

Présentation de la programmation logique avec Castor

par Roshan Naik ([Site perso de Roshan Naik](#))
Traduction par Come David ([Site perso de David Come](#))

Date de publication : 04/01/2010

Cet article a pour but de vous introduire les grands concepts de la programmation logique puis sa mise en pratique en C++ : Castor. Le lecteur n'est pas obligé d'avoir des bases en programmation logique, seules quelques bases en C++ sont nécessaires.

I - Introduction.....	3
II - La programmation logique.....	4
II-1 - Les faits.....	4
II-2 - les règles.....	5
II-2-A - Les règles récursives.....	5
II-3 - Requêtes et assertions.....	5
II-4 - Calcul par inférence.....	6
II-5 - Résumé.....	6
III - Programmation logique en C++.....	7
III-1 - Le type relation.....	9
III-2 - lref: La référence logique.....	9
III-3 - La relation eq: La fonction d'unification.....	10
III-4 - Évaluation des requêtes.....	10
III-5 - Les règles récursives.....	12
III-6 - Relations dynamiques.....	13
III-7 - Inline Logic Reference Expressions (ILE).....	14
III-8 - Conteneurs et séquences.....	15
III-8-A - Génération des séquences.....	15
III-8-2 - Itération dans une séquence.....	17
III-8-2-A - Itération avec head et tail.....	17
III-8-2-B - Itération avec next et prev.....	18
III-8-2-C - Itération avec item.....	18
III-8-3 - Unification d'ensemble.....	19
III-8-4 - Résumé.....	19
III-9 - Les coupes, des élagages alternatifs.....	20
III-10 - Opérateur ou-exclusif.....	21
III-11 - Manières de passer une référence logique à une relation.....	22
III-11-A - Par valeur.....	22
III-11-B - Par référence.....	22
III-11-C - Par référence constante.....	22
III-12 - Debug.....	23
IV - Créer des relations impératives.....	25
IV-1 - Avec la relation predicate.....	25
IV-2 - En tant que coroutine.....	25
V - Inline Logic Reference Expressions.....	28
V-1 - Créer des relations à partir d'ILEs.....	29
V-2 - Limitations des ILEs.....	30
V-3 - Résumé.....	31
VI - Limitation du paradigme logique.....	32
VI-1 - La bi-directionnalité des références logiques.....	32
VI-2 - Les entrés/sorties ne sont pas réversibles.....	32
VII - Exemples de programmes logiques.....	34
VII-1 - Graphe acyclique direct.....	34
VII-2 - Automate à états finis.....	35
VIII - Conclusion.....	38

I - Introduction

Cet article est un tutoriel d'introduction à la programmation logique (PL) en C++. Aucune connaissance préalable n'est requise dans des langages qui supportent nativement la programmation logique. Cet article montre aussi comment la PL se lie avec les autres paradigmes supportés par le C++ et avec la STL. La capacité à choisir un paradigme approprié ou un mélange de paradigmes pour résoudre un problème donné est fondamentale et le coeur de la programmation multi-paradigme. Nous commencerons par une brève introduction à la PL, suivie par une discussion de la programmation logique en C++ pour finir avec des exemples. Le support de la programmation logique est fourni par Castor, une bibliothèque Open Source disponible sur www.mpprogramming.com. Elle ne nécessite pas d'extension au langage pour compiler les codes sources fournis ici.

II - La programmation logique

La programmation logique est un modèle Turing complet (NdT: cela veut dire qu'on peut coder avec tout ce qu'on peut imaginer) La façon de programmer en utilisant le paradigme logique diffère fondamentalement de la manière utilisée avec la programmation impérative ou fonctionnelle. Des programmes se basant uniquement sur la PL sont entièrement déclaratifs. Quand on programme avec des langages basés sur un paradigme impératif (comme C, C++, Java, ...), les programmeurs dictent à l'ordinateur comment résoudre le problème et l'ordinateur n'a pas connaissance du problème en lui même. Ainsi, les algorithmes jouent un rôle central dans ce modèle de programmation.

Dans des langages de programmation logique tel Prolog ou Gödel, c'est exactement l'inverse qui se produit. En PL, le rôle du programmeur est de fournir des informations sur le problème à l'ordinateur et non de fournir un moyen de résoudre le problème. C'est l'ordinateur qui va appliquer un algorithme général de solution du problème pour obtenir le résultat. Le programmeur n'est pas impliqué dans les étapes qui vont résoudre le problème (l'algorithme).

Les informations fournies à l'ordinateur peuvent être classées en deux catégories: les **faits** et les **règles**. Cette base de connaissance décrit le domaine du problème. Un problème spécifique peut être résolu en posant une question ou **requête**. L'ordinateur examine les requêtes dans le contexte fourni par les règles et les faits et détermine la solution. Par exemple, si les échecs (ou un autre jeu de plateau) représente le domaine de notre problème, les faits peuvent être des choses comme:

- Les différents types de pièces (par exemple pion blanc, pion noir, roi blanc, ...)
- Le nombre de pièce de chaque type (8 pions noirs, 1 roi blanc, ...)
- La description du plateau et de son organisation (plateau 8x8, 32 cases noires, 32 blanches, ...)

Et les règles peuvent par exemple comprendre:

- Les mouvements possibles pour chaque type de pièce (exemple: le fou se déplace selon les diagonales)
- Les règles pour déterminer si un pièce est attaquée ou non
- Les règles pour déterminer quand une partie est finie et le vainqueur

Voici le genre de question qu'on peut poser à propos de notre jeu d'échec:

- Étant donné un placement des pièces sur le plateau, quels sont les mouvements disponibles pour une pièce ?
- A partir de la disposition d'un plateau, quelles pièces peuvent bouger ?
- A partir de la disposition d'un plateau, quelles pièces sont attaquées ?

Chaque question précédente correspond à un problème différent mais concret dans le domaine des échecs. Changer le point de vue en passant de "comment résoudre un problème particulier" à "décrire les règles générales d'un problème plus large" nous autorise à chercher des réponses plus variées à des problèmes à l'intérieur du domaine. Dans le reste de cette section, nous illustrerons plus en détail les notions de faits, règles et requêtes en utilisant un exemple de relations familiales. Le but premier est de bien se familiariser avec les mécanismes basiques de la PL.

II-1 - Les faits

Les faits sont essentiellement la forme la plus élémentaire d'assertions vérifiées relatives au domaine du problème. Les faits sont aussi connus en tant que données. Prenons une famille de quatre personnes (le fils: Sam, la fille: Denise, le père: Franck, la mère: Mary et le grand-père: Garry) Voici comment on peut décrire les faits afférents à cette famille de façon plus exacte:

- 1 Sam est un enfant de Mary
- 2 Denise est un enfant de Mary
- 3 Sam est un enfant de Franck
- 4 Denise est un enfant de Franck
- 5 Franck est un enfant de Garry

- 1 Franck est un homme
- 2 Sam est un homme

- 3 Mary est une femme
- 4 Denise est une femme

Les faits présentés au dessus peuvent être utilisés afin de répondre à des questions simples comme *Est ce que Sam est un homme ?*. Cette question basique dont la réponse est oui ou non peut être résolue en regardant les faits portants sur les genres. Étant donné que nous avons une correspondance exacte avec le fait 2 de la seconde liste, la réponse est *oui* ou encore *vrai*. Cependant, la question *Est ce que Sam est une femme ?* retourne *non* ou *faux*. Ceci s'explique car nous n'avons aucune donnée disant que Sam est une femme. Ainsi la réponse par défaut à une requête est non, à moins qu'une correspondance exacte se fasse.

Un type de question légèrement différent est "Quel est le genre de Sam ?" Cette question n'appelle pas une réponse du type vrai/faux mais on peut y répondre en regardant le fait 2 de la seconde liste. Une autre question pourrait être "Qui est l'enfant de Franck ?". De même, ce n'est pas une réponse vrai/faux, cependant elle est un petit plus intéressante car il n'y a pas une seule réponse mais deux (Sam and Denise). Ceci nous montre qu'on ne peut pas s'arrêter d'examiner les faits dès qu'une réponse est trouvée: on doit continuer les recherches tant que tous les faits intéressants n'ont pas été regardés. Quand il y a de multiples réponses à une question, la personne qui demande peut être intéressée par une réponse, plusieurs ou encore la totalité des réponses.

Une question à laquelle on ne peut pas répondre en regardant juste les faits est "Est ce que Mary est un parent de Sam ?". On ne peut pas y répondre car nous n'avons pas encore défini ce que signifie être *parent*. Il n'y a aucun fait direct portant sur un quelconque type de relation parent/enfant. Pour résoudre ce problème, nous pouvons ajouter un fait de la nature "X est un parent Y" pour chaque fait du type "Y est un enfant de X" présent au dessus. Cette approche est encombrante et sujette aux erreurs en particulier quand la base de faits est grande. Une meilleure approche est d'utiliser des *règles* pour inférer ces nouveaux faits.

II-2 - les règles

Les règles sont des déclarations utilisées pour inférer des nouveaux faits (ou donnés) à partir d'un ensemble de faits existants. Voici quelques règles simples pour décrire le liens de parenté dans la famille:

- 1) X est un parent de Y si: Y est un enfant de X
- 2) X est le père de Y si: X est un homme **ET** Y est un enfant de X
- 3) X est la mère de Y si: X est une femme **ET** X est un parent de Y

Ici, X et Y sont des paramètres et non des constantes comme "Sam" ou "Franck". La règle *parent* fournit la possibilité de répondre à une question du type: "Est ce que Mary est un parent de Sam ?" ou "Qui est un parent de Sam ?". Il faut noter que les règles *parent* et *père* sont uniquement définies en terme de faits. De l'autre côté, la règle *mère* est spécifiée en utilisant un mélange de faits et de règles bien qu'elle aurait pu être spécifiée d'une manière similaire à la règle *père*

II-2-A - Les règles récursives

Les règles peuvent aussi être spécifiée de manière récursives. Considérons une règle qui définit la notion d'ancêtre:

X est un ancêtre de Y si:

X est un parent de Y **OU**

Z est un parent de Y **ET** X est un ancêtre de Z.

Maintenant, nous pouvons répondre à la question "Est ce que Garry est un ancêtre de Sam ?" qui devrait répondre "vrai". De façon similaire, si nous demandons "Qui est un ancêtre de Sam ?", on devrait obtenir Franck, Mary et Garry.

II-3 - Requêtes et assertions

Maintenant que nous disposons de la base de connaissance, nous sommes prêts à poser des questions. Les problèmes spécifiques à résoudre le sont en posant des questions. Nous avons vu précédemment des requêtes à propos des liens familiaux. D'autres exemples de requêtes peuvent être par exemple lorsque qu'on manipule un graphe "Quel est le chemin le plus court entre deux noeuds A et B ?" ou encore "Est ce que le graphe G est un graphe cyclique ?"

Les requêtes peuvent se classer en deux catégories. La première est un simple test pour déterminer si une affirmation est vraie: "Est ce que Sam est un enfant de Garry ?". Ce sont des assertions étant donné que le travail principal est de vérifier si la phrase est vraie ou non. Le second type de question amène une ou plusieurs réponses.

Par exemple: "Qui est l'enfant de Franck ?", nous ne vérifions pas si un fait est vrai mais nous demandons au système de déterminer les enfants de Franck. Dorénavant, nous parlerons de ces requêtes en terme de requêtes génératives car elles demandent de générer des solutions.

II-4 - Calcul par inférence

Jusque là, nous n'avons pas été très précis sur la méthode qui permet de trouver (on utilise aussi inférer) les réponses. Étant donné les faits et les règles décrits avant, il est facile pour n'importe quel individu d'inférer mentalement les réponses aux questions qu'on a posées. Le principe fondamental qui se cache derrière est nommé en logique formelle *modus ponens*

Si A est vrai et que A implique B alors B est vrai (A et B étant des énoncés arbitraires)

Ce principe est le coeur du modèle de calcul utilisé en programmation logique. Une version intuitive mais plutôt rude de l'algorithme utilisé pour répondre à des requêtes dans un système tel que Prolog est le suivant:

- Construire une liste de faits et de règles pertinentes
- Prendre un fait pertinent et regarder s'il répond à la question. Recommencez tant que la liste des faits pertinents n'est pas vide
- Prendre une règle pertinente qui peut être appliquée pour dériver de nouveaux faits (en utilisant le *modus ponens*). Recommencez jusqu'à temps que la liste des faits et des règles pertinentes soit épuisée.

A chacune des étapes de l'algorithme d'inférence, on peut choisir plus d'un fait ou d'une règle pour continuer l'exécution. Chaque choix mène à un chemin d'inférence différent et tous les chemins ne mènent pas forcément à une solution. Les chemins qui ne mènent pas une solution sont abandonnés et le processus d'inférence reprend depuis le point le plus récent où des faits et/ou règles étaient disponibles pour l'inférence. Abandonner le chemin courant et reprendre l'exécution depuis un endroit précédemment exploré afin de faire un choix différent est appelé *backtracking*. Le *backtracking* continue tant que tous les chemins d'inférence n'ont pas été examinés. Ainsi, le calcul est réduit à parcourir des chemins d'inférence déterminés par les faits et les règles disponibles.

Ceci est similaire à effectuer une recherche profonde dans un arbre binaire où les feuilles représentent les résultats finaux possibles et les noeuds internes les résultats intermédiaires. Dans une logique formelle pure, l'ordre de sélection des faits et des règles est non déterminé. Ceci implique que l'ordre dans lequel on va obtenir les réponses est non-déterminé aussi. Étant donné que certains chemins peuvent mener plus rapidement à une solution que d'autres, en pratique l'ordre d'exécution est fixé (le même que l'ordre de déclaration) pour permettre un contrôle sur l'efficacité (i.e. pour améliorer l'efficacité de la recherche). Fixer l'ordre d'application des règles simplifie aussi le raisonnement à propos de l'exécution des programmes logiques ce qui est aussi un point important pour le débogage.


II-5 - Résumé

Dans cette section nous avons décrit les faits, règles, requêtes, assertions et l'inférence. Les faits sont simplement des énoncés. Les règles peuvent être utilisées pour dériver de nouveaux faits. Elles peuvent être construites à partir de faits, d'autres règles ou elles-mêmes (exemple des règles récursives). Un ensemble de règles et de faits peut être utilisé pour répondre à des questions pertinentes. Les questions peuvent de façon générale être classées en d'un côté celles qui demandent une réponse oui/non et de l'autre celles pour lesquelles on doit générer la réponse. Les premières questions sont des assertions et les secondes des requêtes génératives. Les assertions ne peuvent avoir qu'une seule réponse (vrai ou faux). Les requêtes génératives peuvent avoir zéro ou plusieurs solutions. "Est ce que Sam est un homme ?" est une assertion. "Qui est un enfant de Mary ?" est une requête générative. L'inférence logique est utilisée pour répondre aux questions. Elle implique un examen des faits et l'application des règles. De nouveaux faits émergent depuis une application des règles qui deviennent à leur tour candidat pour un examen futur durant le processus d'inférence.

III - Programmation logique en C++

Maintenant, traduisons les faits et règles présentés ci-dessus en C++ de telle sorte qu'on puisse les exécuter. Les exemples sont codés avec Castor, une bibliothèque *Open-Source* qui permet l'utilisation du paradigme logique de façon naturelle en C++ en permettant aux faits et aux règles d'être déclarés en tant que classes, fonctions ou juste expressions. Ce bas niveau d'intégration est très utile et permet un environnement de programmation où les frontières du paradigme perdent leur sens.

Les fonctions C++ suivantes représentent les faits *enfant*, *genre* et la règle *père* (décrite précédemment) en utilisant Castor.

 *Tout le code relatif à castor se trouve dans le namespace castor Pour compiler le code, il faut donc soit faire un using namespace castor ou préfixer tous les types avec castor::*

```
#include "castor.h"
//ou #include <castor.h>, dépend de votre installation

//c est un enfant de p
relation enfant(lref<std::string> c, lref<std::string> p)
{
    return eq(c,"Sam") && eq (p,"Mary") //fait 1
    || eq(c,"Denise") && eq (p,"Mary") //fait 2
    || eq(c,"Sam") && eq (p,"Franck") //fait 3
    || eq(c,"Denise") && eq (p,"Franck") //fait 4
    || eq(c,"Franck") && eq (p,"Gary") //fait 5
    ;
}

//le genre de p est enregistré dans g
relation genre(lref<std::string> p, lref<std::string> g)
{
    return eq(p,"Franck") && eq (g,"homme") //fait 1, liste 2
    || eq(p,"Sam") && eq (g,"homme") //fait 2, liste 2
    || eq(p,"Mary") && eq (g,"femme") //fait 3, liste 2
    || eq(p,"Denise") && eq (g,"femme") //fait 4, liste 2
    ;
}

//f est le pere de c
relation pere(lref<std::string> f, lref<std::string> c)
{
    //si f est un homme et c un enfant de f
    return genre(f,"homme") && enfant (c,f); //
}
```

Les faits et les règles sont tout deux déclarés en tant que fonctions retournant un objet de type *relation*. Les paramètres sont des instances de la classe générique *lref* qui signifie référence logique (ou *logic reference* en anglais). Ici, la classe fournit un moyen similaire aux références pour passer un objet en paramètre. A l'inverse des références classiques en C++, les références logiques peuvent rester non initialisées. La valeur pointée par une référence logique peut être obtenue en la déréférençant avec l'opérateur ***.

La fonction *eq* est appelée la relation d'unification. Son travail est de *tenter* de faire correspondre ses deux arguments. Si l'un de ses deux arguments est une référence logique non initialisée, elle va lui assigner la valeur de l'autre argument. Si les deux arguments ont des valeurs bien définies, alors elle va juste comparer les valeurs. Cette tâche est appelée **unification**. Considérons un appel à *eq(c,"Sam")* dans la relation *enfant*. Si *c* a été précédemment initialisé, alors *eq* va juste comparer la valeur de *c* avec "Sam". Cependant, si *c* n'a pas été initialisé, on assignera "Sam" à *c*. Le type *relation*, les références logiques et la relation *eq* sont étudiés plus loin dans les sections 3.1, 3.2 et 3.3 respectivement.

Aucune des relations (que ce soit *eq* ou une relation définie par l'utilisateur comme *enfant* ou *genre*) ne retourne le calcul voulu au moment où elles sont appelées. A la place, elles retournent un objet-fonction qui encapsule le calcul souhaité. L'objet-fonction peut être sauvegardé dans un objet de type *relation* et l'évaluation du calcul sera déclenché par l'appel de l'opérateur *()* sur l'objet encapsulant. Avec ces définitions, nous sommes en mesure de faire certaines assertions et certaines requêtes. Le code suivant est une simple assertion pour vérifier si Sam est un homme:

```
relation samIsMale = genre("Sam", "homme");
if (samIsMale())
    std::cout<<"Sam est un homme";
else
    std::cout<<"Sam n'est pas un homme";
```

De façon similaire, on peut vérifier si Franck est bien le père de Sam.

```
relation samsDadFranck = pere("Franck", "Sam");
if (samsDadFranck())
    std::cout<<"Franck est le père de Sam";
else
    std::cout<<"Franck n'est pas le père de Sam";
```

Nous pouvons aussi faire des requêtes génératives comme "Quel est le genre de Sam ?"

```
lref<std::string> g;
relation samsGender = genre("Sam", g);
if (samsGender())
    std::cout<<"Le genre de Sam est: "<<*g;
else
    std::cout<<"Le genre de Sam est inconnu";
```

Ici nous passons "Sam" en tant que premier argument et nous laissons le second indéfini (c'est-à-dire sans lui avoir attribué une valeur). Notez que la même fonction *genre* est utilisée pour vérifier si Sam est un homme et trouver le genre de Sam.

Quand tous les arguments ont été initialisés (c'est à dire qu'ils disposent d'une valeur), le calcul effectue une vérification pour voir s'ils satisfont la relation portant sur le genre. Les arguments qui ont été laissés non initialisés vont agir en tant que paramètres de sortie et être initialisés avec la valeur qui satisfait la relation. Cette nature bidirectionnelle de *lref* en tant que paramètre est essentielle dans le modèle de programmation logique. Il est toutefois important de noter que les références logiques ne sont généralement pas simultanément à la fois des entrées et des sorties de la fonction comme c'est souvent le cas avec le passage par référence pour les fonctions comme `std::swap`. Que se passe-t-il si on laisse les deux arguments de la relation *genre* non initialisés ?

```
lref<std::string> p, g;
relation anyPersonsGender = genre(p, g);
if (anyPersonsGender())
    std::cout<<"Le genre de "<<*p<<" est: "<<*g;
```

Dans ce cas, Des valeurs vont être assignées à *p* et *g* par la relation *genre*. Étant donné que la première paire déclarée dans la relation est ("Franck", "homme"), *p* sera assigné à *Franck* et *g* à *homme*.

Les requêtes génératives tel que *samsGender* et *anyPersonsGender* présentées au dessus peuvent avoir zéro, une ou plusieurs solutions. Jusqu'à maintenant, nous n'avons généré qu'une solution, la seule possible. Itérer à travers les solutions utilise assez naturellement l'instruction *while* à la place de l'instruction *if* que nous avons utilisée jusque là. L'exemple précédent peut être ré-écrit de la façon suivante afin de décrire toutes les personnes et leur genre.

```
while (anyPersonsGender())
    std::cout<<"Le genre de "<<*p<<" est: "<<*g<<std::endl;
```

De façon similaire, on peut lister tous les enfants de Franck:

```
lref<std::string> c;
int count=0;
relation enfantsdeFranck = pere("Franck", c);
while (enfantsdeFranck())
{
```

```
++count;
std::cout<<*c<<" est un enfant de Franck"<<std::endl;
}
std::cout<<"Franck a "<<count<<" enfants";
```

Une fois que toutes les solutions ont été parcourues, l'appel de *enfantsdeFranck* renvoie *faux* ce qui provoque l'arrêt de la boucle. Quand toutes les solutions ont été parcourues, la référence logique *c* est automatiquement restaurée à son état original de non initialisation.

Il est important de noter comment l'utilisation d'instructions fondamentalement impératives comme *if* ou *while* rendent la transition entre la programmation logique (où les résultats sont générés) et la programmation impérative (où les résultats sont consommés) simple et indolore.

Les fonctions *eq*, le type template *lref* et le type *relation* avec les surcharges des opérateurs *&&* et *||* fournissent les bases de la programmation logique en C++. Ils sont décrits brièvement dans les sections suivantes. Pour une analyse plus profonde, on peut se reporter au Document: CastorDesign présent sur le site de Castor.

III-1 - Le type relation

En programmation logique, il est commun de nommer les faits et les règles en tant que **prédicats** et **relations**. Le terme relation trouve son origine dans la théorie des ensembles où il implique une associations entre des ensembles. Ainsi, *genre* est une relation binaire entre un ensemble d'individu et un ensemble de genres. Généralement, une distinction stricte entre règles et faits n'est pas nécessaire lorsqu'on programme avec Castor étant donné qu'ils peuvent être mélangés librement avec la même fonction/expression.

En cohérence avec la programmation logique, nous désignerons dorénavant les fonctions qui représentent les faits ou les règles comme relations. Ainsi, les fonctions retournant un type relation sont aussi appelées relations.

En interne, le type relation représente une fonction ou un foncteur ne prenant aucun argument et qui retourne un booléen. Ainsi, *enfant*, *genre* et *pere* retournent un foncteur qui peut être évalué de manière paresseuse.

III-2 - lref: La référence logique

Le type template *lref* est une abréviation pour référence logique (*logic reference* en anglais) et fournit un moyen simple pour faire entrer ou sortir des valeurs d'une relation sous forme d'arguments, de manière similaire aux références du C++. A l'inverse des références du C++, une référence logique ne doit pas forcément être initialisée et fournit la fonction membre *defined* afin de vérifier si elle a été initialisée. L'opérateur de déréférencement *** et l'opérateur flèche *->* peuvent être appliqués à une référence afin d'obtenir la valeur de la référence logique.

Maintenant, regardons la sémantique d'une référence logique lors de son initialisation. Quand une référence logique est initialisée (comprendre construite) ou assignée avec une valeur de type T (ou un type convertible en T), la référence stocke en interne une copie de la valeur. Quand une référence est initialisée avec une autre référence (via le constructeur de copie), on dit que les deux références sont *liées ensembles*. Des références liées ensembles pointent sur la même valeur sous-jacente (s'il y en a une). Ainsi, quand il y a le moindre changement à la valeur pointée par l'une des références, ce changement est observé par toutes les références liées ensemble. Une liaison entre des références logiques ne peut être brisée. Cela signifie que si A et B sont des références logiques liées ensembles et que C et D le sont aussi alors on ne peut pas casser la liaison de C vers D pour ensuite la lier à A et B. C continuera à faire partie de la liaison pour toute sa durée de vie. Les références logiques peuvent seulement être liées par initialisation (construction par copie) mais pas par affectation. Une liaison peut seulement être formée durant la construction de la référence et sera cassée lorsque la référence sera détruite. Quand la dernière référence logique dans une liaison est détruite, la valeur sous-jacente est désallouée.

Ndt: Depuis Castor 1.1, des pointeurs peuvent être utilisés afin d'initialiser une *lref*. Quand on utilise des pointeurs, on doit spécifier si la *lref* doit gérer ou non la durée de vie de l'objet référencé par le pointeur. Par exemple:

```
//La durée de vie de "Roshan" sera gérée.
lref<std::string> s(new std::string("Roshan"), true);

//La durée de vie de name ne sera pas gérée.
string name="Naik";
lref<std::string> s2(&name, false);
```

L'affectation de pointeur se fait avec la fonction membre *set_ptr*

```
std::string str="Castor";
s.set_ptr(&str, false); // désalloue la chaîne "Roshan" précédemment allouée et ne gère pas la durée de vie de st
```

Alors qu'utiliser des pointeurs est très utile pour assigner des objets aux lrefs (spécialement quand on mélange plusieurs paradigmes) et efficace, cela peut aussi être très dangereux si on ne les utilise pas avec attention. On doit faire attention à bien spécifier la politique de gestion de durée de vie. Par exemple, si on pointe sur un objet qui est alloué sur la pile, la lref ne doit pas gérer la vie du pointeur, sous peine d'erreur. De façon similaire, si deux lref indépendantes se réfèrent au même objet en utilisant l'initialisation/affectation d'un pointeur, aucune des deux ne doit gérer la durée de vie de l'objet. C'est au programmeur de s'assurer que les objets utilisés par les pointeurs des lrefs sont toujours valides tant qu'elles peuvent tenter d'y accéder. Spécifier accidentellement la politique de gestion de vie à *faux* à la place de *vrai* provoquera une fuite de mémoire.

III-3 - La relation eq: La fonction d'unification

La fonction *eq* est la fonction d'unification et prend deux arguments. Les arguments peuvent être des références logiques ou des valeur classiques. *eq* retourne une expression (comprendre un foncteur) qui ,lorsqu'il est évalué, tente d'unifier les deux arguments. Si l'unification réussie, la fonction renvoie *true*, *false* sinon. L'unification effectuée par *eq* est définie comme il suit:

- Si les deux arguments sont initialisés, leurs valeurs sont comparés et le résultat de la comparaison retourné
- Si seulement un argument est initialisé, l'argument non initialisé se verra modifié et prendra la valeur de l'autre argument
- Si aucun des deux arguments n'est initialisé, une exception est lancée.

Ainsi, l'unification va *générer* une valeur pour l'argument non initialisé pour les rendre égaux ou elle va comparer ses arguments s'ils sont tous les deux initialisés. Il est possible d'implémenter d'autres variations de l'unification mais ce n'est en général pas nécessaire.

Les expressions retournées par les divers appels à *eq* à l'intérieur de la relation *genre* par exemple sont rassemblées pour former une composante d'expression plus grosse à l'aide des opérateurs *&&* et *||*. Il est important de noter que le composant de l'expression est retourné sans être évalué. Ces expressions sont évaluées de manière paresseuse. En d'autres mots, ces expressions sont sauvegardées dans un objet de type relation et seront sujettes à une future évaluation quand il sera nécessaire de la faire. La section suivante examine l'évaluation de ces expressions en détail.

III-4 - Évaluation des requêtes

Étant données les relations au dessus, on peut formuler la requête qui vérifie "Est ce que Sam est une homme ?" et enregistre le résultat dans variable *samIsMal* tel qu'il suit:

```
relation samIsMale = genre("Sam", "homme");
```

L'appel *genre("Sam", "homme");* ne réalise en fait aucun vrai calcul, il retourne tout simplement un foncteur qui peut être évalué plus tard afin de vérifier si Sam est un homme. L'appel à une relation ne l'évalue pas. Cette découpe entre appel et évaluation est appelée évaluation paresseuse (*lazy evaluation* en Anglais). Pour exécuter l'évaluation de l'expression, on applique tout simplement l'opérateur *()* sur la variable qui contient l'expression.

```
if (samIsMale())
    std::cout<<"Le genre de Sam est: "<<*g;
else
    std::cout<<"Le genre de Sam est inconnu";
```

La relation *genre* a été définie précédemment comme il suit:

```
//le genre de p est enregistré dans g
```

```
relation genre(lref<std::string> p, lref<std::string> g)
{
    return eq(p,"Franck") && eq (g,"homme")
    || eq(p,"Sam") && eq (g,"homme")
    || eq(p,"Mary") && eq (g,"femme")
    || eq(p,"Denise") && eq (g,"femme")
    ;
}
```

Après que "Sam" et "homme" aient été passés en tant qu'arguments à *genre*, l'expression retournée ressemble à ce qui suit avec les arguments substitués:

```
eq("Sam","Franck") && eq ("homme","homme")
|| eq("Sam","Sam") && eq ("homme","homme")
|| eq("Sam","Mary") && eq ("homme","femme")
|| eq("Sam","Denise") && eq ("homme","femme")
```

Cette expression contient quatre expressions && joints par trois opérateurs ||. Si une seule des expressions && retourne vrai, toute l'expression a été résolue et l'évaluation s'arrête afin de signifier la réussite. Regardons pas à pas l'exécution qui se produit lorsque l'opérateur () est appliqué à *samIsMale*. La première expression *eq("Sam","Franck") && eq ("homme","homme")* est choisie pour être évaluée. On tente d'unifier "Sam" avec "Franck" lorsque on évalue *eq("Sam","Franck")* ce qui échoue étant donné que "Sam" n'est pas égal à "Franck". Les règles de la logique nous permettent de dire que le reste cette expression && n'a pas besoin d'être évalué. Ainsi, ce chemin d'évaluation est immédiatement abandonné le backtracking reprend l'exécution à la prochaine expression && qui est *eq("Sam","Sam") && eq ("homme","homme")*. Cette fois, chaque moitié de l'expression unifie avec succès leurs arguments car ils sont égaux. Une solution a été trouvée et on renvoie vrai à l'appelant qui se trouve être dans l'instruction if.

Si on refait un appel de l'opérateur () sur la variable *samIsMale*, l'exécution reprend où elle s'était arrêtée précédemment. Dans ce cas, il s'agit de la troisième expression && qui échoue sur l'unification de "Sam" et "Mary". Ceci mène à l'évaluation de la quatrième expression && qui échoue aussi pour des raisons similaires et faux sera retourné à l'appelant. Toutes les expressions && ont maintenant été évaluées et appliquer l'opérateur () à *samIsMale* retournera faux immédiatement et ceci de manière immédiate.

Maintenant, considérons comment spécifier une requête générative tel que "Quel est le genre de Sam ?". Une telle question est construite en appelant *genre* avec le premier argument initialisé à "Sam" et en laissant le second non initialisé.

```
lref<std::string> g;
relation samsGender = genre("Sam",g);
if (samsGender())
    std::cout<<"Le genre de Sam est: "<<*g;
```

Notons comment il est possible d'utiliser la même relation *genre* pour à la fois vérifier si quelqu'un est un homme ou une femme et de trouver le genre d'une personne. Quand une solution sera trouvée, *g* sera initialisé à "homme". L'expression retournée par *genre("Sam",g)* ressemble à cela après substitution des arguments.

```
eq("Sam","Franck") && eq (g,"homme")
|| eq("Sam","Sam") && eq (g,"homme")
|| eq("Sam","Mary") && eq (g,"femme")
|| eq("Sam","Denise") && eq (g,"femme")
```

Quand l'opérateur est appliqué à *samsGender* pour la première fois, l'évaluation de la première expression && échoue car on ne peut unifier "Sam" et "Franck". Le processus de backtracking arrive à la seconde expression *eq("Sam","Sam") && eq (g,"homme")*. L'unification de "Sam" avec "Sam" réussit et ensuite *eq(g,"homme")* est évalué. Étant donné que *g* est non défini, *eq* va assigner la valeur "homme" à *g* et ainsi l'unification réussit. L'évaluation de l'expression entière s'arrête et renvoie *vrai* au *if*. Si l'opérateur() est à nouveau appliqué sur *samsGender*, l'exécution reprend au point où elle s'était arrêtée. Cependant une chose très intéressante se produit avant que l'exécution ne reprenne. Il est critique pour la poursuite de l'évaluation que les différents effets de bord qui ont lieu dans la branche abandonnée soient annulés avant de reprendre l'exécution. En effet, si ces effets de bords ne sont pas annulés avant la reprise de l'exécution, ils peuvent affecter les futures évaluations et mener à des résultats faux.

Ainsi, l'unification de *g* avec *homme* doit être annulé restaurant ainsi *g* à son état original de non initialisation. Cette fonction de désunification est automatiquement fournie par la fonction d'unification *eq*.

La requête ci-dessus n'a qu'une seule solution mais nous avons observé précédemment que des requêtes peuvent avoir plusieurs solutions. Par exemple, on peut vouloir trouver tous les hommes du système. Une fois de plus, on va utiliser la relation *genre* mais on va laisser le premier argument non initialisé et le second argument à *homme*.

```
lref<std::string> person;
relation males = genre(person, "homme");
while(males())
    std::cout<<*person<< " est un homme"<<std::endl;
```

Cette fois, on appelle *males()* jusqu'à temps qu'elle retourne *faux*. Chaque appel à *males* déclenche la recherche de la prochaine solution, c'est-à-dire une valeur acceptable pour la référence *person*. A chaque tour de boucle, on assigne à *person* une valeur différente qui représente une solution. Quand toutes les solutions ont été trouvées, *males* retourne *faux*, la boucle s'arrête et le processus de backtracking restaure *person* à son état original de non initialisation. Etant donné que *person* ne se réfère à plus rien après que la boucle soit finie, tenter d'accéder à sa valeur par le biais de l'opérateur *** ou de l'opérateur *->* lancera une exception.

III-5 - Les règles récursives

La récursion est souvent quelque chose d'essentiel quand on déclare des règles dans le paradigme logique. Dans notre exemple des relations familiales, considérons la définition d'une règle portant sur la relation "ancêtre". Notez qu'il peut y avoir un nombre quelconque de niveaux parent-enfant entre l'ancêtre et le descendant. On peut définir cette règle de façon récursive:

A est un ancêtre de D si:

D est un enfant de A OU

D est un enfant de P ET A un ancêtre de P.

Le code suivant est la traduction naïve de la règle ci-dessus.

```
//Version défectueuse
relation ancetre(lref<std::string> A, lref<std::string> D)
{
    lref<std::string> P;
    return enfant(D,A) || enfant(D,P) && ancetre (A,P);
}
```

Cependant, cette définition contient une récursion infinie. L'instruction *return* contient un appel récursif à *ancetre*. Afin de briser la récursion, on a besoin de repousser l'appel récursif de telle sorte qu'il n'ait lieu que seulement lorsqu'on en a vraiment besoin. Castor fournit une relation aidant à cela avec *recurse* qui sert à définir des règles récursives. *recurse* prend en premier paramètre la relation qui doit subir un appel récursif et ensuite les paramètres de cette relation qui sont nécessaires à son appel. La relation retourne un foncteur qui lorsqu'il est évalué mène à l'actuel appel récursif d'*ancetre*. Le problème d'appel *ancetre(A,P)* est tout simplement ré-écrit en *recurse(ancetre,A,P)*:

```
relation ancetre(lref<std::string> A, lref<std::string> D)
{
    lref<std::string> P;
    return enfant(D,A) || enfant(D,P) && recurse(ancetre,A,P);
}
```

Dans l'exemple au dessus, la récursion a lieu sur la relation *ancetre* qui est définie en tant que fonction libre. L'usage de *recurse* pour des fonctions membres statiques est exactement le même. Pour utiliser *recurse* sur des fonctions membres, il faut passer en plus le pointeur *this* comme il suit:

```
recurse(this, &Type::fonction, ...arguments...);
```

III-6 - Relations dynamiques

Les définitions de toutes les relations décrites jusqu'ici ont été fixées à la compilation. La relation *genre* par exemple fournit une liste définie de paires nom/genre qui ne change pas au moment de l'exécution du programme. Mais quand les informations ne sont disponibles que dynamiquement (depuis un fichier ou une base de données), nous avons besoin d'un mécanisme alternatif pour construire les clauses de notre relation. Ici, nous avons la relation *genre* de la partie II qui définit de façon statique les informations.

```
// relation definie statiquement
relation genre(lref<string> p, lref<string> g)
{
  return eq(p,"Frank") && eq(g,"male")
  || eq(p,"Sam") && eq(g,"male")
  || eq(p,"Mary") && eq(g,"female")
  || eq(p,"Denise") && eq(g,"female");
}
```

Supposons que les informations sur le genre d'une personne ait été lues depuis un fichier ou une base de données et qu'elle se trouve une `std::list<pair<string,string> >` appelée *genderList*. Le premier champ de la paire contient le nom de la personne et le second son genre. Maintenant, nous pouvons définir une relation *gender_dyn* sur *genderList* tel qu'il suit:

```
list<pair<string,string> > genderList = ...;

// dynamically building a relation
Disjunctions genre_dyn(lref<string> p
                      , lref<string> g)
{
  Disjunctions result;
  list<pair<string,string> >::iterator i;
  for( i=genderList.begin();
       i!=genderList.end(); ++i)
    result.push_back(
      eq(p,i->first) && eq(g,i->second) );
  return result;
}
```

Ici, nous utilisons le type *Disjunctions* afin de construire dynamiquement un ensemble de clause OU pour la relation. Le type *Disjunctions* est elle-même une relation qui supporte l'addition dynamique de clauses. Ainsi, on peut déclencher l'évaluation en utilisant l'opérateur `()`. Le type de retour de *genre_dyn* a été changé de relation à *Disjunctions*. Ceci est optionnel, mais utile car il porte implicitement le caractère dynamique de la relation *genre_dyn* aux utilisateurs. Conceptuellement, le type *Disjunctions* est simplement une collection de relations. Quand une *Disjunctions* est évaluée, les relations sont traitées comme s'il existait un opérateur `||` entre chaque paire de relations. Des relations peuvent être ajoutées au début ou à la fin de la *Disjunctions* en utilisant les fonctions membres *push_front* et *push_back*. Durant l'évaluation, les relations contenues sont évaluées du début vers la fin. Notons comment une expression entière (`eq(p,i->first) && eq(g,i->second)`) est ajoutée à la *Disjunctions*. Les relations *Conjunctions* et *ExDisjunctions* sont aussi fournies afin de construire des relations dynamiques. Elles correspondent respectivement aux opérateurs `&&` et `^`. La relation *genre* est redéfinie ci-dessous en utilisant *Conjunctions* et *Disjunctions*.

```
Disjunctions gender_dyn(lref<string> p
                      , lref<string> g) {
  Disjunctions result;
  Conjunctions conj1 = eq(p,"Frank");
  conj1.push_back( eq(g,"male") );

  Conjunctions conj2 = eq(p,"Sam");
  conj2.push_back( eq(g,"male") );

  Conjunctions conj3 = eq(p,"Mary");
```

```

conj3.push_back( eq(g, "female" ) );

Conjunctions conj4 = eq(p, "Denise");
conj4.push_back( eq(g, "female" ) );

result.push_back( conj1 );
result.push_back( conj2 );
result.push_back( conj3 );
result.push_back( conj4 );

return result;
}

```

En résumé, *Conjunctions*, *Disjunctions* et *ExDisjunctions* offrent des moyens de construire des relations dynamiquement. Cette capacité fait d'elles un moyen de faciliter la métaprogrammation à l'exécution dans le paradigme logique [NDT: du code qui se génère lui même mais à l'exécution].

III-7 - Inline Logic Reference Expressions (ILE)

On rencontre souvent des cas où les relations sont de simples opérations faites sur les références logiques avec une application des opérateurs standards comme +, -, etc. Par exemple, supposons l'existence d'une relation *multiply* qui calcule le produit de deux nombres. On peut obtenir le cube d'un nombre de la façon suivante:

```

lref<int> n=2, sq, cu;
multiply(sq,n,n) && multiply(cu,sq,n) ( );
cout << *cu ;

```

En utilisant *multiply*, le carré du nombre est généré et en réutilisant ce carré, on obtient le cube. A quoi ressemblerait le code ci-dessus pour une expression du type $n^5 - n/2 + 5$? Il est évident qu'écrire simplement les expressions arithmétiques comprenant plus d'un couple d'opérateurs devient verbeux et vite illisible. Castor autorise la spécification d'expressions logiques de manière inline. L'exemple du cube au dessus peut être ré-écrit de la manière suivante:

```

lref<int> cu, n =2;
eq_f(cu, n*n*n) ();
cout << *cu;

```

La relation *eq_f* unifie son premier argument avec le résultat de l'évaluation de son second argument qui est un foncteur. Il est important de noter que la valeur de $n*n*n$ n'est pas calculée lorsqu'elle est passée en argument à *eq_f*. Étant donné que l'expression est composée en utilisant une *lref*, elle est transformée en un foncteur qui est passé à *eq_f*. La fonction objet suit l'évaluation quand l'évaluation de *eq_f* commence. Au moment de l'évaluation, *n* doit être initialisé car l'accès à une référence logique non initialisée se solde par une exception. Des expressions arbitraires tel que $n^5 - n/2 + 5$, composées de *lref* et des opérateurs communs surchargés peuvent être utilisées afin de construire un foncteur facilement:

```

eq_f(cu, n*5-n/2+5) ();

```

De telles spécifications d'expression inline impliquant des références logiques sont appelées ILE pour "Inline Logic reference Expression" Les ILEs sont très utiles dans un grand nombre de situations. Afficher à la console avec la relation *write_f* est un autre exemple où elles peuvent être utilisées. Le code suivant montre comment utiliser des ILEs afin d'afficher deux chaînes de caractères séparées par une virgule:

```

lref<std::string> ls="Hello"; std::string s="world";
write_f(ls + std::string(",") + s) ();

```

La relation `write_f` prend une fonction ou un foncteur en tant qu'argument et affiche le résultat de l'évaluation sur `stdout`. On utilise une ILE dans l'exemple ci-dessus pour instancier un foncteur et le passer à `write_f`. Les ILEs peuvent être utilisées en tant qu'argument pour n'importe quelle relation fournie par Castor est suffixée avec `_f`.

Si `T` est le résultat d'une ILE, alors tous les opérateurs définis sur `T` comme l'opérateur virgule, l'opérateur de déréférencement* l'opérateur `&` et `->` peuvent être utilisés sur une `lref<T>` pour produire une ILE. Les ILE en l'état actuel des choses manquent d'un support pour le mélange avec un type `T2` même si les opérateurs entre `T` et `T2` sont définis. Ainsi

```
write_f(ls + ",")(); // erreur à la compilation
```

va échouer à la compilation même si `operator+` est défini entre `std::string` et `const char*`.

Jusqu'ici, nous avons utilisé les ILEs pour passer facilement des expressions à d'autres relations. Les ILEs produisent aussi des booléens qui peuvent être utilisés pour créer de simples relations à la volée. Considérons la relation suivante qui affiche le résultat de la comparaison de ses arguments:

```
relation greaterLessEq(lref<int> n
                      , lref<int> cmpVal) {
  return write(n) && write(" is ") &&
    ( predicate(n<cmpVal) && write("lesser")
    || predicate(n>cmpVal) && write("greater")
    || write("equal") );
}
```

Ici, les expressions `predicate(n>cmpVal)` et `predicate(n<cmpVal)` sont utilisées pour définir de façon pratique des relations de comparaison directement dans le code au lieu de passer par des relations globales tel que `less` ou `greater` qui feraient le même travail. De façon similaire, `predicate(n%2==0)` peut être utilisé afin de créer une relation dans le code qui teste si un nombre est pair ou on peut même concevoir des expressions plus complexes comme `predicate(n*2 >= cmpVal*n/2)`. Il est important de noter que les relations définies à l'aide d'ILEs ne sont pas capables de générer des solutions. Elles vont seulement vérifier si une condition est vraie ou fausse. Ainsi, `predicate(n>cmpVal)` ne va pas générer les valeurs possibles de `n` de telle sorte que la relation soit vraie. Toutes les références logiques utilisées dans une relation doivent être initialisées au moment où l'évaluation de l'ILE a lieu ce sans quoi une exception sera lancée. L'exception sera lancée directement par la référence logique non initialisée lorsque l'ILE tentera de la déréférencer pour l'évaluation. Une présentation plus détaillée des ILEs a lieu dans la section V.

III-8 - Conteneurs et séquences

Castor fournit des moyens pour travailler avec les conteneurs standards et les itérateurs dans une tendance fonctionnelle. Les tâches associées avec les conteneurs peuvent être regroupées de façon générale avec d'un côté le remplissage des conteneurs et de l'autre le parcours. Souvent, dans la programmation logique classique (de la même manière qu'en programmation fonctionnelle ou en métaprogrammation en C++), les opérations qui demandent une modification d'une séquence sont faites en créant une nouvelle séquence qui intègre les modifications. La suppression d'un élément se fait en recopiant tous les éléments sauf celui à supprimer. Ainsi, la suppression d'un élément est une combinaison entre parcours d'une séquence et création d'une nouvelle. L'ajout d'un nouvel élément peut aussi être fait de manière similaire. La modification d'un élément est considérée comme une opération sur l'élément et non la séquence elle-même.

Alors qu'il est possible d'écrire une relation où la suppression et l'ajout de nouveaux éléments se ferait directement sur la séquence originale, nous nous bornerons ici aux techniques classiques dans le style logique. Les sections suivantes décrivent les tâches courantes sur les séquences.

III-8-A - Génération des séquences

Les séquences sont créées en utilisant la relation `sequence`. Le code suivant montre comment créer une liste de trois éléments:

```
lref<list<int> > le;
relation evens = sequence(le) (2) (4) (6);
// see what it generates
if(evens())
    copy(le->begin(), le->end()
        , ostream_iterator<int>(cout, " "));
```

La référence logique *le* qui n'est pas initialisée est passée en premier argument à la relation *sequence*. Les éléments utilisés pour construire la séquence sont passés à la suite et individuellement à ce qui est retourné par l'appel précédent de la relation. Étant donné que *le* n'est pas initialisée à un objet de type *list<int>*, quand la relation *sequence* sera évaluée dans le *if*, *le* se verra assignée à une liste contenant les éléments 2, 4 et 6. Notez que la relation *sequence* déduit automatiquement à l'aide de son premier argument si vous voulez une *list<int>* ou un *vector<string>* ou encore une séquence d'autre autre type. Cette capacité est très utile quand on écrit du code générique.

Les capacités de la relation *sequence* vont plus loin. Des séquences peuvent être créées non pas juste avec des valeurs mais aussi à partir d'autres séquences, de références logiques ou un mélange arbitraire des deux. Dans le code suivant, on crée une liste avec un mélange des deux:

```
string s = "One";
lref<string> lrs = "Two";

vector<string> ls;
ls.push_back("Three"); ls.push_back("Four");

vector<string> lsTemp;
lsTemp.push_back("Five"); lsTemp.push_back("Six");

lref<vector<string> > lrls = lsTemp;

// on crée la séquence dans ln
lref<vector<string> > ln;
relation numbers =
    sequence(ln) ("Zero") (s) (lrs) (ls) (lrls);

//regardons ce qui a été produit.
if(numbers())
    copy(ln->begin(), ln->end(), ostream_iterator<string>(cout, " "));
```

Une des limitations actuelle quand on créer une séquence à partir d'une autre est que toutes les séquences utilisées doivent être du même type. Ainsi, *lref<list < int > >* ne peut pas directement être créée à partir d'un *vector < int >* ou d'une *lref<vector < int > >*. Cependant, on contourne facilement cette limitation avec le support des itérateurs qu'offre *sequence* tel que le montre le code suivant:

```
vector<int> vi;
vector<int>::iterator b1, e1;
lref<list<int> > lrl1;
// généré en utilisant les itérateur de vi
relation r = sequence(lrl1)(vi.begin(), vi.end());

lref<vector<int> > lrvi;
lref<list<int> > lrl12;
lref<vector<int>::iterator> b2, e2;

relation r = begin(b2, lrvi) && end(e2, lrvi)
    && sequence(lrl1) (b2,e2);
```

Les relations *begin* et *end* produisent des références logiques qui pointent respectivement sur le début et l'après fin de la séquence référencée par *lrvi*. Les itérateurs sont produits de manière fainéante, c'est-à-dire qu'ils existent seulement quand la relation est évaluée. Les références logiques produite par *begin* et *end* sont par la suite fournies à *sequence* pour construire *lrl1*. En général, on doit faire attention à éviter d'effectuer des évaluations strictes sur des références logiques puisqu'elles sont généralement initialisées avec des valeurs appropriées lorsque le backtracking et l'unification entrent en jeu.

III-8-2 - Itération dans une séquence

Les relations *head*, *next* et *prev* sont fournies afin d'itérer à travers une séquence. Les relations *head* et *tail* fournissent un moyen d'itération qui est très similaire à celui employé dans les langages fonctionnels. Les relations *next* et *prev* fournissent un support pour itérer avec des itérateurs ce qui est plus proche des techniques classiques d'itération en C++.

III-8-2-A - Itération avec head et tail

Avec cette technique, pour itérer à travers tous les éléments d'une séquence (comme une liste ou vecteur), on divise la séquence : d'un côté la tête (le premier élément, *head* en anglais), de l'autre la queue (le reste des éléments, *tail* en anglais). Diviser la queue en tête et queue permet d'avoir dans la nouvelle tête le second élément de la séquence originale. Dans cette voie, on peut continuer à diviser récursivement tant que la queue n'est pas vide. L'exemple suivant montre comment on peut utiliser les relations *head* et *tail* pour afficher tous les éléments d'une liste d'entier.

```
relation printList(lref<list<int> > li) {
  lref<int> h;
  lref<list<int> > t;
  return head(li, h) && write(h) && write(",")
    && tail(li, t)
    && recurse(printList,t);
}
list<int> li;
// .. on ajoute des nombres à la liste.
printList(li); //on affiche les éléments
```

La relation *head_tail* est disponible pour obtenir facilement la tête et la queue en un seul appel. L'instruction *return* au dessus peut être ré-écrite avec *head_tail* de la manière suivante:

```
return head_tail(li,h,t)
  && write(h) && write(",")
  && recurse(printList,t);
```

Dans les cas où la queue est utilisée seulement si une certaine condition est vraie, les relations *head* et *tail* peuvent être utilisées de façon plus efficace en calculant la queue après évaluation de la condition. Étant donné que le calcul de la queue est une opération en $O(n)$, cela paraît logique de retarder le calcul de la queue après détermination de son utilité. Un exemple typique est lorsqu'on cherche une valeur dans une liste. Une fois que la valeur a été trouvée, il n'est plus nécessaire de calculer le reste de la liste. Dans des cas comme *printList*, où il est simple de voir qu'on va toujours demander le calcul de la queue, *head_tail* devrait être préféré pour la brièveté.

Similaire à *head* et *tail*, il existe aussi *head_n* et *tail_n* qui donnent les n premiers ou derniers éléments dans une séquence. Le code suivant génère les deux premiers et deux derniers éléments d'un vecteur en contenant 4.

```
int a[] = { 1,2,3,4 };
vector<int> v (a+0, a+4);

lref<vector<int> > h;
lref<vector<int> > t;

head_n(v, 2, h)();
// h contient {1,2}
tail_n(v, 2, t)();
// t contient {3,4}
```

L'argument supplémentaire sert à spécifier le nombre d'éléments qu'on souhaite avoir dans la tête ou la queue. Une exception sera lancée si la taille de la séquence est plus petite que celle spécifiée dans *head* ou *tail*

III-8-2-B - Itération avec next et prev

La relation *next* représente une relation binaire entre une valeur et celle qui la suit. Étant donné que le successeur d'un pointeur/itérateur est un autre pointeur/itérateur qui pointe sur le prochain élément, on peut utiliser *next* pour réaliser l'itération. Le code suivant montre le successeur de 2.

```
lref<int> s;
next(2,s)();
cout << *n;
```

next peut aussi être utilisé pour générer la valeur précédente:

```
lref<int> p;
next(p,2)();
cout << *p;
```

Quand les deux arguments sont définis, *next* va vérifier que le second argument est bien la suite du premier. Une exception sera lancée si les deux arguments sont indéfinis. De façon similaire à *next*, *prev* est aussi disponible. La seule différence entre les deux est que l'ordre des arguments est inversé. *next* et *prev* peuvent être utilisé l'un à la place de l'autre en fonction de la préférence du programmeur et de ce qui est le plus lisible dans un contexte donné. Le code suivant démontre l'usage de *next* afin d'afficher tous les éléments d'une paire d'itérateurs'

```
relation printAll( lref<int*> beg_
                 , lref<int*> end_ ) {
    lref<int> val; // for storing **beg_
    lref<int*> n;
    return predicate(beg_==end_
        ^ ( dereference(beg_, val)
            && write(val)
            && next(beg_,n)
            && recurse(&printAll,n,end_ ) );
}

int ai[]={1, 2, 3};
printAll(ai+0, ai+3)();
```

La relation *printAll* passe sur tous les éléments du conteneurs en produisant récursivement le successeur en itérant sur *beg*. A chaque étape de la récursion, on vérifie que la séquence n'est pas vide en comparant *beg* et *end*. Si un élément est présent dans la séquence, on déférence *beg* (d'une manière relationnelle ?? en utilisant *deference*) pour obtenir la valeur pointée puis l'afficher avec *write*. Après avoir affiché la première valeur, on procède à une récursion en produisant l'élément suivant de *beg* dans *n*. Le code suivant peut aussi être ré-écrit en terme d'itérateurs provenant de la STL, tel que `vector<int>::iterator` à la place d'un simple pointeur en substituant toutes les occurrences de *int** par `vector<int>::iterator` dans le code précédent.

III-8-2-C - Itération avec item

Les relations *next* et *prev* sont très utiles lorsque un contrôle explicite sur le processus d'itération est requis. Dans le cas où on souhaite juste accéder à toutes les valeurs d'une séquence une par une, la relation *item* fournit une alternative simple. *item* prend trois arguments: les deux premiers sont une paire d'itérateurs (ou une paire de référence logique sur des itérateurs) représentant la séquence et le troisième élément est une référence logique. Un élément de la séquence est affecté au troisième argument à chaque évaluation de *item*. Les deux premiers arguments doivent être définis.

```
int ai[] = { 1, 2, 3, 4 };
vector<int> vi(ai+0, ai+4);
lref<int> val;
```

```
//on itere avec des itérateurs classiques
relation r = item(vi.begin(), vi.end(), val);
while(r())
    cout << *val << ", ";

// 2 - itération avec des références logiques sur des itérateur.
lref<vector<int>::iterator> lBeg = vi.begin()
                        , lEnd = vi.end();
r = item(lBeg, lEnd, val);
while(r())
    cout << *val << ", ";
```

Dans le code ci-dessus, le troisième argument est laissé non-initialisé afin d'extraire des valeurs depuis la séquence. *item* peut aussi être utilisé afin de vérifier si une valeur particulière est présente dans la séquence.

```
if( item(vi.begin(),vi.end(),4) () )
    cout << "found!";
```

III-8-3 - Unification d'ensemble

L'unification fournie par la relation *eq* telle qu'elle est décrite dans la section 2 au dessus n'est pas limitée aux types scalaires. L'unification peut aussi être réalisée sur des ensembles d'une manière similaire.

```
// produit une séquence
int a[] = { 1,2,3,4 };
vector<int> v (a+0, a+4);
lref<vector<int> > lrv;
eq(lrv,v) ();
assert(*lrv==v);

//compare les objets
lref<list<int> > lrl= list<int>(a+0,a+4);
if(eq(lrl,v) ())
```

La relation *eq* fournit une unification basique pour les ensembles. Dans la section 2, nous avons parlé de comment la relation *sequence* peut être utilisée afin de créer des ensembles. Quand le premier argument n'est pas initialisé, *sequence* génère une séquence remplie avec les éléments spécifiés dans les arguments restants. Cependant, si le premier argument est initialisé, cela va produire une comparaison entre la séquence et les éléments provenant des arguments restants. Ceci fait de *sequence* un moyen d'unification très puissant pour les ensembles. Nous avons déjà vu des exemples pour produire des ensembles. L'exemple suivant en montre l'utilisation pour la comparaison.

```
int a[] = {1,2,3};
lref<list<int> > lrl= list<int>(a+0,a+3);
lref<int> lri=1;
vector<int> v; v.push_back(2);
relation r = sequence(lrl) (lri) (v.begin(), v.end()) (3);
assert(r()); // l'évaluation échoue.
```

A l'inverse de *eq*, *sequence* autorise la création/comparaison de séquences en utilisant indifféremment un mélange d'autres séquences, d'itérateurs, de références logiques sur d'autres séquences,...

III-8-4 - Résumé

Il y a un grand nombre de moyens pour travailler avec des ensembles et Castor fournit seulement un support pour les plus utiles d'entre elles. On peut directement travailler avec une collection en entier ou alors via des itérateurs. Les relations *item*, *next*, *prev* et *deref* sont des utilitaires légers pour travailler avec une collection en utilisant des itérateurs. Les relations *eq*, *sequence*, *head*, *tail* et *item* permettent de travailler sur la collection entière. *sequence*

est la technique la plus riche et flexible mais aussi la plus lourde comparée avec les autres alternatives. D'autres relations comme *empty*, *size*, *insert*, *merge*, *eq_seq*,... sont aussi disponibles afin de travailler avec les collections et les itérateurs. Vous pouvez vous référer au manuel pour avoir une liste complète.

III-9 - Les coupes, des élagages alternatifs

Il est parfois très utile d'éliminer des chemins qui ne produiront pas de solutions ou alors le double d'une solution existante quand le processus de backtracking se produit. De tels élagages explicites sont souvent faits pour des raisons de performance. Il n'y a pas de raison de perdre du temps dans la poursuite d'un chemin qui est connu pour ne mener à rien d'intéressant. Considérons la relation suivante qui cherche une valeur à travers un arbre binaire:

```
// Binary search tree
struct BST {
    BST* l; // sous-arbre gauche
    BST* r; // sous-arbre droit
    int value; // noeud courant
    ...
};

relation b_search(lref<int> val
                , const BST* tree) {
return
    predicate(val==tree->value)
|| predicate(val<tree->value)
    && recurse(b_search, val, tree->l)
|| predicate(val>tree->val)
    && recurse(b_search, val, tree->r)
;
}
```

La relation `b_search` est constituée de trois clauses. La première vérifie que la valeur du noeud courant n'est pas celle qu'on cherche. Les deux clauses restantes vont chercher récursivement dans les sous-arbres gauche et droit en fonction du résultat de la comparaison entre `val` et le noeud courant. Si la première égalité est vraie, il est évident que les autres clauses peuvent être éliminées. Une fois que la valeur aura été trouvée, nous voudrions que le processus de backtracking reste à la clause courante et ignore les autres alternatives possibles. Cette capacité à éliminer des chemins alternatifs est appelée une *coupe* en programmation logique. La classe `cut` et la relation `cutexpr` fournissent un support pour les coupes dans Castor. On peut ré-écrire le code au dessus en utilisant les coupes tel qu'il suit:

```
return cutexpr(
    predicate(val==tree->val) && cut()
|| predicate(val<tree->val) && cut()
    && recurse(b_search, val, tree->l)
|| predicate(val>tree->val)
    && recurse(b_search, val, tree->r)
);
}
```

Chaque occurrence de `cut` marque le point sur lequel nous décidons de **commit** à la clause courante. La place de `cut` est appelée un point de coupe (*cut point* en Anglais). Une `cutexpr` marque seulement les frontières à l'intérieur desquelles à lieu l'élagage supplémentaire. Dans ce cas, `cutexpr` englobe les trois clauses dans la relation. Une fois que l'exécution atteint la fin d'un point de coupe, le retour commencera à l'endroit où la `cutexpr` apparaît et ira jusqu'au point de coupe. Ainsi, toutes les autres alternatives disponibles après `cutexpr` seront retirées des chemins disponibles pour le processus de backtracking. Les coupes n'affectent pas les alternatives qui existent avant la `cutexpr` ou les alternatives qui existent après le point de coupe. La `cutexpr` elle-même ne fait que simplement fournir la portée à l'intérieur de laquelle la coupe peut avoir lieu.

Une coupe non entourée d'une `cutexpr` ou une `cutexpr` sans aucune coupe sont tous les deux des non-sens. De telles choses peuvent apparaître accidentellement lorsqu'on retire des coupes qui font usage de nombreuses coupes. Par le design de Castor, de telles erreurs ne passeront pas à la compilation. Les usages suivants de coupes où une `cutexpr` apparaît dans l'appelant et `cut` dans l'appelé ne sont pas autorisés.

```
// Error: cannot dynamically nest cuts
relation outer(...) {
    return cutexpr( inner(..) || ... );
}

relation inner() {
    return ... && cut() ...
}
```

Un usage excessif de coupes dans un programme logique est généralement déconseillé car cela tend à rendre le programme moins lisible. Aussi, quand elle ne sont pas utilisées avec parcimonie, les coupes peuvent supprimer des chemins valides. L'usage des coupes devrait premièrement être réservé à des cas où le gain de performance est significatif. Il est préférable qu'une relation qui utilise une coupe produise le même résultat que sa version sans coupe. Des coupes qui n'interfèrent pas dans les résultats sont appelées *coupes vertes*. Les coupes qui ne sont pas vertes sont rouges. Beaucoup de coupes peuvent être remplacées de manière plus élégante par la relation de l'opérateur ou-exclusif.

III-10 - Opérateur ou-exclusif

Nous avons vu les opérateurs && et || pour définir des relations. Castor fournit aussi un moyen pour définir des relations de choix exclusifs entre des clauses à l'aide de l'opérateur ^. Cette expression est très utile car elle permet d'évaluer la seconde clause si seulement la première échoue. Regardons de nouveau l'exemple de *greaterLessEq*

```
relation greaterLessEq(lref<int> n
    , lref<int> cmpVal) {
    return predicate(n<cmpVal) && write("n<cmpVal")
        || predicate(n>cmpVal) && write("n>cmpVal")
        || write("n==cmpVal");
}
```

L'utilisation de la relation ci-dessus expose à un problème:

```
relation r = greaterLessEq(2,3);
while(r());
```

va produire la sortie suivante: n<cmpVal
n==cmpVal

Ceci est dû à l'utilisation de while qui oblige le processus de backtracking à se produire même si la première clause correspond. La seconde clause échoue à cause de la comparaison au début et mène à l'évaluation de la troisième clause. Étant donné que la troisième clause n'a pas la protection (n==cmpVal), elle réussit et on observe la sortie ci-dessus. Une solution simple est d'introduire des coupes dans l'expression. Cependant une solution plus simple est de ré-écrire la relation avec l'opérateur ^ tel qu'il suit:

```
relation greaterLessEq(lref<int> n
    , lref<int> cmpVal) {
    return ( predicate(n<cmpVal)
        && write("n<cmpVal") )
        ^ ( predicate(n>cmpVal)
        && write("n>cmpVal") )
        ^ ( write("n==cmpVal") );
}
```

Ici, nous sommes explicites sur le fait que les trois clauses sont mutuellement exclusives. Ce n'est pas seulement plus explicite mais aussi plus lisible et efficace. Une fois qu'une clause de la relation a réussi, la relation réussit aussi. Le backtracking va ignorer toutes les autres clauses qui n'ont pas été évaluées. Ainsi, toute tentative pour chercher plus de solution depuis la relation échouera. Il est important de noter la paire supplémentaire de parenthèse

autour de chaque clauses. En effet, la priorité de l'opérateur `^` est plus élevée que celle des opérateurs `&&` et `||` d'où le besoin de parenthèses en plus pour garder une associativité correcte.

La plupart, mais pas toute, des usages de coupes peuvent être remplacés par l'opérateur ou-exclusif. Le support l'opérateur `^` pour les relations est quelque chose d'unique à castor.

III-11 - Manières de passer une référence logique à une relation

Il y a trois moyen de passer une référence logique à une relation

- Par valeur `lref<T>`
- Par référence `lref<T>&`
- Par référence constante: `const lref<T>&`

III-11-A - Par valeur

Si le paramètre de la relation est `lref<T>`, les arguments acceptables sont `T` et `lref<T>`. Cette versabilité fait de ce mécanisme un choix commun pour spécifier les paramètres dans une relation. Quand l'appelant passe un argument du type `lref<T>`, l'appelée le reçoit tel que la nouvelle référence pointe sur la même valeur sous -adjacente. Ce système est similaire aux pointeurs classiques du C++, l'appelée reçoit une copie (sur la pile) à laquelle se réfère l'appelant.

```
relation foo(lref<int> i) {
    return True();
}

lref<int> li=2;
foo(1); // OK! passes a copy of 1
foo(li); // OK!
```

Ce mode de passage est le plus couramment utilisé dans Castor.

III-11-B - Par référence

Pour certains types où la copie est potentiellement très coûteuse (tel `std::vector`), on peut désirer prévenir les différentes copies implicites qui peuvent se produire. Ceci peut être fait en désactivant la possibilité de passer un argument de type `T` et en demandant directement un argument de type `lref<T>`. Ceci peut aussi fait en spécifiant le paramètre en tant que `lref<T>&`

```
relation foo(lref<int>& i) {
    return True();
}

lref<int> li=2;
foo(1); // Error!
foo(li); // OK!
```

Toutes les relations de castor qui demandent en paramètre un conteneur (comme `empty` ou `size`) utilise cette forme de passage pour les arguments.

III-11-C - Par référence constante

Ce moyen de passage est similaire au précédent avec néanmoins une différence significative. Un paramètre de type référence constante ne peut pas être passé à une relation qui attend une référence en tant que paramètre. Exemple:

```
relation bar(lref<int>& j) {
    return ...
}

relation foo(const lref<int>& i) {
    return bar(i); // Error!
}
```

Une autre différence, mais mineure, est que la valeur sous-jacente de la référence ne peut pas être modifiée directement dans la relation.

```
relation foo(const lref<int>& i) {
    i=2; // Error!
    return bar(i);
}
```

Ceci est rarement un vrai sujet étant donné que mélanger des pratiques impératives avec du code déclaratif ne doit jamais être fait en pratique. Néanmoins, l'affectation de `i` à 2 est fait lorsque la relation est appelée, pas évaluée.

III-12 - Debug

Malheureusement, dans un contexte impératif, déboguer du code déclaratif n'est pas aussi simple que de déboguer du code impératif. Les débogueurs C++ sont fait pour observer du code impératif. En principe, étant donné que nous laissons à l'ordinateur le soin de trouver la solution, nous ne devrions pas être concerné par la manière dont il s'y prend. En pratique, il est très utile de regarder l'évaluation progresser. C'est important pour vérifier la correction des relations, en particulier quand les résultats ne sont pas conformes à ce qui est attendu.

Pour corriger un programme écrit dans le paradigme logique, une certaine façon penser est nécessaire. Étant donné que les relations sont évaluées de manière paresseuse, il n'est pas très utile de placer un breakpoint dans le code. En fait, certaines références logiques des relations peuvent ne pas être initialisées à l'endroit où la relation est appelée mais au moment où la relation commence l'évaluation. Alors que la majorité des des exécutions survient à l'intérieur des foncteurs retournés par les opérateurs `&&`, `||` et la relation `eq`, l'usage de breakpoint pour déboguer demande la compréhension des détails d'implémentation de castor.

Une technique simple (primitive même) est d'insérer des instructions pour tracer la progression du code. Castor fournit la relation `write` pour écrire sur `stdout` selon une manière relationnelle. Dans l'exemple suivant, nous ajoutons des instructions de trace à la relation `ancestre` afin de voir la solution se générer au fur et à mesure.

```
relation ancetre_dbg(lref<string> A
                    , lref<string> D) {
    lref<string> P;
    return enfant(D,A)
        || enfant(D,P) && write(P) && write(",")
        && recurse(ancestor, A, P);
}
```

Le code suivant tente de trouver tous les ancêtres de Sam.

```
lref<string> X;
relation a = ancestor_dbg(X, "Sam");
while(a())
    cout << " : " << *X << " est un parent de de Sam \n";
```

Et va produire la sortie suivante:

```
:Mary est un parent de de Sam
:Frank est un parent de de Sam
Mary, Frank, :Gary est un parent de de Sam
Gary,
```

On peut voir en regardant la sortie que les ancêtres directs (Frank et Mary) sont découverts sans avoir recourt à la récursion. Une fois que les deux parents de Sam ont été trouvés, la sortie montre que le processus de backtracking procède à une évaluation de la clause récursive et recherche les parents de Mary.

IV - Créer des relations impératives

Cette capacité à facilement jongler entre les différents paradigmes est quelque chose d'essentiel dans un contexte multi-paradigme. Jusqu'à maintenant, nous avons vu des exemples de relations construites à partir d'autres relations et comment les relations peuvent être utilisées dans du code impératif. Dans cette section, nous allons boucler la boucle en regardant comment des relations peuvent utiliser du code impératif.

Les relations peuvent être définies de manière impérative ou déclarative. Jusqu'à maintenant, elles ont été faites selon un mode déclaratif puisque le programmeur n'est pas impliqué dans la manière de résoudre le problème. Produire des relations d'une manière impérative est en général plus de travail que produire les mêmes résultats de manière déclarative. En effet, les relations déclaratives sont plus faciles à lire et à écrire. Cependant, il existe des situations où on peut être amené à écrire des relations dans un style impératif. Les raisons classiques peuvent être:

- Interactions avec des éléments plus bas niveau (entrées/sorties, mémoire, drivers, ...) ou d'autres éléments impératifs où les abstractions relationnelles ne sont pas disponibles ou inadéquates.
- Un contrôle plus fin sur l'exécution lorsque les performances le demandent.
- Améliorer la capacité d'explorer la relation avec un débogueur

Nous allons décrire les deux approches pour implémenter des relations de façon relationnelle. Le choix de l'approche peut être fait en fonction de la complexité de la relation.

IV-1 - Avec la relation predicate

Une relation test est une relation qui ne modifie aucun de ses arguments et donc qui n'introduit aucun effets de bord dans le système, chose qui pourrait interférer dans le processus de backtracking. Une telle relation est très utile pour tester si une condition est vraie. Généralement, à cause de leur nature profonde, les relations tests réussissent au moins une fois. La relation *predicate* est très utile pour définir de telle relation. La relation test peut être empaquetée dans une fonction classique qui renvoie un booléen et ensuite appelée à l'aide de la relation *predicate*. Considérons la relation suivante pour vérifier si un fichier existe:

```
bool fileExists_pred(string fileName) {
    if(/* fichier existant */)
        return true;
    return false;
}

relation file_exists (lref<string> fileName_) {
    return predicate(fileExists_pred, fileName_);
}
```

Le processus est en deux étapes distinctes. Premièrement, la fonction-prédicat *fileExists_pred* contient le code impératif pour vérifier si un fichier existe. Deuxièmement, la relation *file_exists* fournit un wrapper à la fonction-prédicat *fileExists_pred* en utilisant la relation *predicate* pour l'appeler. Noter que la relation prend une référence logique alors que la fonction-prédicat prend un `std::string`. Alors que les deux paramètres pourraient être une référence logique de `std::string`, il est plus naturel qu'une relation prenne une référence logique et une fonction-prédicat un type classique. La relation *predicate* suppose que la fonction cible ne prend pas une référence logique en argument. En conséquence, si `filename_` est une référence logique, elle est automatiquement déréférencée lors de l'appel de la fonction. Si `filename_` n'est pas une référence logique, il sera passé directement.

Il faut aussi noter l'utilisation de '_' à la fin du nom du paramètre. Ceci permet de spécifier que l'argument n'est pas bi-directionnel. Ainsi, sa valeur doit être définie au moment de l'évaluation de la relation. Si elle n'est pas définie à ce moment, une exception du type *InvalidDeref* est lancée lorsque la relation *predicate* tente de déréférencer `filename_`.

IV-2 - En tant que coroutine

Chaque relation est en vérité l'instance d'un foncteur. Jusqu'ici, nous avons défini les relations en tant que fonctions et expressions. Étant donné que cette approche syntaxique constitue un moyen déclaratif de spécifier les détails des

opérations du foncteur sous-jacent, le nom de son type et les définitions impératives nous sont cachées. Dans cette section, nous verrons comment créer ces foncteurs directement. Ceci à l'avantage de nous donner un contrôle total sur les étapes de résolution et ainsi la relation est plus facilement déboguable. Cependant, Le mauvais coté est que cela demande d'écrire plus de code et de faire plus attention.

Nous avons vu que l'évaluation paresseuse est une partie intégrante du fonctionnement des relations. A chaque fois qu'une valeur est calculée, elle immédiatement retournée à l'appelant. Au prochain appel de la relation, elle recommence où elle s'est arrêtée, génère une autre valeur et a renvoie à l'appelant. Ce mode de fonctionnement est à l'opposé des fonctions qui calculent toutes les valeurs avant de les renvoyer. Les fonctions qui supportent le mécanisme d'arrêt-reprise sont appelées *coroutines*.

Nativement, le C++ ne supporte pas les coroutines. Castor fournit la classe `Coroutine` et quatres macros (`co_begin`, `co_end`, `co_yield`, `co_return`) pour cette tâche. Bien que ces utilitaires soient fait pour créer des relations en tant que classes, on peut les utiliser en dehors du contexte de la programmation logique.

Dans Castor, une coroutine est un foncteur. Pour définir notre classe, nous la dérivons depuis la classe `Coroutine` et implémentons la fonction `bool operator()(void)` tel qu'il suit:

```
// Une relation pour générer ou vérifier si une valeur est dans un certain intervalle
template<typename T>
class Range_r : public Coroutine {
    lref<T> val, min_, max_;
public:
    Range_r(lref<T> val, lref<T> min_, lref<T> max_)
        : min_(min_), max_(max_), val(val)
    { }

    bool operator() () {
        co_begin();
    ...
        co_end();
    }
};
```

Notez l'usage de la macro `co_begin` pour démarrer le corps de l'opérateur et `co_end` pour terminer le corps. Aucune instruction ne doit suivre ou précéder ces deux macros dans le corps de l'opérateur. Ces deux macros définissent simplement une instruction `switch` qui s'étend sur tout l'opérateur. La définition complète de l'opérateur est la suivante.

```
bool operator() () {
    co_begin();
    if(val.defined())
        co_return( (*min_<*val && *val<*max_)
                    || (*min_==*val) || (*max_==*val) );
    for(val=min_; (*val<*max_) || (*val==*max_)
        ; ++val.get())
        co_yield(true);
    val.reset(); // Important pour le backtracking
    co_end();
}
```

Notez l'absence d'instructions `return`. Elles sont remplacées par `co_return` et `co_yield`. Les deux macros retournent à l'appelant la valeur qui résulte de l'évaluation de leurs arguments.

La macro `co_yield` indique un point où l'exécution est suspendue temporairement et une valeur `true/false` est retournée à l'appelant. Au prochain appel, l'opérateur () va reprendre son exécution directement à l'intérieur de la méthode, directement à ce *yield point*. Utiliser cette macro impose à la coroutine de se rappeler quel point à atteint l'exécution précédente. D'un autre coté, utiliser `co_return` indique qu'aucune autre reprise d'exécution est nécessaire et la valeur `true/false` produite par la relation est renvoyée à l'appelant. Tous les autres appel à l'opérateur () renverront faux immédiatement.

Conceptuellement, `co_yield` indique un arrêt temporaire de l'exécution et `co_return` une exécution qui est finie. Étant donné qu'en programmation logique, renvoyer faux est aussi un moyen pour signaler que l'exécution est finie, `co_yield(false)` indique aussi que l'exécution est finie. Quand l'argument de `co_yield` est évalué à faux, tous les futurs appels à l'opérateur () retournerons faux immédiatement. Ce comportement est similaire à `co_return(false)`.

La définition de l'opérateur() vérifie tout d'abord si *val* est initialisé afin de déterminer si la relation doit générer une valeur ou simplement vérifier si elle est dans l'intervalle. Si *val* est initialisé, on doit seulement réaliser un test, sinon on doit générer des valeurs pour *val*. Le corps de l'instruction *if* compare *val* aux bornes et renvoie le résultat à l'appelant. Étant donné qu'après les tests, il n'y a rien plus à faire, *co_return* est utilisé afin de retourner le résultat de l'opération. Dans le cas où *val* n'est pas initialisée, la boucle est exécutée. Le haut de la boucle génère des valeurs pour *val* allant de min à max. 0 chaque itération, le corps de la boucle renvoie vrai à l'appelant. A chaque nouvel appel de l'opérateur, l'exécution reprend là où elle s'était arrêté et réalise ainsi une itération de plus. A chaque fois, *val* est incrémenté et on retourne vrai à l'appelant. Une fois que *val* dépasse la valeur maximale, la boucle prend fin et *val* est remise dans son état de départ. Faire cette remise à zéro est important car le processus de backtracking demande que toutes les relations annulent les effets de bord qu'elles ont pu avoir avant de signaler l'achèvement. Déléguer la réversion des effets de bord aux relations permet au processus de backtracking d'être simple et efficace. Quand le foncteur est générique (à l'aide de template), il est courant de fournir une fonction d'aide afin de permettre une déduction automatique du type comme il suit:

```
template<typename T>
Range_r range(lref<T> val, lref<T> min_
              , lref<T> max_) {
    return Range_r<T>(val,min,max);
}
```

Une chose importante de garder à l'esprit est que lorsqu'on implémente des coroutines, il faut mieux éviter de définir des variables non statiques à l'intérieur de l'opérateur () car les valeurs ne peuvent pas persister entre deux appels. Ces variables devraient plutôt être promues en tant que variable membres.

V - Inline Logic Reference Expressions

Une ILE est une expression composée d'au moins une référence logique et des opérateurs surchargés les plus courants. A l'inverse des expressions classiques en C++, les ILEs ne subissent pas l'évaluation pour produire un résultat au moment de leur définition. A l'inverse, elles produisent un arbre d'expression qui représente la sémantique de l'expression. Cet arbre d'expression peut être évalué plus tard afin de produire le résultat. L'exemple suivant montre la sémantique des ILEs:

```
int i=0; // variable classique
i+1+1; // expression classique, vaut 2

lref<int> li=2; // référence logique
(li+i)*2; // ILE: produit un arbre d'expression.
```

L'exemple suivant utilise une ILE là où une fonction était attendue.

```
template<typename Func>
void printResult( Func f ) {
    cout << f();
}

printResult( li*li / 2 ); // passe une ILE en tant qu'argument.
```

Dans le code ci-dessus, l'ILE subit l'évaluation à l'intérieur de *printResult* quand *f* est évaluée. On peut aussi retourner des fonctions depuis d'autres fonctions.

```
void runtests();

boost::function<int()> halfOf( lref<int> li ) {
    return li/2; // renvoie une ILE
}

boost::function<int()> f = halfOf(4);
cout << f(); // l'ILE est évaluée ici
```

Les ILEs fournissent un moyen efficace de créer des expressions dont l'évaluation doit être retardée. L'avantage vient du fait qu'on n'est pas obligé d'encapsuler explicitement les ILEs dans une fonction classique. De telles expressions sont appelées des *lambdas expressions*. Chaque ILE contient une copie de toutes les variables et valeurs qui lui sont nécessaires pour calculer l'expression. Cet ensemble de variables et valeurs est appelée la *closure* de l'ILE. Comme copier des références logiques équivaut à copier des pointeurs (c'est-à-dire que la copie se réfère à la même valeur sous-jacente), tous les changements à l'objet original se répercuteront à la copie de l'ILE. Quand de multiples ILE se réfèrent à un même objet, l'objet sera toujours vivant tant que le compteur de références ne sera pas à zéro. Ceci rend l'évaluation de l'ILE sûre même après qu'on ait quitté la portée où l'ILE a été créée. Cette sécurité est d'autant plus essentielle quand retarde l'évaluation de l'ILE. L'exemple suivant illustre la sémantique des closures.

```
int x=2;
lref<int> lx=3;
boost::function<int()> ile = lx+x;
cout << ile(); // affiche 5
x=1; // ceci ne sera pas observable par l'ILE car elle a une copie de x'
cout << ile(); // affiche 5
lx=4; // met à jour la valeur, visible par l'ILE.
cout << ile(); // affiche 8
```

Comme une ILE est une expression et non une fonction, elle n'accepte aucun argument au moment de l'évaluation. C'est une expression auto-suffisante qui peut être évaluée à n'importe quel moment et autant de fois qu'on en a besoin. Comme nous l'avons vu au dessus, chaque évaluation peut potentiellement produire un résultat différent.

Les changements qui ont lieu sont causés par le code extérieur qui modifie l'objet pointé par une des références logiques, comme dans le code au dessus. Un changement peut aussi être dû à l'ILE qui introduit des effets de bord dans sa propre closure tel qu'il suit:

```
lref<int> lx=3;
boost::function<int()> ile = ++lx;
cout << ile(); // affiche 4
cout << ile(); // affiche 5
cout << ile(); // affiche 6
```

Des ILEs tel que qui introduisent des effets de bord sont dites impures. Celles qui ne le font pas, comme $x+2$ sont dites pures.

V-1 - Créer des relations à partir d'ILEs

A l'origine, les ILEs ont été conçues en tant que moyen facile et rapide pour créer des relations anonymes depuis une simple expression au lieu de créer une relation nommé. Les ILEs qui retournent un booléen peuvent devenir une relation à l'aide de la relation *predicate*. La relation *predicate* est un adaptateur qui permet aux fonctions ou aux foncteurs (qui ont jusqu'à 6 arguments maximums) d'être utilisés comme une relation. Une relation créée en passant une ILE à *predicate* est appelée relation-ILE.

L'exemple suivant montre l'usage des ILEs pour générer les nombres pairs.

```
//Print all even numbers in the inclusive range 1...20
lref<int> x;
relation evens = range<int>(x,1,20)
                && predicate(x%2==0);
while (evens())
    cout << *x << " ";
```

Au dessus, la relation-ILE $predicate(x\%2==0)$ teste si x est pair. On commence avec x qui n'est pas initialisé. la relation *range* génère des valeurs pour x et comme est dans la closure de l'ILE, tous les changements sur x seront visibles dans l'ILE. Ceci permet à la relation-ILE de tester les valeurs une par une. Ceci est un exemple du pattern classique générer et tester utilisé en programmation logique. Dans les deux exemples, la relation *range* sert à générer des valeurs et la relation-ILE à filtrer les valeurs qui ne correspondent pas.

L'exemple suivant montre l'utilisation d'ILEs pour générer des triplets de Pythagore.

```
// Affiche tous les triplets de Pythagore inférieure à 30.
lref<int> x,y,z; int max=30;
relation pythTriplets =
    range<int>(x,1,max)
    && range<int>(y,1,max)
    && range<int>(z,1,max)
    && ( predicate(x*x+y*y==z*z)
        || predicate(z*z+y*y==x*x) );
while (pythTriplets())
    cout << *x << ", " << *y << ", " << *z << "\n";
```

Ici, les trois utilisations de *range* génèrent des valeurs pour x , y et z qui sont comprises entre 1 et 30 et les deux relations-ILE vérifient si x , y et z forment un triplet de Pythagore.

Un autre usage courant des ILEs est lié à la relation *eq_f* qui prend une fonction ou un foncteur en tant que second argument. Elle unifie son premier argument avec la valeur obtenue après évaluation de son second argument. Ainsi, si x et y sont des références logiques, alors $eq_f(x, y*3)$ unifie x avec la valeur obtenue en évaluant $y*3$.

Les ILEs impures ne devraient pas être utilisées pour créer des relations-ILE. Si de telles relations produisent des effets de bord, alors ils ne seront pas inversé durant le processus de backtracking et conduiront à des résultats étonnants. Considérons $predicate(++x<5)$ qui incrémente la valeur de x à chaque évaluation. Ici, l'ILE est impure car elle modifie la valeur de x à chaque évaluation. Le backtracking compte sur la réversibilité des effets de bord. Quand

de tel effets de bord sont voulus, ils devraient être regroupés dans une relation qui s'assure d'annuler les effets de bords. La relation *predicate(++x<5)* peut être ré-écrite en *inc(x) && predicate(x<5)* ou *next(x,y) && predicate(y<5)*. Ainsi, on incrémente *x* et on l'unifie plus tard *y* avec la valeur incrémentée de *x* (sans affecter la valeur de *x*). C'est ce style qu'on préfère en général.

L'avantage d'une relation-ILE est sa brièveté, le mauvais côté est qu'elle ne peut pas faire autant de choses qu'une relation classique. Les relations-ILE peuvent seulement réaliser des tests sur des valeurs produites par d'autres relations, elle sont incapables de générer des solutions par elles mêmes. Ainsi, toutes les références logiques impliquées dans une ILE doivent être initialisées au moment de l'évaluation.

V-2 - Limitations des ILEs

Il existe des limitations aux ILEs dont les utilisateurs doivent être conscients. La première est due au langage, la seconde au design.

Une question qui peut sembler dénuée de sens si quelqu'un la pose est : "Quel est le type du résultat produit pas l'évaluation d'une ILE ?". La réponse est bien sûr : "Cela dépend de l'expression représentée par l'ILE". Une réponse plus précise pourrait être : "Le même type que l'expression qui résulterait d'une ré-écriture en remplaçant les références logiques avec leurs types effectifs". Ainsi, l'expression *x*y* devra retourner un objet du même type et de même valeur que *x* ou *y* soient des références logiques ou des variables classiques. Malheureusement, ceci n'est pas toujours vrai étant donné qu'il n'est pas toujours possible de calculer le type de retour d'une expression arbitraire ou d'un appel de fonction en C++. On infère le type de retour de l'expression à l'aide des règles suivantes:

- Tous les opérateurs de comparaison ainsi que *&&*, *||* et *!* retournent un booléen.
- Le type de retour de la version pré-fixée de *++* et de *--* est *T&* où *T* est le type de l'argument
- On suppose que tous les autres opérateurs (unaire ou binaire) ont un type de retour identique à celui de leur premier argument.

Un moyen facile pour déterminer le type de retour d'une ILE est d'observer si dans son ensemble elle représente ou non une comparaison. Si oui, elle renvoie un booléen, sinon le type de retour est déterminé par le premier argument de l'opérateur qui est évalué en dernier dans l'ILE. Voici quelques exemples:

```
lref<int> x=3;
double y=1;
2.0 * x; // double
x * 2.0; // int
y*x; // double
x*y; // int
2*x + 3.1; // int
3.1 + 2*x; // double
5 < x; // bool
x == x; // bool
x < 3*x+5; // bool
3*x+5 < x; // bool
++x; // int&
```

Pour les opérateurs, tous les opérateurs surchargeables sauf les suivants sont supportés pour la création d'ILEs.

- L'opérateur *&* pour l'obtention d'adresse
- L'opérateur *** pour déréférencer
- L'opérateur *-<* pour accéder aux membres
- L'opérateur *[]*
- L'opérateur virgule
- Tous les opérateurs d'assignation (*=*, *+=*, **=*,...)

Comme l'assignation est utilisée pour assigner une nouvelle valeur à une référence logique, il est évident que ses opérateurs ne peuvent pas être utilisés pour produire une ILE. De façon similaire, l'opérateur de déréférencement et l'opérateur *-<* sont utilisés pour accéder aux valeurs sous-jacentes et cela les évince des ILEs. L'opérateur de prise d'adresse et l'opérateur virgule prennent la sémantique par défaut du C++ quand ils sont utilisés sur des *lref*. Finalement, il n'y a aucun moyen de connaître le résultat de l'opérateur d'indexage, il reste ainsi non supporté.

V-3 - Résumé

La liste suivante est le résumé essentiels des différents points couverts dans cette section sur les ILEs

- Les ILEs sont des expressions pouvant être traitées comme des foncteurs
- Chaque ILE possède sa propre closure, ainsi elle peut être évaluée en toute sécurité même après la sortie de portée de certaines variables
- Les ILEs sont soit pure soit impure
- Seuls les ILEs pures devraient être utilisées pour créer des relations-ILE
- Les relations-ILE ne peuvent pas produire de valeurs, elles peuvent juste tester les valeurs produites par d'autres relations
- Le type de retour d'une ILE est déterminé par les règles présentées avant
- La plupart des opérateurs surchargeables sur une référence logique sont supportés pour créer une ILE

VI - Limitation du paradigme logique

VI-1 - La bi-directionnalité des références logiques

Typiquement, les références logiques en tant que paramètre d'une relation sont bi-directionnelle. Une relation génère une valeur pour une référence logique passé en paramètre si celle ci n'est pas initialisée. Cependant, ce n'est pas toujours faisable ou sensé pour une relation de générer des valeurs que pour certains de ses paramètres. La valeur de ces références doit être définie au moment où la relation est évaluée. De telle références sont des appelées paramètres d'entrée.

Considérons la relation `not_eq` qui réussie si ses deux paramètres ne sont pas égaux.

```
relation not_eq( lref<int> lhs_, lref<int> rhs_ )
{
    return ...
}
```

Dans ce cas, c'est évident de comparer les deux arguments pour vérifier s'ils sont égaux ou non. Mais si un des deux arguments n'est pas initialisé, sur quelle bases pouvons nous générer une valeur pour qu'elle soit différence de l'autre ? Il y a plusieurs possibilités et aucune approches ne peut être facilement proclamée meilleurs qu'une autre. Par exemple, on peut commencer à 0 et aller jusqu'à `numeric_limits<int>::max()`. Ou démarrer de `numeric_limits<int>::min()` et aller à `numeric_limits<int>::max()`. Ou encore aller à l'envers en partant de `numeric_limits<int>::max()`.

Tenter de générer des valeurs dans un intervalle aussi grand est rarement un moyen très productif pour résoudre les problèmes. La futilité d'une telle approche est encore plus évidente si on considère la même relation sur une `std::string`. Ceci fait de `lhs_` et de `rhs_` des paramètres d'entrée. Par convention, de telles paramètres sont suffixés par '_' alors que les paramètres classiques ne le sont pas.

Un autre exemple est la relation `size` fournie par Castor.

```
relation size(lref<Cont>& cont_
, lref<typename Cont::size_type> sz) {
    return ...
}
```

Le premier argument est un conteneur et le second la taille du conteneur. Il est simple de tester la taille du conteneur est bien `sz` et aussi facile de remplir le conteneur de `sz` objets. Mais que se passe-t-il si les deux arguments ne sont pas initialisé ? C'est un non sens total de remplir le conteneur d'un nombre arbitraire d'éléments. Ainsi, on spécifie `cont_` comme étant un paramètre d'entrée.

Dans chacun de ces cas, c'est au consommateur de la relation de décider comment les valeurs des paramètres d'entrés doivent être générées.

VI-2 - Les entrés/sorties ne sont pas réversibles

Le processus de backtracking est par sa nature profonde très dépendant de la capacité d'annuler les effets de bord quand il poursuit de nouveaux chemins. D'un autre côté, les entrés/sorties sont quelque chose que l'on peut rarement annuler. Une fois que quelque chose a été écrit dans un socket ou un document imprimé, il n'y a pas de retour en arrière. Ainsi, entrés/sorties et backtracking sont brouillés. Il y a deux moyens de les faire co-exister. La première option est de demander la réversibilité de toutes les entrés/sorties. C'est clairement impossible à faire. La seconde option est de ne pas prendre en compte les entrés/sorties qui ont lieu lors de l'évaluation d'une relation. C'est le seul moyen réaliste pour qu'ils coexistent. Cependant, c'est au consommateur de la relation de vérifier que de tel effets de bord n'affectent pas avec les choix fait par le backtracking. Les effets de bord devraient être utilisé de façon à ne pas interférer avec le backtracking.

Les relations `write` et `read` sont des exemples de relations d'entrés/sorties dans Castor. Considérons la (mauvaise) utilisation de `read` pour vérifier si le mot lu depuis `stdin` est "Hello" ou "Word".

```
relation r = read("Hello") ^ read("World");
```

Alors que la première lecture du code peut être "lire soit Hello soit World", la réalité fait que le code suppose que si la première lecture échoue, ses effets de bords seront annulés de telle sorte que la seconde lecture puisse re-lire les données. La bonne façon de faire est la suivante:

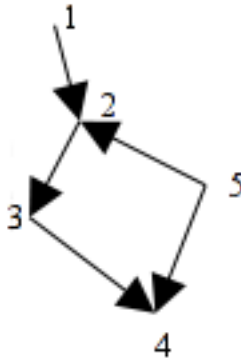
```
lref<string> w;  
relation r = read(w) &&  
    (eq(w, "Hello") ^ eq(w, "World"));
```

Ici, nous lisons la valeur (quelle qu'elle soit) et on vérifie si elle vaut soit "Hello" soit "World".

VII - Exemples de programmes logiques

Dans cette section, nous allons résoudre quelques problèmes en utilisant la programmation logique.

VII-1 - Graphe acyclique direct



Le graphe

Ici, nous allons nous intéresser à la recherche de chemin dans un graphe acyclique direct de 5 noeuds. Ce graphe peut facilement être décrit en terme d'arêtes. Dans la relation *edge* suivante, chaque ligne représente une arête directe du graphe. Les paramètres *n1* et *n2* représentent respectivement le point de départ et celui d'arrivé.

```

// représentation du graphe de la figure au dessus.
relation edge(lref<int> n1, lref<int> n2) {
    return eq(n1,1) && eq(n2,2)
    || eq(n1,2) && eq(n2,3)
    || eq(n1,3) && eq(n2,4)
    || eq(n1,5) && eq(n2,4)
    || eq(n1,5) && eq(n2,2)
;
}
  
```

Avec la définition de la relation ci-dessus, on peut dire qu'un chemin existe entre deux noeuds si:

Une arête directe existe en *Start* et *End* **OU**

Une arête directe existe entre *Start* et un noeud *X* **ET** un chemin existe entre *X* et *End*

```

relation path(lref<int> start, lref<int> end) {
    lref<int> nodeX;
    return edge(start, end)
    || edge(start, nodeX)
    && recurse(path, nodeX, end);
}
  
```

Avec les définitions présentés, nous sommes capable de poser des requêtes à propos du graphe. Une question simple est de voir si un chemin existe entre deux noeuds.

```

relation p1_4 = path(1, 4);
if(p1_4())
    cout << "Un chemin existe entre 1 et 4";
  
```

Un problème typique dans la théorie des graphes est de trouver tous les noeuds atteignables depuis un noeud donné. Le code suivant montre tous les noeuds qu'on peut atteindre depuis le noeud 5.

```
lref<int> node;
relation p5 = path(5, node);
while (p5())
  cout<< *node <<" est atteignable depuis le noeud 5\n";
```

En laissant la variable *node* non initialisée, on laisse l'ordinateur générer des valeurs pour nous. Il est intéressant de noter que le noeud 4 apparaît deux fois. Ceci est dû au fait que le noeud 4 est accessible par 2 chemins distincts depuis le noeud 5. De façon similaire, en laissant le premier argument non initialisé et en spécifiant 5 en tant que second argument, on va obtenir la liste des noeuds qui peuvent atteindre le noeud 5.

Il est aussi possible de lister toutes les arêtes existantes en laissant les deux arguments non initialisés.

```
lref<int> n1, n2;
relation p = path(n1, n2);
cout<< "Path found between these nodes\n"
while (p())
  cout << "(" << *n1 <<","<< *n2 << ") ";
```

Le code au dessus va afficher les paires dans l'ordre suivant: (1,2) (2,3) (3,4) (5,4) (5,2) (1,3) (1,4) (2,4) (5,3) (5,4). Une fois encore, a paire (5,4) est présente 2 fois car il existe deux chemins pour aller de 5 à 4.

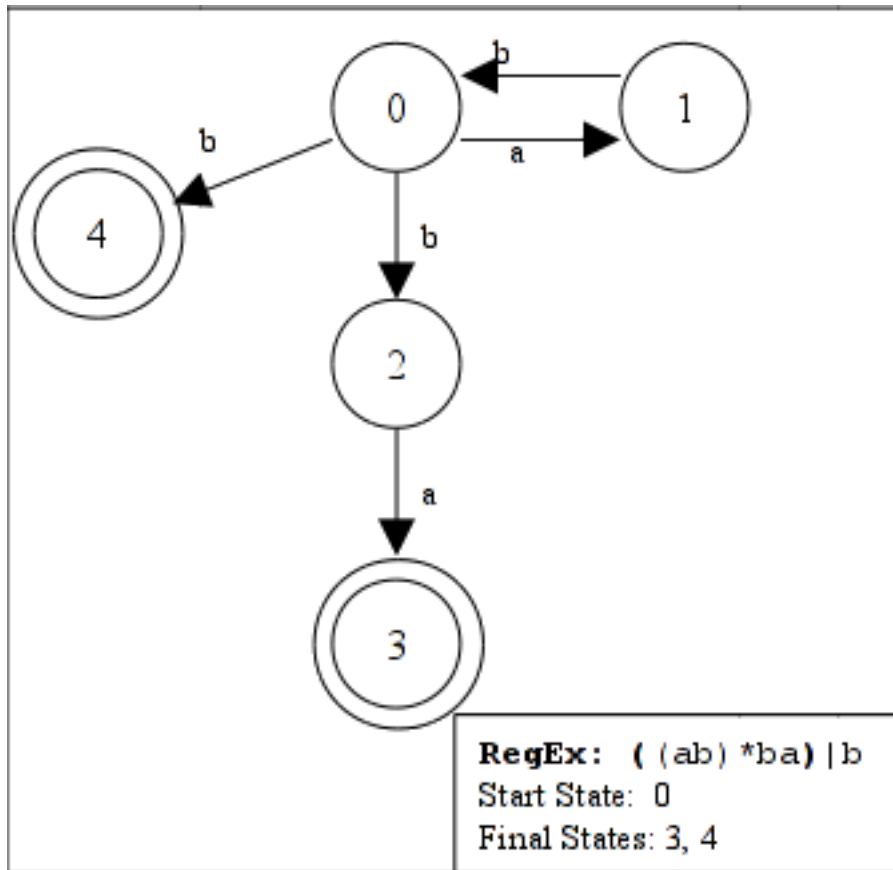
VII-2 - Automate à états finis

Un automate à états finis peut être représenté comme un ensemble de transitions et d'états final. La classe *Nfa* ci dessous exprime le non déterminisme de automate fini présent en figure 2 qui est équivalent à l'expression régulière **((ab)*ba) | b**. La classe possède deux fonctions membres statiques *transition* et *final*. La relation *transition* décrit chaque transition en terme de couple d'états et le caractère d'entrée permet de réaliser la transition. La relation *final* donne l'état final de l'automate.

Une relation libre *run* définit la règle pour que l'exécution d'un automate soit réussie en fonction d'une chaîne d'entrée. L'automate sur lequel elle opère est spécifié en tant que paramètre template. L'exécution est une réussite si:

La chaîne est vide **ET** l'état final est atteint **OU**

Il existe une transition entre l'état courant *st1* et un autre état *st2* déclenchée par le premier caractère de la chaîne **ET** l'exécution est réussie en partant depuis l'état *st2*.



L'automate'

La fonction statique *transition* de la classe NFA décrit l'état de transition et la fonction statique *final* l'état final. La relation libre *run* prend l'automate en entrée en tant que paramètre template et l'exécute.

```

class Nfa { // => ((ab)*ba|b
  static const char a = 'a', b = 'b';
public:
  // @ transitions NFA
  static relation transition( lref<int> st1
                            , lref<char> ch
                            , lref<int> st2) {

    return
      eq(st1,0) && eq(ch,a) && eq(st2,1)
    || eq(st1,0) && eq(ch,b) && eq(st2,2)
    || eq(st1,0) && eq(ch,b) && eq(st2,4)
    || eq(st1,1) && eq(ch,b) && eq(st2,0)
    || eq(st1,2) && eq(ch,a) && eq(st2,3)
  };

  // tous les états finaux de l'automate'
  static relation final(lref<int> state) {
    return eq(state,3) || eq(state,4);
  }
};

// détermine la réussite d'un automate
template<typename FA>
relation run(lref<string> input
            , lref<int> currSt=0) {
  lref<char> firstCh;
  lref<string> rest;
  lref<int> nextSt;

  return

```

```
eq(input, "") && FA::final(startSt)
|| head(input, firstCh) && tail(input, rest)
    && FA::transition(currSt, firstCh, nextSt)
    && recurse(run<FA>, rest, nextSt)
;
}
```

Avec la relation au dessus, n'importe quelle chaîne de caractère peut être testée avec l'automate.

```
if( run<Nfa>("aabba")() )
    cout << "Correspondance";
else
    cout << "Pas de correspondance";
```

VIII - Conclusion

Un grand merci à Alp pour son aide et ses conseils, à Mat007, JolyLoic, 3DArchi, Goten et à durouc05 pour les différentes relectures.

Référence:

Roshan Naik, . Ce document débat du design de castor et de l'implémentation de la programmation logique en C++
Roshan Naik,

Timothy Budd, **Multiparadigm programming in Leda** Timothy Addison-Wesley, 1995. Ce livre est une grande source de technique pour le multi-paradigme dans un langage.