux dernières propriétés sont semblables à demander que $(\hom(\mathcal{C}), \circ)$ soit un monoïde. Ce n'est évidemment pas le cas puisqu'on a plusieurs identités et parce que $\circ$ n'est pas définie sur la totalité de $\hom(\mathcal{C}) \times \hom(\mathcal{C})$, mais l'intuition reste utile.

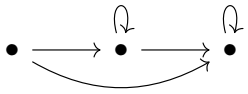
3.1 Exemples de catégories

Avec la définition d'une catégorie qui vient d'être faite, il est un peu difficile de voir comment les catégories peuvent se matérialiser. Nous explorerons donc maintenant des catégories simples qui permettent de se forger une intuition dans le but éventuel d'étudier des catégories plus utiles. Dans cet optique, on se doit de commencer par la catégorie la plus simple ; la catégorie vide. Comme son nom l'indique, cette catégorie n'a pas d'objets ni de flèches, et sa loi de composition est une application vide $\circ : \emptyset \times \emptyset \to \emptyset$. Les propriétés d'une catégorie sont trivialement satisfaites. Une autre catégorie simple (et peu utile) est la catégorie triviale, qui ne contient qu'un seul objet et sa flèche identité :

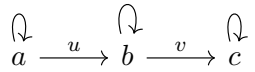
De manière générale, on peut voir une catégorie comme un graphe dirigé avec certaines propriétés. Par exemple, le graphe suivant est une catégorie :



Les objets sont les sommets du graphe, les flèches sont les arrêtes et la loi de composition est une application qui associe à deux arrêtes u et v une troisième arrête w qui part de la source de u jusqu'à la destination de v. Bien entendu, tous les graphes ne sont pas des catégories. Par exemple, le graphe suivant ne peut pas être une catégorie :



En effet, il n'y a pas d'arrête qui part du sommet de gauche vers lui-même, ce qui ne respecte pas le fait que chaque objet doit avoir une flèche identité. Aussi, bien qu'il respecte les identités, le graphe suivant ne peut pas être une catégorie :



En effet, il existe une arrête entre les sommets a et b et une arrête entre les sommets b et c, mais il n'y en a pas entre les sommets a et c. Si on essayait de définir la loi de composition, on se rendrait compte qu'il n'y a pas de valeur possible pour $v \circ u$. Ainsi, la composition serait mal définie et le graphe ne peut pas représenter une catégorie. En général, on peut créer une catégorie à partir de n'importe quel graphe dirigé en ajoutant simplement les flèches manquantes.

Une des raisons d'étudier les catégories est leur capacité d'unifier des notions sous un même emblème. Suivant cette ligne de pensée, il est possible d'inscrire la plupart des structures provenant de l'algèbre abstraite dans un contexte catégorique. On commence d'abord par la catégorie Set, dont les objets sont les ensembles, les flèches sont les applications ensemblistes et la composition est la composition usuelle de fonctions. On peut ensuite ajouter de la structure à ces ensembles et demander que les applications ensemblistes respectent cette structure. Ceci nous mène à considérer, par exemple, la catégorie Grp, où les objets sont les groupes et les flèches sont des homomorphismes de groupes, avec la composition usuelle. On peut continuer sur ce chemin et considérer la catégorie Ring des anneaux, où les flèches sont des homomorphismes d'anneaux, puis la catégorie $\mathrm{K}-\mathrm{Vect}$ des espaces vectoriels sur un corps K, où les flèches sont les transformations K-linéaires, et ainsi de suite. Du côté de l'analyse, on peut considérer par exemple la catégorie Met des espaces métriques, où les flèches sont les applications 1-Lipschitz. Les exemples sont nombreux ; en fait, la plupart des théories mathématiques peuvent être exprimés dans ce cadre, parfois moyennant une petite dose d'imagination.

3.2 La catégorie Hana

Il est possible de représenter l'univers dans lequel on évolue en métaprogrammation à l'aide d'une catégorie. Ceci nous permettra d'importer des concepts de la théorie des catégories vers la métaprogrammation. Précisément,la catégorie Hana est la catégorie dont les objets sont l'ensemble des types généralisés :

$$\mathrm{ob}(\mathit{Hana}) := \{\texttt{T} : \texttt{T} \text{ est un type généralisé}\}$$

Ensuite, les flèches de Hana sont les fonctions génériques qui travaillent sur des objets ayant un certain type généralisé, et ce peu importe leur type réel. Plus précisément, soit $\texttt{X}, \texttt{Y} \in \mathrm{ob}(\mathit{Hana})$, c'est-à-dire que `X` et `X` sont deux types généralisés. Une flèche $\texttt{f} : \texttt{X} \to \texttt{Y}$ est un objet C++ avec un `operator()` tel que $\forall \texttt{x} \in \texttt{X}$,

1. l'expression `f(x)` est bien formée, c'est-à-dire qu'elle compile
2. $\texttt{f(x)} \in \texttt{Y}$, c'est-à-dire que le type généralisé de l'expression `f(x)` est `Y`
3. l'expression `f(x)` n'a pas d'effets de bord

La propriété (3) est nécessaire pour s'assurer que les fonctions qu'on considère sont des fonctions au sens mathématique, c'est-à-dire que

$$x = y \implies f(x) = f(y)$$

Sans se restreindre explicitement à cette classe de fonctions, on perdrait la capacité de raisonner mathématiquement à propos des fonctions de Hana puisque le langage C++ n'est pas pur, contrairement au Haskell. Ce n'est cependant pas une restriction qui nous handicap beaucoup, puisque Hana s'intéresse surtout à la manipulation d'objets hétérogènes, ce qui requiert l'utilisation d'un style de programmation fonctionnel de toute façon. On notera aussi qu'on ignore les fonctions qui ne terminent pas, un abus qui est généralement accepté puisque justifiable [2]. Finalement, on ignore aussi les détails du C++ comme les restrictions sur le type `void` ou le fait qu'on ne peut pas passer un tableau comme argument à une fonction. Il n'y a pas de perte de généralité puisqu'on peut créer des types équivalents à `void` et `T[N]`, mais qui se comportent comme des vrais objets.

Finalement, la loi de composition sur Hana est, sans grande surprise, l'extension de la composition de fonctions usuelle aux fonctions *templates* qui constituent les flèches de Hana, $\hom(Hana)$. Rigoureusement, soit $\texttt{X}, \texttt{Y}, \texttt{Z} \in \mathrm{ob}(Hana)$ des types généralisés. On peut définir la composition par

```
template <typename X, typename Y, typename Z>
auto compose = [](auto f, auto g) {
    return [=](auto x) {
        static_assert(std::is_same<datatype_t<decltype(x)>, X>{}, "");
        static_assert(std::is_same<datatype_t<decltype(g(x))>, Y>{}, "");
        static_assert(std::is_same<datatype_t<decltype(f(g(x)))>, Z>{}, "");
        return f(g(x));
    };
};
```

Ainsi, `compose<X, Y, Z>` peut être appelée avec n'importe quelles fonctions `f` et `g`, mais celles dont les domaines et codomaines ne coïncident pas avec `X`, `Y` et `Z` résulteront en une erreur de compilation. Dans la réalité, il est trop répétitif de préciser les types `X`, `Y` et `Z` et on préfère la définition suivante pour `compose` :

```
auto compose = [](auto f, auto g) {
    return [=](auto x) {
        return f(g(x));
    };
};
```

La tâche de s'assurer que les domaines et codomaines sont respectés est ainsi laissée au programmeur, mais l'interface qui en résulte est beaucoup plus facile à utiliser. Il nous

reste à démontrer que Hana est bien une catégorie. Pour ce faire, il suffit de vérifier qu'il existe une flèche identité pour chaque objet de la catégorie et que la composition est bien associative. D'abord, la flèche identité d'un type généralisé peut être définie en utilisant une fonction *template* ou une fonction anonyme générique de la manière suivante :

```
template <typename X>
auto id = [](auto x) {
    static_assert(std::is_same<datatype_t<decltype(x), X>>{}, "");
    return x;
};
```

Ainsi, étant donné un type généralisé `X`, l'identité sur `X` est `id<X>`. Puisque le type C++ de l'argument de `id<X>` est quelconque, on peut passer n'importe quel objet à cette fonction, tant que le type généralisé de cet objet est exactement `X`. Si ce n'est pas le cas, le compilateur rapportera une erreur de compilation. Pour alléger la notation, on écrira `id` au lieu de `id<X>`, un peu comme on a fait pour `compose` plus haut. De plus, il est clair que l'identité est un élément neutre de la loi de composition. En effet, soit `A` et `B` des types généralisés et $\texttt{f} : \texttt{A} \to \texttt{B}$ une fonction. Alors,

```
compose(id, f) == [](auto x) { return id(f(x)); }
               == [](auto x) { return f(x); }
               == f
```

De manière similaire,

```
compose(f, id) == [](auto x) { return f(id(x)); }
               == [](auto x) { return f(x); }
               == f
```

ce qui montre que l'identité est bien l'élément neutre de la composition. Ensuite, il reste à montrer que la composition est associative, ce qui n'est pas difficile non plus. Pour des types généralisés `A`, `B`, `C`, `D`, soit $\texttt{f} : \texttt{A} \to \texttt{B}$, $\texttt{g} : \texttt{B} \to \texttt{C}$ et $\texttt{h} : \texttt{C} \to \texttt{D}$ des fonctions. Alors,

```
compose(h, compose(g, f))
    == [](auto x) { return h(compose(g, f)(x)); }
    == [](auto x) { return h([](auto x) { return g(f(x)); }(x)); }
    == [](auto x) { return h(g(f(x))); }
```

De l'autre côté, on obtient que

```
compose(compose(h, g), f)
    == [](auto x) { return compose(h, g)(f(x)); }
    == [](auto x) { return [](auto x) { return h(g(x)); }(f(x)); }
    == [](auto x) { return h(g(f(x))); }
```

ce qui montre que la composition est associative, et Hana est donc bien une catégorie.

4 Les foncteurs

Nous avons introduit la notion de catégorie à l'aide d'exemples provenant de l'algèbre abstraite, de l'analyse puis de l'informatique. L'algorithme pour *catégoriser* une théorie était toujours un peu le même : isoler les objets d'étude de la théorie, isoler les propriétés importantes qui les définissent puis considérer l'ensemble des transformations qui, de manière informelle, respectent ces propriétés. Or, la théorie des catégorie est elle-même une théorie qui étudie certains objets appelés "catégories". Il serait donc naturel de se demander s'il est possible de définir des transformations entre les catégories et, si c'est le cas, quelles propriétés doivent posséder ces transformations pour qu'on obtienne quelque chose d'intéressant. Comme on s'apprête à le voir, ce questionnement et sa réponse sont à la source de la notion de *foncteur*, qui n'est en effet rien d'autre qu'une transformation entre catégories.

D'abord, une catégorie $\mathcal{C}$ possède non seulement des objets, mais aussi des flèches entre ces objets. Ainsi, peu importe la forme que prendra notre transformation inter-catégories, elle devra spécifier où elle envoie les objets, mais aussi les flèches. Considérons donc une application $F : \mathcal{C} \to \mathcal{D}$ d'une catégorie $\mathcal{C}$ vers une catégorie $\mathcal{D}$. On devra avoir que pour chaque objet $A \in \mathrm{ob}(\mathcal{C})$, F envoie A vers un objet de $\mathcal{D}$, disons $F(A) \in \mathrm{ob}(\mathcal{D})$. De manière similaire, pour chaque flèche $f : A \to B \in \hom(\mathcal{C})$, F devra envoyer f sur une flèche de $\mathcal{D}$, disons $F(f) \in \hom(\mathcal{D})$.

Il est intéressant de remarquer que l'on a pas spécifié la source ni la destination de la flèche $F(f)$. En fait, pour une flèche $f : A \to B$, seuls deux choix s'offrent à nous : soit $F(f) : F(A) \to F(B)$, soit $F(f) : F(B) \to F(A)$. Ces deux choix sont valables. Dans le premier cas, on dira que c'est un foncteur *covariant*, ou juste un foncteur, et dans l'autre cas on dira que c'est un foncteur *contravariant*. Dans le cadre actuel, nous nous intéresserons seulement aux foncteurs covariants.

Ensuite, une catégorie possède une loi de composition et chaque objet doit posséder une flèche identité ; il serait probablement intéressant qu'une transformation entre catégories préserve ces propriétés d'une certaine manière. On demandera donc que la loi de composition soit respectée, c'est-à-dire que pour des flèches $f : A \to B$ et $g : B \to C$ de $\mathcal{C}$,

$$F(g \circ_{\mathcal{C}} f) = F(g) \circ_{\mathcal{D}} F(f)$$

Il est possible de visualiser cette équation sous la forme d'un diagramme. En effet, dire que cette équation est satisfaite est équivalent à dire que le diagramme suivant commute :

$$\begin{array}{ccccc} A & \xrightarrow{f} & B & \xrightarrow{g} & C \\ \downarrow{\scriptstyle F} & & & & \downarrow{\scriptstyle F} \\ F(A) & \xrightarrow{F(f)} & F(B) & \xrightarrow{F(g)} & F(C) \end{array}$$

En d'autres mots, on souhaite que les chemins bleus et rouges donnent exactement la même réponse. Ensuite, au même titre qu'on souhaite que la loi de composition soit

respectée, on demandera que les flèches identité soient préservées par un foncteur, c'est-à-dire que pour un objet A de $\mathcal{C}$,

$$F(id_A) = id_{F(A)}$$

Ceci nous mène à la définition formelle suivante :

Définition (Foncteur)**.** Un foncteur $F : \mathcal{C} \to \mathcal{D}$ d'une catégorie $\mathcal{C}$ vers une catégorie $\mathcal{D}$ est une application qui

1. à chaque objet $X \in \mathrm{ob}(\mathcal{C})$ associe un objet $F(X) \in \mathrm{ob}(\mathcal{D})$
2. à chaque flèche $f : A \to B \in \hom(\mathcal{C})$ associe une flèche $F(f) : F(A) \to F(B) \in \hom(\mathcal{D})$
3. respecte la loi de composition, c'est-à-dire que pour toutes flèches $f : A \to B$ et $g : B \to C \in \hom(\mathcal{C})$, $F(g \circ_{\mathcal{C}} f) = F(g) \circ_{\mathcal{D}} F(f)$
4. préserve les identités, c'est-à-dire que pour tout objet $X \in \mathrm{ob}(\mathcal{C})$, $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$

Lorsque le domaine et le codomaine d'un foncteur sont la même catégorie, i.e. lorsqu'on a un foncteur du type $F : \mathcal{C} \to \mathcal{C}$, on dira parfois que F est un *endofoncteur*.

4.1 Les foncteurs dans Hana

Dans le cadre de la bibliothèque Hana, on s'intéressera principalement aux endofoncteurs sur la catégorie Hana, c'est-à-dire aux foncteurs du type $Hana \to Hana$. D'abord, rappelons-nous que les objets de la catégorie Hana sont des types généralisés. Un foncteur F devra donc envoyer un type généralisé `T` vers un autre type généralisé $F($`T`$)$. Ensuite, les flèches de Hana sont des fonctions dont le domaine et le codomaine sont des types généralisés. Ainsi, F devra envoyer une telle fonction `f` $:$ `T` $\to$ `U` vers une fonction $F($`f`$) : F($`T`$) \to F($`U`$)$. Dans Hana, un foncteur se matérialise comme un type généralisé paramétré muni d'une fonction appelée `transform` ayant la signature suivante :

$$\texttt{transform} : \texttt{F(T)} \times (\texttt{T} \to \texttt{U}) \to \texttt{F(U)}$$

Comme on pouvait s'en douter, étant donné un type généralisé `T`, `F` envoie `T` sur un nouveau type généralisé `F(T)`. Cependant, le rôle de `transform` demande un peu plus de justification. On notera que l'ordre des paramètres de `transform` est fâcheux pour l'explication qui suit. C'est tout de même cet ordre qui est utilisé dans Hana parce qu'il est avantageux lorsqu'on veut passer une fonction anonyme C++ à `transform`. Supposons que l'on ait une fonction `f` $:$ `T` $\to$ `U`. On peut appliquer partiellement `f` à `transform` pour obtenir la fonction

$$\texttt{transform(-, f)} : \texttt{F(T)} \to \texttt{F(U)}$$

On peut remarquer que `transform(-, f)` possède la signature requise pour être l'image d'une flèche de Hana par le foncteur `F`. Dans la catégorie Hana, un foncteur `F` envoie donc

un type généralisé `T` sur un nouveau type généralisé `F(T)`, et une fonction `f : T → U` sur une nouvelle fonction `transform(-, f) : F(T) → F(U)`. Bien entendu, pour qu'il s'agisse bien d'un foncteur, certaines propriétés doivent être respectées. D'abord, le foncteur doit respecter la loi de composition de Hana, i.e. pour toutes fonctions `f : A → B` et `g : B → C`, on doit avoir que

```
compose(transform(-, g), f) == compose(transform(-, g), transform(-, f))
```

ce qui est équivalent à dire que pour tout `xs` de type généralisé `F(T)`,

```
transform(xs, compose(g, f)) == transform(transform(xs, f), g)
```

Ensuite, le foncteur doit aussi respecter l'identité, c'est à dire que $F(\mathrm{id}_{\mathtt{T}}) = \mathrm{id}_{\mathtt{F(T)}}$. Dans Hana, puisque l'identité est une fonction générique qui fonctionne sur tous les types généralisés, ceci se traduit simplement en disant que pour tout `x` de type généralisé `F(T)`,

```
transform(x, id) == x
```

Voyons maintenant comment certaines structures fournies par Hana peuvent être vues comme des foncteurs.

4.2 Le foncteur Maybe

En programmation, on est souvent amené à utiliser des fonctions qui peuvent échouer. Par exemple, lorsqu'on divise une quantité par une autre, il peut arriver que le dénominateur soit nul, auquel cas la division ne peut avoir lieu. Un autre exemple serait une fonction qui accède au n-ième élément d'une séquence ; il peut arriver que la séquence contienne moins de n éléments, auquel cas la fonction ne peut pas retourner de résultat. Or, en programmation, il nous faut une manière de gérer ces éventualités. Un mécanisme classique pour y arriver est d'utiliser des exceptions. Cependant, un des désavantages des exceptions est qu'elles ne sont généralement pas encodées dans le type de la fonction qui peut les produire, et le fardeau de bien les gérer appartient donc au programmeur. Un autre problème majeur dans le contexte de la métaprogrammation est que les exceptions n'existent pas au moment de la compilation, et ce mécanisme nous est donc complètement inaccessible.

Une autre manière de gérer la possibilité d'un échec est d'introduire la notion de valeur optionnelle. Essentiellement, une valeur optionnelle est une valeur qui peut être là ou ne pas y être, accompagnée d'un mécanisme pour savoir si la valeur est là ou non. Une fonction qui peut échouer devient donc une fonction qui n'échoue jamais, mais qui peut retourner une valeur optionnelle vide. Pour une valeur de type `T`, on dénotera une valeur optionnelle de ce type par le type `Maybe(T)`. Une valeur optionnelle peut être créée de deux manières. D'abord, on peut créer une valeur optionnelle (qui existe) avec `just(x)`, où `x` est une valeur de type `T`. Ensuite, on peut créer une valeur optionnelle qui est vide avec `nothing`.

Par exemple, on pourrait écrire une fonction qui effectue une division entière de la manière suivante (en écrivant `Int` au lieu de `IntegralConstant(int)` pour alléger) :

$$\begin{aligned} \texttt{div} : \texttt{Int} \times \texttt{Int} &\to \texttt{Int} \\ x, y &\mapsto x/y \end{aligned}$$

Cependant, on pourrait aussi écrire une fonction qui effectue la division de manière sécuritaire en utilisant `Maybe` :

$$\begin{aligned} \texttt{safe\_div} : \texttt{Int} \times \texttt{Int} &\to \texttt{Maybe(Int)} \\ \texttt{x}, \texttt{y} &\mapsto \begin{cases} \texttt{nothing si y == int\_<0>} \\ \texttt{just(x / y)} \text{ sinon} \end{cases} \end{aligned}$$

Pour l'instant, ceci ne nous est pas très utile parce que nous n'avons pas de façon d'extraire la valeur optionnelle lorsqu'elle existe. C'est le rôle de la fonction `from_just`, qui est définie par

$$\begin{aligned} \texttt{from\_just} : \texttt{Maybe(T)} &\to \texttt{T} \\ \texttt{just(x)} &\mapsto \texttt{x} \\ \texttt{nothing} &\mapsto \texttt{erreur} \end{aligned}$$

Pour que ceci nous soit d'une quelconque utilité, il nous faut aussi une manière de savoir quand est-ce qu'une valeur optionnelle est vide ou pleine :

$$\begin{aligned} \texttt{is\_just} : \texttt{Maybe(T)} &\to \texttt{IntegralConstant(bool)} \\ \texttt{just(x)} &\mapsto \texttt{true\_} \\ \texttt{nothing} &\mapsto \texttt{false\_} \end{aligned}$$

On peut ensuite utiliser le résultat de `safe_div` comme suit :

```
auto maybe_result = safe_div(int_<10>, int_<5>);
if (is_just(maybe_result)) {
    auto result = from_just(maybe_result);
    f(result); // on peut utiliser le résultat de safe_div
}
else {
    // on sait que safe_div a planté
}
```

On constate rapidement que le gain est plutôt faible par rapport à une solution où l'on vérifierait explicitement que `y != 0` et où l'on n'utiliserait pas de `Maybe`. En effet, il est quand même nécessaire de vérifier si la valeur optionnelle est vide ou pleine avant de

pouvoir utiliser le résultat de la fonction (si résultat il y a). Ce qu'on aimerait est une manière de seulement spécifier le contenu de la branche (par exemple comme une fonction) et de soit appliquer la fonction au contenu du `Maybe`, soit ne pas l'appliquer et avoir `nothing`. Mathématiquement, si la branche est représentée par une fonction $\texttt{f} : \texttt{A} \to \texttt{B}$, on aimerait (automatiquement) être capable de produire une fonction

$$\begin{aligned} \texttt{g} : \texttt{Maybe(A)} &\to \texttt{Maybe(B)} \\ \texttt{just(x)} &\mapsto \texttt{just(f(x))} \\ \texttt{nothing} &\mapsto \texttt{nothing} \end{aligned}$$

De cette manière, on pourrait simplement écrire

```
auto maybe_result = safe_div(x, y);
g(maybe_result);
```

Comme on pouvait s'en douter, nous venons de mettre le doigt sur un foncteur. En effet, étant donné un objet `A` de la catégorie Hana, le foncteur `Maybe` envoie `A` sur `Maybe(A)`. Ensuite, étant donné une flèche $\texttt{f} : \texttt{A} \to \texttt{B}$, `Maybe` envoie `f` sur

$$\begin{aligned} \texttt{transform(-, f)} : \texttt{Maybe(A)} &\to \texttt{Maybe(B)} \\ \texttt{just(x)} &\mapsto \texttt{just(f(x))} \\ \texttt{nothing} &\mapsto \texttt{nothing} \end{aligned}$$

Afin de démontrer la fonctorialité de `Maybe`, commençons par regarder comment celui-ci est implémenté dans Hana. D'abord, on a `just` et `nothing`, qui représentent des valeurs optionnelles.

```
template <typename T>
struct _just { T value; };

auto just = [](auto x) {
    return _just<decltype(x)>{x};
};

struct _nothing { };
_nothing nothing{};
```

Ensuite, on a la fonction `transform`, qui applique une fonction à une valeur optionnelle et qui retourne un résultat optionnel. On notera l'utilisation de l'*overloading* dans l'implémentation :

```
template <typename T, typename F>
auto transform(_just<T> x, F f) {
```

```
    return just(f(x.value));
}

template <typename F>
auto transform(_nothing, F) {
    return nothing;
}
```

Si un `just(x)` est passé à `transform`, on applique la fonction `f` à `x` et on retourne le résultat comme un `Maybe`. Sinon, on retourne simplement `nothing`, puisqu'il n'y a pas de valeur à laquelle on pourrait appliquer `f`. Montrons maintenant que `Maybe` est un endofoncteur sur Hana. D'abord, on doit montrer que `transform`er une valeur optionnelle avec la fonction identité ne change pas cette valeur optionnelle. Soit donc `x` $\in$ `Maybe(A)` une valeur optionnelle de type `A`. Si `x` est de la forme `just(v)`, alors

```
transform(x, id) == transform(just(v), id)
                 == just(id(v))
                 == just(v)
                 == x
```

Si, au contraire, `x` est un `nothing`, alors

```
transform(x, id) == transform(nothing, id)
                 == nothing
```

ce qui montre que `transform(-, id) == id`. Ensuite, il faut montrer que la composition des fonctions est respectée par `transform`. Soit `x` $\in$ `Maybe(A)` une valeur optionnelle et `f` : `A` $\to$ `B` et `g` : `B` $\to$ `C` des fonctions quelconques. Si `x` est de la forme `just(v)`, alors

```
transform(x, compose(g, f)) == transform(just(v), compose(g, f))
                            == just(compose(g, f)(v))
                            == just(g(f(v)))
```

d'un autre côté, on a que

```
transform(transform(x, f), g) == transform(transform(just(v), f), g)
                              == transform(just(f(v)), g)
                              == just(g(f(v)))
```

ce qui montre l'égalité recherchée. Ainsi, `Maybe` est bien un foncteur sur la catégorie Hana. De plus, avec cette définition de `transform`, on peut maintenant écrire

```
auto increment = [](auto x) { return x + int_<1>; };

transform(safe_div(int_<4>, int_<2>), increment); // just(int_<3>)

transform(safe_div(int_<4>, int_<0>), increment); // nothing
```

ce qui est un gain d'expressivité par rapport à notre solution initiale. Cependant, qu'arrive-t-il si nous désirons composer plusieurs fonctions qui peuvent potentiellement échouer ? En d'autres mots, nous sommes présentemment capable de composer des fonctions de la forme `X` $\to$ `Maybe(Y)` et `Y` $\to$ `Z` ; la question est de savoir comment nous pourrions composer des fonctions de la forme `X` $\to$ `Maybe(Y)` et `Y` $\to$ `Maybe(Z)`. C'est en effet possible, mais il nous faudra attendre la section sur les monades pour y arriver.

4.3 Le foncteur Tuple

Jusqu'ici, nous avons vu comment faire des calculs arithmétiques et des transformations sur des types au moment de la compilation. Nous avons aussi vu comment il est possible de représenter une valeur optionnelle à l'aide d'un foncteur, ce qui est particulièrement utile pour représenter le résultat d'une fonction qui peut échouer. Nous allons maintenant voir comment il est possible de manipuler des séquences d'objets, et comment la notion de foncteur peut nous aider à formaliser ceci.

Étant donné une séquence d'objets $x_1, \ldots, x_n$ d'un ensemble X et une fonction $f : X \to Y$, il est parfois utile d'appliquer f à chaque élément de la séquence pour obtenir une nouvelle séquence $f(x_1), \ldots, f(x_n)$. Ce *pattern* étant omniprésent en programmation, on aimerait le formaliser dans Hana et le généraliser, si possible. Premièrement, il nous faut une séquence capable de contenir des objets d'un type généralisé `X` quelconque. On notera que puisque des objets d'un même type généralisé n'ont pas forcément le même type C++, on ne peut pas utiliser une séquence conventionnelle comme `std::vector`. On devra en effet utiliser une séquence hétérogène capable de contenir des objets ayant différents types C++. La véritable construction dans Hana est assez complexe pour des raisons techniques, mais l'idée est essentiellement la suivante :

```
template <int i, typename T>
struct _element { T value; };

template <typename T1, ..., typename Tn>
struct _tuple : _element<0, T1>, ..., _element<n-1, Tn> {
    // ... constructeur omis pour la simplicité
};

template <typename ...T>
_tuple<T...> make_tuple(T ...t) {
    return {t...};
}
```

On utilise le type `_element<i, T>` pour stocker l'objet de type `T` situé à l'index `i` dans la séquence. On se donne aussi un peu de sucre syntaxique (`make_tuple`), qui nous permettra de construire des `_tuple` sans devoir spécifier le type de chacun des éléments. La raison

pour laquelle on hérite de tous les `_element`s dans `_tuple` et que cela nous permet d'accéder à chaque élément individuel grâce à l'astuce suivante :

```
template <int i, typename T>
T get(_element<i, T> e) {
    return e.value;
}

char y = get<1>(make_tuple('x', 'y', 'z'));
```

Puisque `_tuple` hérite de tous les `_element`s, il existe une conversion de `_tuple` vers n'importe laquelle de ses classes de base. Ensuite, puisqu'on fixe le `i` explicitement en appelant `get<i>`, le compilateur constate que la seule signature possible pour `get` est `char get(_element<1, char>)`. Il suffit ensuite de retourner la valeur associée à cet élément. En bref, on a donc que `get<i>(make_tuple(x1, ..., xn)) == xi+1`.

Étant donné une séquence d'objets `x1`, ..., `xn` de type généralisé `X`, on dénotera par `Tuple(X)` le type généralisé de `make_tuple(x1, ..., xn)`. De cette manière, on considérera `Tuple` comme un type généralisé paramétré, où le paramètre représente le type généralisé des éléments que le `Tuple` contient. On notera en passant qu'il serait possible de stocker des éléments ayant des types généralisés différents au sein d'un même `_tuple`, puisque `_tuple` ne pose aucune restriction sur le type de ces éléments. Cependant, on restreindra notre étude aux `_tuple`s qui contiennent tous des éléments d'un même type généralisé, puisque c'est ceux-là qui nous donnent les propriétés les plus intéressantes.

Nous sommes maintenant en mesure de formuler concrètement le problème initial qui était d'appliquer une fonction à chaque élément d'une séquence. Étant donné une fonction `f : X` $\to$ `Y` et une séquence `make_tuple(x1, ..., xn)` de type généralisé `Tuple(X)`, on aimerait obtenir une séquence `make_tuple(f(x1), ..., f(xn))` (de type généralisé `Tuple(Y)`). Voici comment on pourrait faire ceci :

```
template <typename T1, ..., typename Tn, typename F>
auto transform(_tuple<T1, ..., Tn> ts, F f) {
    return make_tuple(f(get<0>(ts)), ..., f(get<n-1>(ts)));
}
```

Étant donné une séquence d'objets sous la forme d'un `_tuple`, on applique simplement `f` à chaque élément (extrait avec `get`), puis on retourne une nouvelle séquence qui contient le résultat. À ce point-ci, le foncteur caché sous le tapis devrait commencer à être évident.

En effet, en tant que foncteur, `Tuple` associe à tout objet `X` de *Hana* un nouvel objet `Tuple(X)` représentant une séquence d'éléments de type (généralisé) `X`, et à toute flèche `f : X` $\to$ `Y` une nouvelle flèche `transform(-, f) : Tuple(X)` $\to$ `Tuple(Y)`, qui consiste à appliquer `f` à chaque élément de la séquence.

Démontrons maintenant que `Tuple` est bien un foncteur. D'abord, étant donné un objet quelconque `xs` de type généralisé `Tuple(X)`, appliquer `transform` à `xs` et la fonction identité

sur `X` ne change pas `xs` du tout. En effet, en supposant que `xs` contienne n éléments, on a alors

```
transform(xs, id) == make_tuple(id(get<0>(xs)), ..., id(get<n-1>(xs)))
                  == make_tuple(get<0>(xs), ..., get<n-1>(xs))
                  == xs
```

Ainsi, on a bien que `transform(-, id) == id`, et `Tuple` respecte donc les flèches identités. Il reste à montrer que `Tuple` respecte la loi de composition, c'est-à-dire que pour toutes fonctions `f` : `X` → `Y` et `g` : `Y` → `Z`, on a

```
transform(transform(xs, f), g) == transform(xs, compose(g, f))
```

Or, on trouve facilement que

```
transform(transform(xs, f), g)
    == transform(make_tuple(f(get<0>(xs)), ..., f(get<n-1>(xs))), g)
    == make_tuple(g(f(get<0>(xs))), ..., g(f(get<n-1>(xs))))
    == make_tuple(compose(g, f)(get<0>(xs)), ..., compose(g, f)(get<n-1>(xs)))
    == transform(xs, compose(g, f))
```

ce qui montre que la loi de composition est bien respectée. Le fait d'exiger que `Tuple` soit un foncteur est bien plus qu'un simple caprice de mathématicien. En effet, exiger que la composition soit respectée par `transform` permet de faire des optimisations pour éliminer des structures temporaires inutiles. Par exemple, lorsque l'on applique

```
transform(transform(xs, f), g)
```

alors un `_tuple` intermédiaire qui contient `transform(xs, f)` doit être créé, puis envoyé aux deuxième `transform`. Si les objets retournés par `f` sont coûteux à copier, ceci peut représenter un problème. Or, la loi des foncteurs nous dit qu'on peut réécrire cette ligne comme `transform(xs, compose(g, f))`, qui ne crée pas de structure temporaire inutile.

5 Les transformations naturelles

Nous venons de voir la notion de foncteur, qui peut être vue comme une transformation entre deux catégories. Pour continuer dans la même direction, nous allons maintenant considérer ce que sont les transformations entre foncteurs. Ces transformations portent le nom de transformations naturelles et jouent un rôle fondamental en théorie des catégories.

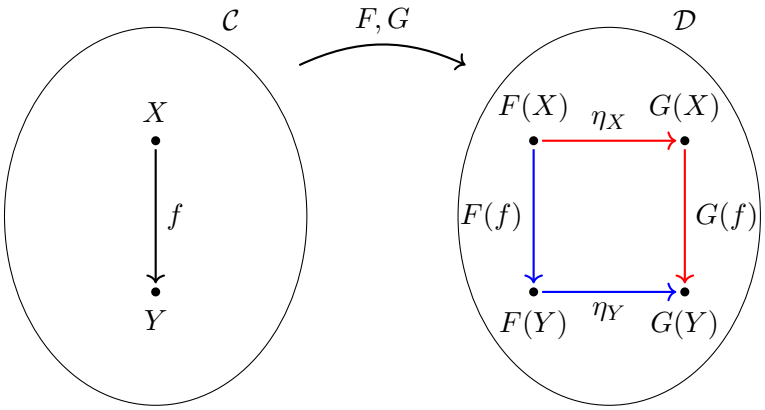
Définition (Transformation naturelle)**.** Soit $\mathcal{C}$ et $\mathcal{D}$ deux catégories. Une transformation naturelle $\eta : F \to G$ entre deux foncteurs $F : \mathcal{C} \to \mathcal{D}$ et $G : \mathcal{C} \to \mathcal{D}$ est une famille de flèches qui à chaque objet $X \in \text{ob}(\mathcal{C})$ associe une flèche $\eta_X : F(X) \to G(X) \in \hom(\mathcal{D})$ telle que pour tout objet $Y \in \text{ob}(\mathcal{C})$ et toute flèche $f : X \to Y \in \hom(\mathcal{C})$, on a

$$\eta_Y \circ F(f) = G(f) \circ \eta_X$$

Le fait que η soit une famille de flèches plutôt qu'une application peut sembler un peu déconcertant. Il suffit alors de se rappeler qu'une famille indexée n'est en fait qu'une fonction sous le couvert. Ainsi, on aurait aussi pu voir η comme une application $\eta : \text{ob}(\mathcal{C}) \to \hom(\mathcal{D})$ qui à chaque objet $X \in \text{ob}(\mathcal{C})$ associe une flèche $\eta(X) : F(X) \to G(X) \in \hom(\mathcal{D})$. On préfère ici la notation indexée parce qu'elle est plus commune dans la littérature et qu'elle nous évite certaines difficultés de notation un peu plus loin.

Assurons-nous maintenant que l'égalité $\eta_Y \circ F(f) = G(f) \circ \eta_X$ fait bien du sens. Du côté gauche, on a que $\eta_Y : F(Y) \to G(Y)$ est une flèche de $\mathcal{D}$ et que $F(f) : F(X) \to F(Y)$ est aussi une flèche de $\mathcal{D}$. Ainsi, la composition de ces deux flèches fait du sens et nous donne une flèche $\eta_Y \circ F(f) : F(X) \to G(Y)$ de $\mathcal{D}$. Du côté droit, on a que $G(f) : G(X) \to G(Y)$ est une flèche de $\mathcal{D}$ et que $\eta_X : F(X) \to G(X)$ est aussi une flèche de $\mathcal{D}$. Ainsi, la composition de ces deux flèches fait du sens et nous donne une flèche $G(f) \circ \eta_X : F(X) \to G(Y)$ de $\mathcal{D}$. Si elle n'est pas très intuitive, l'égalité de ces deux expressions fait donc au moins du sens.

Or, il existe une manière visuelle de se représenter cette équation qui nous aidera à en tirer un peu d'intuition. Il suffit de représenter les catégories $\mathcal{C}$ et $\mathcal{D}$ ainsi que l'action de F, G et η sur deux objets quelconques $X, Y \in \text{ob}(\mathcal{C})$ et une flèche $f : X \to Y$.



L'équation $\eta_Y \circ F(f) = G(f) \circ \eta_X$ peut se lire comme "η_Y après $F(f)$ doit être égal à $G(f)$ après η_X", ce qui veut simplement dire que les chemins bleus et rouges doivent en fait donner le même résultat, et ce peut importe les X, Y et f considérés. Ainsi, l'équation présentée plus haut peut se réécrire sous une forme plus visuelle en affirmant que le diagramme suivant commute :

$$\begin{array}{ccc} F(X) & \xrightarrow{\eta_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow{\eta_Y} & G(Y) \end{array}$$

5.1 Les transformations naturelles dans Hana

Dans Hana, un foncteur est un type généralisé paramétré et une flèche $f : X \to Y$ est une fonction d'un type généralisé `X` vers un type généralisé `Y`. Ainsi, étant donné deux types généralisés paramétrés `F` et `G` qui sont des foncteurs, une transformation naturelle `nat : F → G` sera une famille de fonctions qui à chaque type généralisé `X` associe une fonction `nat<X> : F(X) → G(X)`. De plus, la loi énoncée plus haut se traduit dans le langage de Hana en disant que pour tout type généralisé `Y` et toute fonction `f : X → Y`, on doit avoir

```
compose(nat<Y>, transform(-, f)) == compose(transform(-, f), nat<X>)
```

Par exemple, essayons d'écrire une fonction qui serait une transformation naturelle du foncteur `Maybe` vers le foncteur `Tuple`. On cherche donc une famille de fonctions qui à chaque type généralisé `X` associe une fonction `toTuple<X> : Maybe(X) → Tuple(X)`. La définition la plus évidente d'une telle fonction serait

```
toTuple<X> : Maybe(X) → Tuple(X)
             just(x)  ↦ make_tuple(x)
             nothing  ↦ make_tuple()
```

ce qui se traduit dans Hana par

```
template <typename X, typename T>
auto toTuple(_just<T> j) {
    return make_tuple(from_just(j));
}

template <typename X>
auto toTuple(_nothing) {
    return make_tuple();
}
```

A-t-on vraiment une transformation naturelle ? Si c'est le cas, on devrait avoir que pour tout type généralisé `Y` et toute fonction `f` : `X` $\to$ `Y`,

```
compose(toTuple<Y>, transform(-, f)) == compose(transform(-, f), toTuple<X>)
```

ce qui est équivalent à dire que pour tout objet `m` de type généralisé `Maybe(X)`,

```
toTuple<Y>(transform(m, f)) == transform(toTuple<X>(m), f)
```

Énoncé de cette manière, il est assez évident que c'est le cas. En effet, si `m` est un `just(x)`, alors

```
toTuple<Y>(transform(just(x), f)) == toTuple<Y>(just(f(x)))
                                  == make_tuple(f(x))
                                  == transform(make_tuple(x), f)
                                  == transform(toTuple<X>(just(x)), f)
```

De manière similaire, si `m` est un `nothing`, alors

```
toTuple<Y>(transform(nothing, f)) == toTuple<Y>(nothing)
                                  == make_tuple()
                                  == transform(make_tuple(), f)
                                  == transform(toTuple<X>(nothing), f)
```

et `toTuple` est donc bien une transformation naturelle. Cependant, cette définition n'est pas la seule qui permette à `toTuple` d'être une transformation naturelle. En effet, on aurait aussi pu considérer

```
template <typename X, typename T>
auto toTuple2(_just<T> j) {
    return make_tuple(from_just(j), from_just(j));
}

template <typename X>
auto toTuple2(_nothing) {
    return make_tuple();
}
```

Ici, la différence est que l'on crée un tuple avec deux copies du même élément lorsqu'on a un `just(x)`. Il est facile de voir que c'est une transformation naturelle. En effet, on a

```
toTuple2<Y>(transform(just(x), f)) == toTuple2<Y>(just(f(x)))
                                   == make_tuple(f(x), f(x))
                                   == transform(make_tuple(x, x), f)
                                   == transform(toTuple2<X>(just(x)), f)
```

Ceci est en fait une manifestation du résultat plus général suivant.

Théorème. *Soit $F, G, H : \mathcal{C} \to \mathcal{D}$ des foncteurs et $\alpha : F \to G$, $\beta : G \to H$ des transformations naturelles. Alors la famille de flèches $\beta \circ \alpha$ définie par $(\beta \circ \alpha)_X = \beta_X \circ \alpha_X$ est une transformation naturelle $F \to H$. En d'autres mots, la composition de deux transformations naturelles est une transformation naturelle.*

Démonstration. Soit $X, Y \in \mathrm{ob}(\mathcal{C})$ des objets de $\mathcal{C}$ et $f : X \to Y \in \hom(\mathcal{C})$ une flèche de $\mathcal{C}$. Premièrement, $\beta_X \circ \alpha_X$ est bien une flèche $F(X) \to H(X)$, puisque α_X est une flèche $F(X) \to G(X)$ et β_X est une flèche $G(X) \to H(X)$. Ensuite, on a

$$\begin{aligned}
(\beta \circ \alpha)_Y \circ F(f) &= (\beta_Y \circ \alpha_Y) \circ F(f) && \text{(par définition)} \\
&= \beta_Y \circ (G(f) \circ \alpha_X) && \text{(par la loi)} \\
&= (H(f) \circ \beta_X) \circ \alpha_X && \text{(par la loi, encore)} \\
&= H(f) \circ (\beta \circ \alpha)_X && \text{(par définition)}
\end{aligned}$$

ce qui montre que $\beta \circ \alpha$ est naturelle. □

Dans l'exemple plus haut, `toTuple2` était simplement la composition de la transformation naturelle `toTuple` définie plus haut avec la transformation naturelle `twice` définie par

```
twice<X> : Tuple(X) → Tuple(X)
make_tuple(x1, ..., xn) ↦ make_tuple(x1, ..., xn, x1, ..., xn)
```

6 Les monades

Une monade est un endofoncteur avec davantage de structure. Cette structure additionnelle nous permet de représenter plusieurs phénomènes courants en programmation qui, avant l'arrivée des monades, étaient considérés comme complètement déconnectés des mathématiques.

Définition (Monade)**.** Soit $\mathcal{C}$ une catégorie et $I_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ le foncteur identité sur $\mathcal{C}$. Une monade est un endofoncteur $F : \mathcal{C} \to \mathcal{C}$ accompagné de deux transformation naturelles $\eta : I_{\mathcal{C}} \to F$ et $\mu : F \circ F \to F$ telles que pour tout objet $X \in \text{ob}(\mathcal{C})$ et toute flèche $f : X \to Y \in \hom(\mathcal{C})$,

1. $\mu_X \circ F(\mu_X) = \mu_X \circ \mu_{F(X)}$
2. $\mu_X \circ F(\eta_X) = \mu_X \circ \eta_{F(X)} = id_{F(X)}$
3. $\eta_Y \circ f = F(f) \circ \eta_X$
4. $\mu_Y \circ F(F(f)) = F(f) \circ \mu_X$

6.1 Les monades dans Hana

Pour déchiffrer cette définition un peu opaque, regardons maintenant comment une monade est représentée dans Hana. Premièrement, on donne un nom plus descriptif aux transformations naturelles η et μ. Puisque η associe à tout objet $X \in \text{ob}(\mathcal{C})$ une flèche $\eta_X : I_{\mathcal{C}}(X) = X \to F(X)$, on l'appellera `lift`. Plus précisément, étant donné un type généralisé paramétré `F` et un type généralisé `X`, `lift<X>` : `X` $\to$ `F(X)` prend un objet de `X` et "l'injecte" ou le "monte" dans le foncteur `F`.

Ensuite, puisque μ associe à tout objet $X \in \text{ob}(\mathcal{C})$ une flèche $\mu_X : F(F(X)) \to F(X)$, on l'appellera `flatten`. Ici, l'intuition est que `flatten<X>` : `F(F(X))` $\to$ `F(X)` prend un objet enveloppé dans deux niveaux de fonctorialité et élimine, ou écrase, un niveau de fonctorialité. Pour alléger la notation, on notera parfois simplement `flatten` et `lift` au lieu de `flatten<...>` et `lift<...>`. En reformulant la définition d'une monade de cette manière, on obtient que les deux transformations naturelles doivent respecter les équations suivantes. D'abord, on doit avoir que

```
compose(flatten<X>, transform(-, flatten<X>))
    == compose(flatten<X>, flatten<F(X)>)
```

ce qui est équivalent à dire que pour tout objet `xs` $\in$ `F(F(F(X)))`,

```
flatten(transform(xs, flatten)) == flatten(flatten(xs))
```

Considérant `xs` comme une structure à "3 niveaux", cette loi dit donc qu'écraser les niveaux intérieurs puis ensuite les niveaux extérieurs est équivalent à écraser les niveaux extérieurs puis ensuite les niveaux intérieurs. Quant à la deuxième loi, elle peut s'écrire comme

```
compose(flatten<X>, transform(-, lift<X>))
    == compose(flatten<X>, lift<F(X)>)
    == id
```

ce qui est équivalent à dire que pour tout objet `xs` $\in$ `F(X)`,

```
flatten(transform(xs, lift)) == flatten(lift(xs)) == xs
```

D'abord, à droite on a `flatten(lift(xs)) == xs`, ce qui veut simplement dire que monter une valeur dans un foncteur puis écraser ce niveau de fonctorialité devrait être équivalent à ne rien faire du tout. À gauche, on a que `flatten(transform(xs, lift)) == xs`, ce qui veut dire que monter les valeurs dans un foncteur puis ensuite écraser ce nouveau niveau de fonctorialité devrait être équivalent à ne rien faire. Cette loi permet de s'assurer que `flatten` et `lift` ne cachent pas d'effets arbitraires. Pour la suite, supposons que `f` est une fonction `X` $\to$ `Y`. La troisième loi peut s'écrire comme

```
compose(lift<Y>, f) == compose(transform(-, f), lift<X>)
```

De manière équivalente, on peut dire que pour tout objet `x` $\in$ `X`,

```
lift(f(x)) == transform(lift(x), f)
```

Ceci représente le fait qu'appliquer une fonction à une valeur simple montée avec `lift` à l'intérieur du foncteur est la même chose que d'effectuer la transformation à l'extérieur du foncteur, puis de monter le résultat dans le foncteur. Finalement, étant donné un objet `xs` $\in$ `F(F(X))`, on peut écrire la quatrième loi comme

```
flatten(transform(xs, transform(-, f))) == transform(flatten(xs), f)
```

Il est plus difficile de développer une intuition pour cette loi, mais les exemples de monades qui suivent la rendront probablement plus claire. Avant de donner des exemples utiles de l'utilisation des monades dans Hana, il nous faut introduire une fonction nommée `bind`, qui sert à enchaîner des fonctions dont le retour est monadique. Plus précisément, `bind` est définie comme

$$\begin{aligned} \texttt{bind} : \texttt{F(X)} \times (\texttt{X} \to \texttt{F(Y)}) &\to \texttt{F(Y)} \\ \texttt{xs} \times \texttt{f} &\mapsto \texttt{flatten(transform(xs, f))} \end{aligned}$$

Étant donné une valeur monadique `xs` $\in$ `F(X)` et une fonction à retour monadique `f` : `X` $\to$ `F(Y)`, `bind` applique la fonction `f` à l'intérieur de `xs`, ce qui nous donne un objet de type `F(F(Y))`. Ensuite, `bind` applique `flatten` pour obtenir un résultat final de type `F(Y)`. Une manière possible de voir `bind` est comme l'application d'une fonction à une valeur, mais une application à laquelle on aurait ajouté des effets représentés par la monade `F`. Pour alléger la notation, Hana définit l'opérateur `operator|` comme étant équivalent à `bind` : `xs | f == bind(xs, f)`. Nous utiliserons parfois cette notation par la suite. Nous sommes maintenant prêts pour voir des exemples concrets de Monades qui sont utilisées dans Hana.

6.2 La monade Maybe

Nous avons vu précédemment que `Maybe` était un foncteur. En fait, `Maybe` est plus qu'un foncteur ; c'est aussi une monade. D'abord, l'opération `lift` devrait avoir une signature `X` $\to$ `Maybe(X)`. La définition la plus naturelle pour une fonction ayant cette signature est `lift` = `just`.

Ainsi, `lift` prend une valeur normale et la "monte" dans une valeur optionnelle. Ensuite, la fonction `flatten` devrait avoir une signature `Maybe(Maybe(X))` $\to$ `Maybe(X)`. Encore une fois, la définition la plus naturelle est la bonne :

$$\begin{aligned} \texttt{flatten} : \texttt{Maybe(Maybe(X))} &\to \texttt{Maybe(X)} \\ \texttt{just(just(x))} &\mapsto \texttt{just(x)} \\ \texttt{just(nothing)} &\mapsto \texttt{nothing} \\ \texttt{nothing} &\mapsto \texttt{nothing} \end{aligned}$$

Montrons maintenant qu'il s'agit bien d'une monade. Pour montrer que la première loi tient, considérons `mmm` $\in$ `Maybe(Maybe(Maybe(X)))`. Il y a quatre cas possibles.

1. Si `mmm` = `nothing`, alors

```
flatten(transform(nothing, flatten))
    == flatten(nothing)
    == nothing
    == flatten(flatten(nothing))
```

2. Si `mmm` = `just(nothing)`, alors

```
flatten(transform(just(nothing), flatten))
    == flatten(just(flatten(nothing)))
    == flatten(just(nothing))
    == nothing
    == flatten(flatten(just(nothing)))
```

3. Si `mmm` = `just(just(nothing))`, alors

```
flatten(transform(just(just(nothing)), flatten))
    == flatten(just(flatten(just(nothing))))
    == flatten(just(nothing))
    == nothing
    == flatten(flatten(just(just(nothing))))
```

4. Si au contraire on a `mmm` = `just(just(just(x)))`, alors

```
flatten(transform(just(just(just(x))), flatten))
    == flatten(just(flatten(just(just(x)))))
    == flatten(just(just(x)))
    == flatten(flatten(just(just(just(x)))))
```

Ceci montre que la première loi est bien respectée. Pour la deuxième loi, considérons maintenant `m` $\in$ `Maybe(X)`. Il y a deux cas possibles.

1. Si `m` = `nothing`, alors

```
flatten(transform(nothing, lift)) == flatten(nothing)
                                  == nothing
                                  == flatten(just(nothing))
                                  == flatten(lift(nothing))
```

2. Si `m` = `just(x)`, alors

```
flatten(transform(just(x), lift)) == flatten(just(lift(x)))
                                  == flatten(just(just(x)))
                                  == flatten(lift(just(x)))
                                  == just(x)
```

La troisième loi est facile à montrer. Soit `x` $\in$ `X` et `f` : `X` $\to$ `Y`. Alors

```
lift(f(x)) == just(f(x))
           == transform(just(x), f)
           == transform(lift(x), f)
```

Finalement, pour un `mm` $\in$ `F(F(X))`, la quatrième loi nous donne

1. Si `mm` = `nothing`, alors

```
flatten(transform(nothing, transform(-, f)))
    == flatten(nothing)
    == nothing
    == transform(nothing, f)
    == transform(flatten(nothing), f)
```

2. Si `mm` = `just(nothing)`, alors

```
flatten(transform(just(nothing), transform(-, f)))
    == flatten(just(transform(nothing, f)))
    == flatten(just(nothing))
    == nothing
    == transform(nothing, f)
    == transform(flatten(just(nothing)), f)
```

3. Si `mm` = `just(just(x))`, alors

```
flatten(transform(just(just(x)), transform(-, f)))
    == flatten(just(transform(just(x), f)))
    == flatten(just(just(f(x))))
    == just(f(x))
```

```
== transform(just(x), f)
== transform(flatten(just(just(x))), f)
```

Ceci montre que `Maybe` est bien une monade. Voici une manière de nous en servir. D'abord, définissons la fonction `sfinae`, qui nous permettra de manipuler des expressions potentiellement invalides sans causer d'erreur de compilation. Étant donné une fonction `f`, `sfinae(f)` est une fonction qui applique `f` à son argument si cet appel est valide, et qui retourne `nothing` sinon. Voici comment on peut l'implémenter :

```
template <typename F, typename ...X>
constexpr auto sfinae_impl(int, F f, X ...x) -> decltype(just(f(x...))) {
    return just(f(x...));
}

template <typename F, typename ...X>
constexpr auto sfinae_impl(long, F, X ...) {
    return nothing;
}

auto sfinae = [](auto f) {
    return [=](auto ...args) {
        return sfinae_impl(int{}, f, args...);
    };
};
```

D'abord, `sfinae(f)` est juste une fonction qui appelle `sfinae_impl` avec les arguments qu'on lui passe. On aurait aussi pu décider de passer les arguments directement à `sfinae`, i.e. d'écrire `sfinae(f, args...)` au lieu de `sfinae(f)(args...)`. On décide de curryifier le premier argument de `sfinae` parce que ceci facilite généralement son utilisation. Le corps de la technique se passe dans l'appel à `sfinae_impl`. D'abord, puisque `sfinae_impl` est surchargée mais que `sfinae` l'appelle avec un argument de type `int`, la première version sera préférée par le compilateur (la deuxième version nécessite une conversion implicite de `int` vers `long`). Cependant, si l'expression `just(f(args...))` est invalide, la première version de `sfinae_impl` possédera alors une signature invalide, et le compilateur devra plutôt utiliser la deuxième surcharge, sans toutefois causer d'erreur de compilation parce que la première fonction ne pouvait pas être choisie. Ce phénomène appelé *SFINAE* (*Substitution Failure Is Not An Error*) est couramment utilisé dans les bibliothèques de programmation générique pour tester la validité d'une expression. Donc, si `just(f(args...))` est une expression valide, alors la première fonction sera choisie et le résultat sera `just(f(args...))`. Sinon, la deuxième fonction sera choisie et le résultat sera `nothing`. Par exemple, définissons un type `Person` avec deux attributs ainsi que des fonctions permettant d'accéder à ces attributs :

```
struct Person {
```

```
    std::string name;
    unsigned int age;
    // ...
};

auto name(Person p) { return p.name; }
auto age(Person p) { return p.age; }
```

Comme on s'y attendrait, tenter d'appeler `age` ou `name` sur autre chose qu'un objet de type `Person` produirait une erreur de compilation, puisque `name` est définie seulement pour des objets de type `Person` :

```
// error: no matching function for call to 'name'
name(1);
```

Pour éviter une erreur de compilation et prendre en main la gestion de l'invalidité de l'appel, nous pouvons utiliser `sfinae` :

```
Person john{"John", 30};
sfinae(name)(john); // just("John")
sfinae(name)(1);    // nothing
```

Mais qu'arrive-t-il si nous souhaitons composer des fonctions qui peuvent échouer ? C'est là que la monadicité de `Maybe` va nous venir en aide. Considérons `f` une fonction qui prend un pointeur vers une `Person` et qui renvoie la longueur de son nom. On voudrait de plus que `f` nous renvoie un `nothing` plutôt que de faire échouer la compilation si *n'importe quelle étape* de la chaîne est invalide :

```
auto deref = [](auto x) -> decltype(*x) { return *x; };
auto length = [](auto x) -> decltype(x.length()) { return x.length(); };

auto f = [](auto x) {
    return sfinae(deref)(x) | sfinae(name)
                            | sfinae(length);
};
```

On utilise `bind` (sous sa forme `operator|`) pour créer une composition monadique des fonctions `sfinae(deref)`, `sfinae(name)` et `sfinae(length)`. De cette manière,

```
// 'nothing' car on tente de déréférencer un non-pointeur.
f(john);

// 'nothing' car on tente d'accéder à l'attribut 'name' dans un
// objet de type 'int', qui n'en a pas.
```

```
f(static_cast<int*>(0));

// ‘nothing‘ car on tente d’appeler ‘.length()‘ sur un objet de
// type ‘long‘, ce qui est invalide.
struct Foo { long name; };
Foo foo;
f(&foo);

// ‘just(4u)‘; tout va bien.
f(&john);
```

Ainsi, la monade `Maybe` nous permet de composer des fonctions qui peuvent échouer. Si on ne s'était pas rendu compte qu'il s'agissait d'une monade, il est peu probable que nous aurions trouvé une interface aussi composable pour manipuler ces fonctions. De plus, le fait que `Maybe` soit une monade rend possible l'utilisation de plusieurs algorithmes génériques qui fonctionnent pour n'importe quelle monade, par exemple les accumulations monadiques (*monadic folds*), qui ne sont pas traités ici. Les algorithmes de cette famille peuvent être utilisés (avec `Maybe`) pour réduire une structure à l'aide d'une opération qui peut échouer à chaque étape de réduction, ce qui mène à un gain d'expressivité et de généralité significatif sur les alternatives existantes en métaprogrammation.

6.3 La monade Tuple

Un autre foncteur que nous avons vu dans une section précédente est `Tuple`. Tout comme `Maybe`, il s'agit non seulement d'un foncteur mais aussi d'une monade. Afin d'alléger la notation, on dénotera par `[x1, ..., xn]` l'expression `make_tuple(x1, ..., xn)`. De plus, on écrira une séquence de séquences de la manière suivante

$$\texttt{[[x11, ..., x1m], ..., [xn1, ..., xnk]] == [[x...]...]}$$

et on dénotera la concaténation de ces séquences par

$$\texttt{[x11, ..., x1m, ..., xn1, ..., xnk] == [x... ...]}$$

On peut maintenant définir `lift` comme

$$\begin{aligned} \texttt{lift} : \texttt{X} &\to \texttt{Tuple(X)} \\ \texttt{x} &\mapsto \texttt{[x]} \end{aligned}$$

`lift` prend simplement un élément et le met dans un tuple singleton. Ensuite, on peut définir `flatten` comme

$$\begin{aligned} \texttt{flatten} : \texttt{Tuple(Tuple(X))} &\to \texttt{Tuple(X)} \\ \texttt{[[x...]...]} &\mapsto \texttt{[x... ...]} \end{aligned}$$

Montrons maintenant que `Tuple` est bien une monade. D'abord, pour montrer la première loi, considérons `xss` $\in$ `Tuple(Tuple(Tuple(X)))`. On a alors

```
flatten(transform([[[x...]...]...], flatten))
    == flatten([flatten([[x...]...]])...])
    == flatten([[x... ...]...])
    == flatten(flatten([[[x...]...]...]))
```

Ensuite, pour la deuxième loi, on a bien que pour tout `xs = [x...]` $\in$ `Tuple(X)`,

```
flatten(transform([x...], lift)) == flatten([lift(x)...])
                                 == flatten([[x]...])
                                 == [x...]
                                 == flatten([[x...]])
                                 == flatten(lift([x...]))
```

La troisième loi est facile à voir. Étant donné un objet `x` $\in$ `X` et une fonction `f` : `X` $\to$ `Y`, on a bien

```
lift(f(x)) == [f(x)]
           == transform([x], f)
           == transform(lift(x), f)
```

Finalement, la quatrième et dernière loi est elle aussi respectée, puisque pour un objet `xs = [[x...]...]` $\in$ `Tuple(Tuple(X))`, on a

```
flatten(transform([[x...]...], transform(-, f)))
                    == flatten([transform([x...], f)...])
                    == flatten([[f(x)...]...])
                    == [f(x)... ...]
                    == transform([x... ...], f)
                    == transform(flatten([[x...]...]), f)
```

Ceci montre que `Tuple` est bien une monade. Comment ceci peut-il nous être utile ? Il se trouve que `Tuple` peut être utilisé pour composer des fonctions "non déterministes". En effet, une fonction `f` : `X` $\to$ `Tuple(Y)` peut être vue comme une fonction non déterministe qui associe plusieurs résultats possibles à chaque valeur d'entrée. Or, la fonction `bind` associée à `Tuple` possède la signature `Tuple(X)` $\times$ (`X` $\to$ `Tuple(Y)`) $\to$ `Tuple(Y)`. Ainsi, elle permet de passer le résultat d'une fonction ayant plusieurs valeurs possibles (un `Tuple(X)`) à une fonction non déterministe acceptant un seul `X`. En utilisant `bind` en chaîne, il est possible de créer un pipeline de transformations non déterministes sans que cela ne soit trop encombrant. Par exemple, supposons qu'on ait une fonction générique f :

```
template <typename T>
auto f(T&& t) {
    // ...
}
```

`f` étant un *template*, son corps n'est pas instancié par le compilateur avant qu'on en fasse la requête pour un type `T` explicite. Ainsi, il est possible que la fonction soit invalide pour certains choix de `T` (pour lesquels on aimerait qu'elle soit valide), mais que cette erreur ne soit attrapée par le compilateur que beaucoup plus tard, lorsqu'on appelle la fonction avec un des types `T` problématique. Ceci pose particulièrement problème pour tester les bibliothèques génériques qui sont presque uniquement constituées de *templates*. Or, il se trouve que la monade `Tuple` peut nous aider à s'assurer que la fonction `f` compile bien avec toutes les variations possibles d'un type donné. D'abord, écrivons une fonction `cv_qualifiers : Type → Tuple(Type)` qui associe à un `Type` une liste de ses variations *cv-qualifiées* :

```
template <typename T>
auto cv_qualifiers(_type<T>) {
    return make_tuple(
        type<T>, type<T const>, type<T volatile>, type<T const volatile>
    );
}
```

Ainsi, étant donné un type `T`, `cv_qualifiers` retourne un `Tuple` qui contient toutes les variations possibles de qualifications `const` et `volatile`. De manière similaire, définissons une fonction `ref_qualifiers : Type → Tuple(Type)` qui associe à un type une liste des références possibles à ce type.

```
template <typename T>
auto ref_qualifiers(_type<T>) {
    return make_tuple(type<T&>, type<T&&>);
}
```

Maintenant, étant donné un type `T`, disons `int`, il nous est possible de générer toutes les combinaisons de qualificatifs `const-volatile` et de références en `bind`ant le retour de `cv_qualifiers` dans l'entrée de `ref_qualifiers`, ou vice-versa. Finalement, on peut utiliser la fonction `sfinae` présentée plus tôt pour s'assurer que la fonction `f` compile bien avec toutes les variations de types d'arguments qui nous intéressent :

```
// args == [int&, int&&, int const&, in const&&, int volatile&,
//          int volatile&&, int const volatile&, int const volatile&&]
auto args = bind(cv_qualifiers(type<int>), ref_qualifiers);
```

```
transform(args, [](auto arg) {
    using T = typename decltype(arg)::type;
    using Valid = decltype(is_just(sfinae(f)(std::declval<T>())));
    static_assert(Valid::value, "");

    return 0; // on doit retourner quelque chose pour transform
});
```

D'abord, on génère la séquence de tous les types d'arguments qu'on souhaite peut-être envoyer à `f`. Ensuite, on utilise `transform` pour appliquer une fonction anonyme à chacun de ces types. Étant donné un type d'argument sous la forme d'un `type<T>`, cette fonction anonyme extrait le type `T`, puis utilise `sfinae` pour s'assurer que l'appel à la fonction `f` avec ce type d'argument est bien formé.

7 Conclusion

Nous avons introduit les concepts de base de la métaprogrammation en C++, puis nous avons présenté certains outils qui permettent de faire des calculs au moment de la compilation. Nous avons ensuite introduit des concepts de base de la théorie des catégories en insistant sur l'intuition, puis nous avons présenté une formalisation de la métaprogrammation à l'aide de ces notions. Finalement, nous avons montré comment cette formalisation peut aider à obtenir des interfaces expressives et composables à l'aide d'exemples concrets.

Les contributions originales de ce travail sont l'introduction d'un système permettant d'exprimer des calculs au moment de la compilation avec la même syntaxe que le C++ usuel, la représentation de l'univers de la métaprogrammation statique à l'aide d'une catégorie, ainsi que l'observation que le phénomène de *SFINAE* peut être modélisé à l'aide d'une monade de gestion des erreurs.

Le langage C++ étant en plein changement, de nouvelles opportunités s'ouvriront certainement pour pousser plus loin l'étude de la métaprogrammation statique. De plus, avec la présence grandissante du paradigme fonctionnel dans les langages populaires, il y a fort à parier que de nouvelles notions inspirées de la théorie des catégories feront leur apparition en C++, ouvrant au passage un monde de possibilités pour les designers de bibliothèques génériques.

Références

[1] David Abrahams and Aleksey Gurtovoy. *C++ template metaprogramming : concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[2] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. *ACM SIGPLAN Notices*, 41(1) :206–217, 2006.

[3] Joel de Guzman, Dan Marsden, and Tobias Schwinger. The Boost Fusion library. `http://www.boost.org/doc/libs/release/libs/fusion/doc/html/index.html`, 2008.

[4] Louis Dionne. The Hana library. `http://github.com/ldionne/hana`, 2015.

[5] Aleksey Gurtovoy and David Abrahams. The Boost MPL library. `http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html`, 2002.

[6] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1978.

[7] Barry Mazur. When is one thing equal to some other thing ? 2007.

[8] Bartosz Milewski. What does Haskell have to do with C++ ? `http://bartoszmilewski.com/2009/10/21/what-does-haskell-have-to-do-with-c`, 2009.

[9] Wikibooks. Haskell/category theory. `http://en.wikibooks.org/w/index.php?title=Haskell/Category_theory`, 2015.