Université de Montréal

Implémentation d'un langage fonctionnel orienté vers la méta programmation

par
Pierre Delaunay

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Mars, 2017

RÉSUMÉ

Ce mémoire présente l'implémentation d'un nouveau langage de programmation nommé Typer. Typer est un langage fonctionnel orienté vers la méta programmation. Il a été conçu pour augmenter la productivité du programmeur et lui permettre d'écrire des applications plus fiables grâce à son système de types.

Pour arriver à ses fins, Typer utilise l'inférence de types et implémente un puissant système de macros. L'inférence de types permet à l'utilisateur d'omettre certains éléments, le système de macros, quant à lui, permet de compléter le programme pendant la compilation lorsque l'inférence n'est pas suffisante ou pour générer du code.

Typer utilise les types dépendants pour permettre à l'utilisateur de créer des types très expressifs pouvant même être utilisés pour représenter des preuves formelles. De plus, l'expressivité des types dépendants permet au compilateur d'effectuer des vérifications plus approfondies pendant la compilation même.

Ces mécaniques permettent au code source d'être moins verbeux, plus concis et plus simple à comprendre, rendant, ainsi l'écriture de programmes ou/et de preuves plus plaisante. Ces fonctionnalités sont implémentées dans l'étape que nous appelons l'élaboration, à l'intérieur de laquelle de nombreuses transformations du code source ont lieu. Ces transformations incluent l'élimination des aides syntaxiques, la résolution des identificateurs, l'expansion des macros, la propagation et l'inférence des types.

Mots clés Lisp, Scheme, Assistant à la preuve, Coq, Type dépendant.

ABSTRACT

This dissertation present the implementation of a new programming language named Typer Typer is a functional programming language oriented towards meta programming. It has been created to increase the programmer productivity and enable him to write safer programs thanks to his type system.

To achieve his goal, Typer use type inference and a powerful macro system. Type inference enable to user to elide some elements while the macro system enable us to complete the program during compilation.

Typer use dependent type which enable the user to create very expressive types which can even be used to represent formal proofs. Furthermore, dependent type's expressivity enable the compiler to perform a in-depth checks during compilation.

Those mechanics enable the source code to be less verbose, shorter and easier to understand, making the writing of new programmes more enjoyable. Those functionalities are implemented during the step we call the elaboration in which numerous transformations occur. Those transformations include the removal of syntactic sugar, identifier resolution, macro expansion and the propagation and the inference of types.

Mots clés Lisp, Scheme, proof assistant, Coq, Dependant type.

TABLE DES MATIÈRES

LISTINGS

LISTE DES ANNEXES

LISTE DES SIGLES

API	Application Programming Interface
AST	Abstract Syntax Tree
DSL	Domain Specific Language
HTML	Hyper Text Markup Language
ORM	Object-relational mapping
XML	Extensible Markup Language

REMERCIEMENTS

Pour ce mémoire, je remercie mon directeur de recherche qui a été patient à mon égard et m'a permis d'étendre mes connaissances sur l'informatique et les compilateurs. Je remercie aussi ma famille qui m'a encouragé tout au long de mes études.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Les programmes s'infiltrent de plus en plus dans nos vies. Les bogues et surtout leurs conséquences peuvent devenir désastreuses. Par exemple, entre 1985 et 1987, la machine de radiothérapie *Therac-25* a délivré des doses fatales de radiation à au moins 6 patients à cause d'une erreur de programmation [4].

De telles erreurs sont coûteuses, il est peu probable qu'elles soient un jour complètement éradiquées cependant nous pouvons essayer de développer des systèmes réduisant la présence de bogues. C'est pourquoi les programmeurs sont toujours à la recherche de nouveaux moyens permettant d'écrire avec plus d'aise des programmes plus fiables.

1.2 Problématique

Typer est un langage ayant pour objectif de rendre les programmeurs plus productifs et de permettre l'écriture de programmes plus fiables.

Pour cela nous voulons donner la consistance de Lisp issue de la notation infixe à Typer. Effectivement, les appels d'expressions sont identiques, il n'existe pas d'inconsistance entre l'appel d'une fonction définie par l'utilisateur ou l'appel d'une fonction binaire telle que l'addition ou encore entre l'appel d'une macro. Cette consistance permet l'utilisateur d'écrire des programmes de façon générale et évite au programmeur d'écrire du code pour gérer des cas particuliers.

Nous voulons permettre aux programmeurs d'écrire ses propres extensions au

langage grâce à la méta programmation, plutôt que de fournir un large ensemble de fonctionnalités dont chaque utilisateur devrait se satisfaire.

Finalement, nous voulons offrir un système de type expressif permettant au programmeur d'écrire des programmes plus fiables. Notamment grâce à l'utilisation des types dépendants qui permet même à l'utilisateur de représenter des preuves mathématiques.

Il existe de nombreux langages de programmation mais aucun ne possédant toutes les fonctionnalités que nous désirons. C'est pourquoi nous avons décidé d'implémenter un nouveau langage.

Grâce à l'ensemble des outils génériques mis à la disposition des programmeurs, l'utilisateur peut écrire des extensions du langage, définir des DSL, créer des aides à la preuve, ou encore l'écriture d'outils d'analyse de programmes Typer, tout en gardant le cœur de Typer minimaliste.

Mon projet consiste en l'implémentation du langage expérimental Typer. Typer est un langage typé statiquement, possédant un puissant système de macros similaire à celui de Lisp, permettant à l'utilisateur de générer du code pendant la compilation. Le système de types que Typer offre est cohérent, permettant à celui-ci de pouvoir représenter des preuves mathématiques si l'utilisateur le souhaite.

Par rapport à un assistant à la preuve conventionnel tel que Coq, Typer est avant tout un langage de programmation. Effectivement, bien que Coq puisse être utilisé pour écrire des programmes, ceux-ci doivent passer à travers une étape appelée l'*extraction* qui traduit le code Coq en code Ocaml. Cette étape n'est pas complètement transparente au programmeur [41].

Typer a été construit pour permettre à l'utilisateur d'écrire des programmes. Le programmeur peut s'il le désire utiliser le système de types offert par Typer

pour écrire des preuves.

1.3 Contributions

Mon travail s'est concentré sur l'étape de *l'élaboration*. *L'élaboration* est le processus par lequel le compilateur transforme le code source écrit par le programmeur en complétant les éléments facultatifs que celui-ci n'a pas spécifiés (élidés), mais qui sont nécessaires lors de la vérification de types. Ainsi l'élaboration part d'un arbre purement syntaxique ou s-exp, fait l'analyse sémantique, et renvoie une représentation semblable au lambda-calcul typé.

L'élaboration est au cœur de Typer. Elle permet au code source d'être moins verbeux, plus concis et plus simple à comprendre.

Les transformations qui ont lieu pendant cette étape, incluent l'élimination des aides syntaxiques, la résolution des identificateurs, l'expansion des macros, la propagation et l'inférence des types. De plus, *l'élaboration* est la dernière phase qui peut signaler une erreur liée au code fourni, après cette phase, les erreurs sont internes au compilateur.

J'ai également implémenté un interpréteur, un utilitaire de débogage et de nombreux tests unitaires. L'effaçage de types a aussi été ajouté après l'élaboration comme optimisation primitive pour simplifier l'interprétation.

Typer s'inscrit dans un effort de réconcilier l'écriture de programmes et l'écriture de preuves [8]. Pour cela, Typer utilise les types dépendants, comme l'assistant à la preuve Coq, pour permettre à l'utilisateur de représenter des preuves s'il le souhaite, tout en offrant à l'utilisateur un langage orienté vers l'écriture de programmes.

Typer continue dans la lancée des langages Lisp, en offrant un langage homoiconique. Effectivement, le code Typer est une liste de S-exp , et manipuler du code

Typer revient à manipuler une liste. Le fonctionnement des macros est d'ailleurs basé sur cette observation : les macros génèrent du code Typer en manipulant une S-exp.

Typer suit une philosophie minimaliste similaire à celle de Scheme, ou un petit ensemble de fonctionnalités offrant à l'utilisateur un maximum de flexibilité est préféré à de nombreuses bibliothèques par défaut.

1.4 Plan

Dans la prochaine section nous présentons comment les abstractions permettent au programmeur de simplifier le code et d'éviter les redondances.

Par la suite nous présentons différents systèmes de méta programmation, en passant par les directives préprocesseur C, jusqu'aux méta classes en Python.

Puis, nous faisons une courte introduction sur les assistants à la preuve et comment ceux-ci peuvent être utilisés.

Nous arriverons à Typer et sa définition où nous présentons les différentes constructions que celui-ci permet ainsi que ses fonctionnalités.

Et enfin, nous entrerons plus en détails dans l'implémentation même de Typer, où nous nous concentrerons sur l'élaboration et l'évaluation, c'est à dire les parties auxquelles j'ai contribué le plus.

1.5 Abstraction en Programmation

Abstraction Une abstraction est un "être ou chose purement imaginaire" [24]. Les abstractions sont utilisées par le programmeur pour simplifier le code que celui-ci écrit. Les abstractions n'existent pas au niveau de la machine. Par exemple, un langage de programmation est une abstraction qui permet au programmeur d'écrire

un programme sans pour autant connaître l'assembleur.

Les abstractions sont au cœur des langages de programmation. Dans cette section, nous allons montrer comment le code source d'un langage est représenté par le compilateur, puis comment les abstractions permettent de simplifier cette représentation. Rendant le code moins redondant, et augmentant la modularité et la lisibilité du code.

Un programme peut être représenté comme un arbre. Par exemple, l'expression mathématique `4 + 2 y` est représentée comme l'arbre présenté à la Fig. 1.1. Les abstractions en programmation permettent au programmeur de décrire des arbres complexes tout en rendant le code source plus concis et plus lisible.

Un exemple d'une abstraction qui est présente dans la majorité des langages de programmation est la *fonction*. Les fonctions sont des sous-arbres représentant une opération ou des opérations paramétrées qui vont être utilisées à plusieurs endroits. Cela permet au programmeur d'éviter de recopier à plusieurs endroits le code de la fonction.

Par exemple, l'arbre de la Fig. 1.2 représente un programme qui exécute deux fois la même opération `op1` à des endroits différents. Sans les fonctions, le programmeur est obligé de dupliquer deux fois le code. Cela rend le programme verbeux et difficile à lire. De plus, si le processus `op1` est mis à jour, le programmeur devra

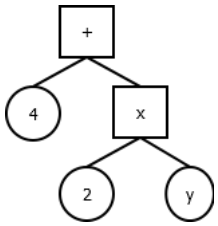

Figure 1.1 : Représentation de $4 + 2y$

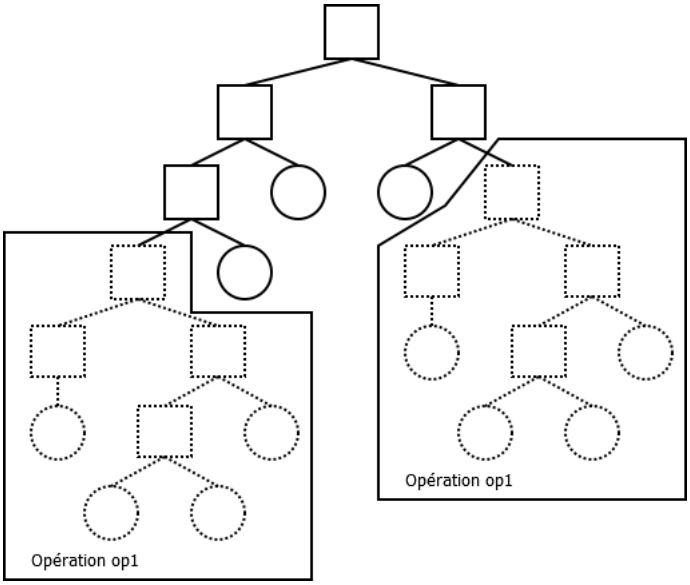

Figure 1.2 : Programme sans fonctions

modifier son code à deux endroits différents.

Grâce aux fonctions, le programmeur peut définir une fonction ex1 qui exécute l'opération op1. Il peut par la suite appeler la fonction à différents endroits à l'intérieur du programme. La Fig. 1.3 illustre le nouveau programme ainsi obtenu.

Un autre exemple d'abstraction est la *macro*. Les macros permettent au programmeur de créer des raccourcis qui vont générer du code pour nous. Ainsi, à la

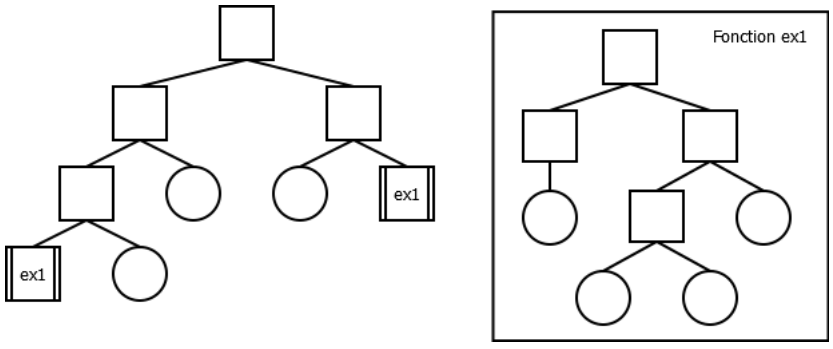

Figure 1.3 : Programme avec fonctions

différence des fonctions, l'arbre de syntaxe est simplifié seulement du coté du programmeur, une fois les macros étendues l'arbre de syntaxe résultant devrait être équivalent au code que le programmeur aurait écrit sans utiliser les macros.

Typer est un langage fonctionnel qui possède un système de types assez puissant pour qu'on puisse l'utiliser pour représenter des preuves formelles et un puissant système de macros. Avec une telle combinaison, nous pensons pouvoir offrir au programmeur une façon de créer de nouvelles abstractions pour augmenter sa productivité et lui permettre d'écrire des programmes plus fiables grâce au système de types de Typer.

Par exemple, le programmeur peut utiliser les différentes sortes d'arguments que Typer offre pour implémenter la notion d'argument par défaut. Un argument par défaut est un argument qui va prendre une valeur par défaut, si l'utilisateur ne le spécifie pas, lors de l'appel de la fonction.

Ainsi, le programmeur peut spécifier une fonction retournant une valeur par défaut pour un type quelconque, permettant ainsi au programmeur d'élider certains arguments lors d'un appel de fonction. Effectivement, le compilateur se chargera de remplir les arguments manquants.

L'exemple ci-dessous implémente, en Typer, une fonction qui retourne une valeur optionnelle. Si aucune valeur n'est contenue par `opt` la valeur par défaut du type `a` est retournée.

En premier, nous créons un attribut `default` pour le type `Int` qui va représenter la valeur par défaut à utiliser lorsqu'un argument est manquant. Dans notre exemple cette valeur est 0.

Par la suite nous déclarons une fonction prenant trois arguments, le type `a`, une valeur `d` de type `a` ainsi qu'une valeur optionnelle `opt` de type `a`. Le premier

argument est inféré par Typer, le deuxième argument va rechercher la valeur par défaut à utiliser si l'argument n'est pas spécifié et finalement, le troisième argument est un argument standard que le programmeur doit fournir.

Ainsi, la fonction maybe retourne la valeur de opt si celle-ci contient une valeur, sinon la valeur par défaut est retournée.

Nous entrerons plus en détails sur les différents types d'arguments dans les sections suivantes.

Listing 1.1 : Arguments par défaut (Typer)

```
default = add-attribute default Int (lambda args -> integer_ 0);

maybe : (a : Type) ≡> (d : a) => Option a -> a;
maybe = lambda a ≡>
  lambda d => lambda opt ->
    case opt
      | some val => val
      | none     => d;

val : Int;
val = maybe none; % retourne 0
```

CHAPITRE 2

LES SYSTÈMES DE MÉTA PROGRAMMATION

Un méta programme est un programme qui génère un autre programme. Nous allons nous pencher plus particulièrement sur la méta programmation c'est à dire une façon de générer du code pendant la compilation.

De nombreux systèmes de méta programmation sont basés sur les macros. Le mot *macro* a plusieurs significations. De façon générale, les macros sont des règles qui permettent de spécifier des séries de commandes à exécuter lors d'un événement spécifié par l'utilisateur. Par exemple, les actions d'une souris ou d'un clavier peuvent être enregistrées pour être par la suite re-exécutées. Cela permet à l'utilisateur d'automatiser des tâches répétitives.

En programmation, les macros permettent de spécifier un nom qui va par la suite être remplacé par un morceau de code spécifié par l'utilisateur.

Nous allons présenter différentes façons d'implémenter un système de méta programmation. Nous entrerons plus en détails dans les sections suivantes.

Il existe différentes approches pour créer des systèmes de méta programmation. Le préprocesseur C manipule une liste de lexèmes, le système est très simple, l'utilisateur peut remplacer une séquence de lexèmes par une autre, lui permettant ainsi de générer du code. Le système est limité par les manipulations que l'utilisateur peut effectuer sur la liste de lexèmes. Les préprocesseurs C ne supportent pas la récursion et aucun calcul ne peut être effectué à l'intérieur des macros. De plus, les macros sont déclarées de façon globale, il est donc important que l'utilisateur gère les déclarations des macros par lui même et évite de déclarer plusieurs fois des macros ayant des noms identiques.

Les macros des langages Lisp manipulent un arbre de syntaxe, à la différence des préprocesseurs C, n'importe quelle fonction Lisp peut être utilisée à l'intérieur des macros Lisp. Une macro est une fonction Lisp qui va manipuler un arbre de syntaxe puis le retourner. L'utilisateur doit retourner un arbre de syntaxe correct, sinon une erreur se produira après l'expansion des macros. Définir une macro en Lisp est similaire à définir une fonction, ainsi les macros peuvent être déclaréeslocalement si besoin.

Les templates C++ permettent à l'utilisateur de définir un patron de classe ou de fonction, qui va être utilisé pour générer du code. À la différence des systèmes précédents, la syntaxe des templates est vérifiée avant la génération puisque les templates sont aussi du code C++. Les templates peuvent être utilisés pour faire des calculs pendant la compilation. Ils peuvent être déclarés de façon locale. Une contrainte du système est, que tous les templates doivent être dans un fichier d'entête, pour que le compilateur puisse avoir accès à l'entierté du code template afin qu'il puisse générer le code.

2.1 Les usages de la méta programmation

Extension Syntaxique et ajout de nouvelles fonctionnalités

La méta programmation peut être utilisée pour ajouter de nouvelles fonctionnalités ou de nouvelles syntaxes au compilateur. Ces nouvelles fonctionnalités vont aider le programmeur à écrire du code plus compréhensible et éviter les redondances. L'exemple ci-dessous crée une nouvelle façon d'ajouter des attributs *read only* à une classe en C++.

Listing 2.1 : Extension syntaxique (C++)

```
1 #define ATTRIBUTE(type, name) \
```

```
2     private: type _##name; \
3     public: type name() const { return _##name; }
```

Par la suite nous allons voir comment certains systèmes de méta programmation permettent d'ajouter de nouvelles fonctionnalités, comme le pattern matching, au langage. [16]

Gain en généralité

Certains systèmes de méta programmation permettent au programmeur d'écrire un algorithme fonctionnant pour différents cas, tout en permettant la spécialisation de l'algorithme dans des cas spéciaux. Cela permet au programmeur d'éviter la duplication de code, diminuant le risque d'erreur et rendant le code plus facile à mettre à jour.

L'exemple ci-dessous montre l'utilisation de la fonction sort. La fonction peut être utilisée pour plusieurs structures de données différentes. Effectivement, le compilateur va générer une fonction sort pour chaque type. De la même façon, un nouveau type sera créé pour chaque vector<T> de type T, Ainsi, vector<int> et vector<string> sont deux types différents.

Le programmeur n'a donc pas besoin d'implémenter deux fois le même algorithme, ou la même structure de données pour différents types, il peut utiliser la méta programmation pour les générer automatiquement.

Listing 2.2 : Gain en généralité avec les templates (C++)

```
1 template<typename RandomIterator>
2 void sort(RandomIterator first, RandomIterator last){
3   ...
4 }
```

```
5
6  int main(){
7    vector<int> v;
8    array<int, 10> a;
9
10   ...
11
12   sort(begin(v), end(v));
13   sort(begin(a), end(a));
14 }
```

Optimisation

La méta programmation peut être utilisée pour faire des calculs coûteux en ressources pendant la compilation, réduisant ainsi le temps d'exécution du programme [39]. Par exemple, cela peut être utilisé pour créer des tables de hashage parfait [26].

Un autre exemple, qui est souvent utilisé dans les bibliothèques de calcul algébrique, est l'utilisation de la méta programmation pour optimiser les calculs.

Listing 2.3 : Optimisation grâce aux templates (C++)

```
1 Matrix m1, m2, m3;
2 Matrix r = m1 + m2 + m3;
```

Dans le code ci-dessus la multiplication des trois matrices entre-elles va générer 2 matrices temporaires (m' = m1 + m2 et m'' = m' + m3). Cependant, nous pouvons utiliser la méta programmation pour représenter les opérations de façon symbolique pendant la compilation pour, par la suite, générer du code plus efficace. Ainsi les matrices temporaires peuvent être éliminées rendant l'application finale plus

efficiente [29, p. 237].

Cette technique est appelée *expression template*. Les templates C++ sont utilisés pour définir un arbre d'expressions. L'arbre est évalué lorsque le résultat du calcul représenté est utilisé, le tout est fait pendant la compilation grâce au template de C++.

L'exemple ci-dessous, implémente une classe Expression qui va être utilisée comme base pour implémenter chaque nœud de notre arbre d'expressions. Par la suite, nous définissons l'opération d'addition à travers la classe Sum. Cette classe garde en mémoire les deux expressions à additionner sans effectuer l'opération.

Ainsi l'expression $m1 + m2 + m3$ créé un template imbriqué Sum<Sum<m1, m2>, m3> représentant les opérations à effectuer mais ne fait pas les calculs.

Listing 2.4 : Optimisation grâce aux templates (C++)

```
template<typename VecExpr>
class Expression{
  double eval(int i) const {
    return static_cast<VecExpr const&>(*this).eval(i);
  }
  int     size(int i) const {
    return static_cast<VecExpr const&>(*this).size(i);
  }
};

template <typename E1, typename E2>
class Sum : public Expression<Sum<E1, E2> > {
    E1 const& _u;
    E2 const& _v;
public:
    Sum(E1 const& u, E2 const& v) : _u(u), _v(v) {}
```

```
17      double eval(int i) const { return _u[i] + _v[i]; }
18      size_t size()      const { return _v.size(); }
19  };
20
21  template <typename E1, typename E2>
22  Sum<E1,E2> const operator+(E1 const& u, E2 const& v) {
23      return Sum<E1, E2>(u, v);
24  }
```

Nous avons besoin d'une classe représentant les valeurs même. Cette classe est implémentée ci-dessous. Les expressions symboliques sont évaluées lorsque le constructeur du vecteur est appelé.

Listing 2.5 : Optimisation grâce aux templates (C++)

```
1  class Vec : public Expression<Vec> {
2    std::vector<double> _data;
3  public:
4    double eval(int i) const { return _data[i]; }
5    int    size() const      { return _data.size(); }
6
7    template<typename E>
8    Vec(Expression<E> const& vec) : _data(vec.size()) {
9        for (size_t i = 0; i != vec.size(); ++i) {
10         _data[i] = vec.eval(i);
11       }
12   }
13 };
```

Ainsi `Vec r = m1 + m2 + m3;` est en réalité équivalent à `Vec r = Sum<Sum<m1, m2>, m3>`. Lorsque le constructeur est appelé, la fonction `eval` de la classe `Sum` est appelée à l'intérieur de la boucle, calculant pour chaque index du vecteur le résultat

de l'addition. Ainsi, aucun vecteur temporaire n'est effectivement créé.

L'exemple est tiré du site web suivant [15], il a été simplifié pour en faciliter la lecture.

2.2 Préprocesseur C

Les directives de préprocesseur C commencent par un # et finissent à la fin de la ligne. Plusieurs formes sont disponibles : #define, #if, #else #endif sont parmi les directives les plus utilisées. #define peut être utilisé pour déclarer un alias à une valeur. Par exemple, le code suivant #define MIN_VAL -12 va demander au compilateur de remplacer tous les identifiants MIN_VAL par la valeur équivalente -12.

Les define peuvent aussi prendre des arguments. Les arguments de la macro sont remplacés par leur valeur pendant l'expansion de macros.

L'exemple ci dessous définit une macro SQR qui possède un argument unique X. Lorsque l'utilisateur appelle la macro SQR(2), l'argument X est remplacé par sa valeur 2 puis SQR(2) est remplacé par 2 * 2.

Listing 2.6 : Macro C

```
1 #define SQR(X) ((X) * (X))
```

Les macros C opèrent sur des séquences de lexèmes. Par exemple, l'appel macro SQR(2) est composé de quatre lexèmes : SQR, (, 2,) qui seront remplacés pendant l'expansion de macro par les trois lexèmes suivants 2, *, 2 [18].

L'avantage d'un tel système est que celui-ci est très simple, cependant il n'est pas conscient du langage. Effectivement, la macro peut retourner du code C incorrect, dans ce cas le compilateur ne retournera une erreur qu'après l'expansion des macros. Ceci peut causer des messages d'erreur peu compréhensibles, rendant les macros difficiles à déboguer.

Le système de macros C n'étant pas conscient du langage, l'argument X peut être une liste de lexèmes quelconques.

Si une expression est fournie en argument, SQR(1 + 1), le code généré sera (1 + 1) * (1 + 1). L'addition sera ainsi calculée deux fois alors qu'un calcul peut être évité. De la même façon, si un appel de fonction avait été fourni à la macro, celle-ci aurait été appelée deux fois. Ainsi, dans le cas d'une fonction ayant des effets de bord, le comportement de la macro peut être différent du comportement attendu.

Le préprocesseur C peut aussi être utilisé pour gérer différentes architectures ou systèmes d'exploitation dans une même fonction, c'est ce qu'on appelle la compilation conditionnelle. Dans l'exemple ci-dessous, la fonction load_shared_library a une implémentation différente en fonction du système d'exploitation ciblé. Cela permet de fournir un API cohérent sur toutes les plateformes sans avoir à modifier le code de façon importante.

Listing 2.7 : Compilation conditionnelle (C)

```
1 void *load_shared_library(const char* lib_name){
2 #if defined (WIN32)
3   ... code lié à Windows
4 #elif defined (__linux__)
5   ... code lié à Linux
6 #endif
7 }
```

Les macros peuvent aussi être utilisées pour représenter des données qui vont être utilisées plusieurs fois dans différents contextes. Un exemple connu est la déclaration des statuts HTTP. Dans l'exemple ci-dessous nous déclarons une macro enregistrant chaque entrée dans une seconde macro 'X', cette macro sera définie par la suite et sera utilisée pour accéder aux éléments sauvegardés dans la macro

principale. Pour des raisons de simplicité, l'exemple est écrit en C++ cependant le système de macros entre les deux langages est identique.

Listing 2.8 : Les X-Macro (C)

```
#define MACRO_HTTP_STATUS(X) \
    X(Ok                , 200)\
    X(Created           , 201)\
    X(Accepted          , 202)

enum HTTPStatus{
  #define X(a, b) a = b,
    MACRO_HTTP_STATUS(X)
  #undef X
}

std::string to_string(HTTPStatus c){
  static std::unordered_map<HTTPStatus, std::string> keys = {
    #define X(a, b) {b, #a},
      MACROX_HTTP_STATUS(X)
    #undef X
  };
  return keys[c]
}
```

Un autre usage des macros C est l'extension syntaxique. L'exemple ci-dessous implémente un nouveau type de boucle, celle-ci permet d'itérer à travers une chaîne de caractères.

Listing 2.9 : Extension syntaxique (C)

```
#define FOR_EACH_CHAR(var_name, string)      \
  for (int i = 0, (var_name) = (string)[i], \
```

```
        n = strlen(string); i < n;              \
        ++i, (var_name) = (string)[i])

// Utilisation de la macro
int main()
{
  FOR_EACH_CHAR(c, "my string") {
     printf("%c \n", c);
  }
}
```

Le système de macros C est essentiellement basé sur la substitution textuelle.

2.2.1 OpenMP

OpenMP (Open Multi-Processing) est une extension des langages C/C++ et Fortran qui permet aux utilisateurs de créer des applications qui tirent profit du parallélisme. Pour paralléliser son code, l'utilisateur annote le code à exécuter en parallèle en utilisant le préprocesseur C [5]. Le compilateur va par la suite générer le code nécessaire pour que l'application utilise le parallélisme.

L'exemple ci-dessous utilise la construction *omp parallel* pour exécuter un morceau de code plusieurs fois dans plusieurs threads. Par exemple, si un processeur à deux cœurs est utilisé, Hello devrait être imprimé deux fois (il se peut que l'impression du mot soit entrecoupée puisque l'ordre d'impression de chaque lettre est indéterminé) [44].

Listing 2.10 : OpenMP parallel (C)

```
#pragma omp parallel
{
  printf("Hello!\n"); // body
```

```
}
```

L'exemple ci-dessus est transformé pendant la compilation pour permettre le parallélisme. Le code ci-dessous montre le résultat que le compilateur pourrait générer [25].

Listing 2.11 : OpenMP parallel (C)

```c
void GOMP_parallel_start (void (*fn)(void *), void *data, unsigned
    num_threads)

void subfunction (void *data)
{
  printf("Hello!\n"); // body
}

int main(){
  setup(data);
  GOMP_parallel_start (subfunction, &data, num_threads);
  subfunction (&data);
  GOMP_parallel_end ();
}
```

L'exemple ci-dessous utilise la construction *omp for* pour distribuer les itérations de la boucle sur plusieurs threads. Ainsi chaque thread exécutera une portion différente de la boucle [44].

Listing 2.12 : OpenMP for (C)

```c
#pragma omp for
for(int i = 0; i < 10; ++i)
{
        ...
```

```
}
```

Il faut noter que *omp for* redistribue le travail à travers les threads qui sont existantes. Au début de l'exécution du programme il n'existe que la thread principale, ainsi pour répartir l'exécution de la boucle sur plusieurs threads, il nous faut en premier créer un ensemble de threads. Pour ce faire nous pouvons combiner *omp parallel* et *omp for*, comme le montre l'exemple ci-dessous.

Listing 2.13 : OpenMP for (C)

```
#pragma omp parallel for
for(int i = 0; i < 10; ++i)
{
        ...
}
```

OpenMP possède de nombreuses autres directives que l'utilisateur peut utiliser pour répondre à ses besoins. Par exemple, il existe des directives de synchronisation, si celui-ci a besoin de partager des données à travers plusieurs threads [5].

OpenMP est une extension des langages C/C++ et Fortran, l'extension est implémentée en partie à travers une bibliothèque et en partie à l'intérieur même du compilateur. Effectivement, les directives que nous avons montrées sont utilisées par le programmeur pour annoter son code. Par la suite le compilateur va générer le code nécessaire pour transformer le code annoté afin que celui s'exécute en parallèle. Par exemple, OpenMP doit gérer la création des threads, le découpage des boucles et la création de mécanismes de synchronisation.

OpenMP permet ainsi au programmeur de transformer une application n'utilisant pas le parallélisme, seulement en ajoutant quelques annotations et sans avoir à faire de modifications majeures. De plus, l'extension OpenMP peut être désactivée

si nécessaire. L'utilisateur a donc un seul code pouvant générer deux programmes, un utilisant le parallélisme et un autre s'exécutant de façon séquentielle.

2.3 Les macros dans la famille de langages Lisp

Le système de macros des langages Lisp est très connu pour sa flexibilité. La façon dont les macros sont définies est très similaire à la façon dont les fonctions sont définies.

Avant d'expliquer plus en détails comment les macros des langages Lisp fonctionnent, il est important de comprendre la syntaxe de Lisp.

Pour cela, nous allons reprendre l'exemple présenté dans la section 1.5. Pour représenter l'expression `4 + 2 y` en Lisp, nous pouvons écrire `(+ 4 (* 2 y))`. Ce qui est interprété de la façon suivante : nous créons une liste avec trois éléments `+`, `4` et `(* 2 y)`, le troisième élément est une liste avec trois éléments `*`, `2` et `y`. Ainsi les arbres en Lisp sont créés par le biais de listes imbriquées [1].

Un exemple de code Lisp peut être trouvé ci-dessous. Celui-ci définit une fonction `diff-impl` qui possède deux arguments `lst` et `var`.

Listing 2.14 : Exemple de code Common Lisp

```
1 (defun diff-impl (lst var)
2   (let ((op (car lst)) (args (cdr lst)))
3     (cond
4       ((eq op var) 1)
5       ((eq op '+) (diff-add args var))
6       ((eq op '*) (diff-mult args var))
7       (T 0))))
```

Nous pouvons faire un parallèle entre le code Lisp ci-dessous, et la façon dont les listes sont créées en Lisp. Effectivement, nous pouvons remarquer que la fonction

écrite par le programmeur est aussi une liste ! De façon plus générale, un programme Lisp est une liste. Cette propriété du langage s'appelle l'homoiconicité **??**.

C'est une des raisons pour laquelle il est aussi facile de faire de la méta programmation en Lisp ; manipuler un programme revient à manipuler une simple liste ! De la même façon, générer du code Lisp revient à générer une liste !

Pour générer du code Lisp dans une macro, le plus simple est d'utiliser les fonctions quasiquote et unquote. quasiquote permet d'écrire le code Lisp qui sera généré verbatim, alors que unquote permet de spécifier une expression qui doit être évaluée à l'intérieur d'un quasiquote.

Par exemple le code (defmacro sqr (x) (quasiquote (* x x))) va générer le code (* x x), avec x étant une référence à une variable x. Bien sûr, ce code ne peut pas marcher si x n'a pas été défini par l'utilisateur.

Pour remplacer x par l'argument qui a été fourni à la macro, nous pouvons écrire à la place le code ci-dessous. quasiquote et unquote peuvent être abrégés par ‘ et , respectivement.

Listing 2.15 : Définition d'une macro en Lisp

```
1 (defmacro macro-sqr (x) `(* ,x ,x)) ;; Macro
2 (defun fonction-sqr (x) (* x x))    ;; Fonction
```

L'exemple présenté est équivalent à son homologue C. Nous allons voir par la suite que les deux systèmes opèrent à des niveaux différents. Les macros C opèrent sur des séquences de lexèmes, alors que les macros des langages Lisp opèrent sur des S-exp. Les S-exp sont les expressions de base qui définissent l'arbre de syntaxe de Lisp.

Lorsque l'appel à la macro sqr est reconnu, celle-ci est remplacée par le code spécifié. Un des avantages des macros Lisp est l'utilisation de son propre langage

comme langage de macros. Ainsi, écrire des macros Lisp est identique à écrire du code Lisp.

La macro présentée ci-dessus a le même problème que sa version C. Si une expression est passée en argument, celle-ci va être évaluée deux fois. Cependant, ce problème peut être résolu en Lisp, l'exemple ci-dessous traite ce problème.

Listing 2.16 : Macro en Lisp

```
1 (defmacro sqr (x) `(let ((y ,x)) (* y y)))
```

Un des facteurs limitant les macros C est le nombre d'actions possibles à l'intérieur d'une macro. Une macro C est représentée comme une liste de lexèmes. Les arguments sont remplacés par les valeurs fournies par l'utilisateur, puis la liste de lexèmes est insérée à l'endroit où la macro est appelée.

Cependant, les macros Lisp ont accès à l'entièreté du langage Lisp. Ainsi, les macros Lisp ont la possibilité de manipuler les arbres d'expression syntaxique (s-expression) avant d'être insérées dans le code. Effectivement, les macros Lisp sont des fonctions qui génèrent du code Lisp, n'importe quel code peut être exécuté à l'intérieur de la macro pour calculer l'expansion de celle-ci. L'exemple 2.19 est un bon exemple d'une macro qui est impossible à réaliser en C.

Le système de macros des langages Lisp offre de nouvelles façons au programmeur de générer du code automatiquement. C'est d'ailleurs pour ça que Lisp est souvent associé avec la méta-programmation. L'exemple ci-dessous implémente une boucle for.

Listing 2.17 : Extension syntaxique (Lisp)

```
1 ;; (for (elem :in list) body)
2 (defmacro for (init body)
3   (let ((item (car init))
```

```
4           (lst  (caddr init)))
5       `(mapcar (lambda (,item) ,body) ,lst)))
6
7 (for (elem :in '(1 2 3 4))
8   (print elem))
```

Les macros peuvent aussi être utilisées pour pré-calculer du code pendant la compilation afin de réduire les calculs nécessaires lors de l'évaluation (à la façon des expressions constantes (constexpr) en C++). Le programmeur peut donc construire des abstractions qui n'influencent pas la performance pendant l'exécution.

Listing 2.18 : Évaluation pendant l'expansion de macro (Lisp)

```
1 (defmacro sqr (x) (eval `(+ ,x ,x)))
```

La flexibilité des macros Lisp nous permet même d'implémenter de nouvelles fonctionnalités du langage sous forme de bibliothèques (qui combinent macros et fonctions). Ces fonctionnalités sont habituellement intégrées dans le compilateur. Cela permet au langage d'avoir un cœur minimaliste et d'implémenter des fonctionnalités dans le langage même, plutôt qu'à l'intérieur du compilateur [43]. Par exemple, le système de programmation orientée objet de Common-Lisp *CLOS* est implémenté de cette manière [17].

L'exemple ci-dessous implémente le pattern matching. La macro est utilisée de la façon suivante (match cible (pat1 exp1) ... (patn expn)). cible, pat1, exp1, patn, expn peuvent être n'importe quelle expression Lisp. L'utilisateur n'a pas à se soucier de savoir si match est une macro ou non.

Listing 2.19 : Pattern Matching (Lisp)

```
1 (defmacro match (sujet . clauses)
2   (let* (
```

```
3     (target (gensym))
4     (branches (mapcar (lambda (branch_x) (gensym)) clauses))
5     (err (gensym)))
6   `(let ((,target ,sujet))
7      ,@(mapcar
8           (lambda (b1 b2 clause)
9             `(defun ,b1 () (if (equal ,target ',(car clause))
10                 ,(cadr clause) (,b2))))
11         branches
12         (append (cdr branches) (list err))
13         clauses)
14     (defun ,err () (error "match failed"))
15   (,(car branches)))))
```

Le code suivant (match reponse ("oui" (print "cas oui")) ("non" (print "cas non"))) génère le code ci-dessous.

Listing 2.20 : Lisp Macro après expension

```
1 (LET ((#:G3210 REPONSE))
2  (DEFUN #:G3211 NIL (IF (EQUAL #:G3210 '"oui") (PRINT "cas oui") (#:
     G3212)))
3  (DEFUN #:G3212 NIL (IF (EQUAL #:G3210 '"non") (PRINT "cas non") (#:
     G3213)))
4  (DEFUN #:G3213 NIL (ERROR "match failed")) (#:G3211))
```

Les variables #:GN ont été générées lors de l'expansion de la macro par la fonction (gensym). Ces variables sont garanties de n'être utilisées nulle part ailleurs. Nous allons voir par la suite pourquoi il est important d'utiliser des variables uniques et quels problèmes peuvent se produire lorsque plusieurs variables partagent le même nom.

2.3.1 Scheme

Scheme est un dialecte du langage Lisp. Il a une philosophie minimaliste préférant ainsi promouvoir un petit standard mais permettant à l'utilisateur de programmer ses propres extensions au langage, notamment grâce aux macros.

Il existe plusieurs façons de définir des macros en Scheme, nous allons nous concentrer sur la définition de macros à travers define-syntax. define-syntax prend le nom de la macro en argument puis nous devons utiliser syntax-rules pour définir les motifs que la macro peut prendre [11]. Dans le cas de la macro sqr, un argument unique a est attendu.

Listing 2.21 : Scheme SQR macro

```
(define-syntax sqr
  (syntax-rules ()
    ((_ a) (* a a))))
```

Scheme a une importante qualité, ses macros sont hygiéniques. Voyons le problème des macros standards d'un peu plus près. Dans le code suivant (defmacro fun3 (a b c) `(fun2 ,a (fun2 ,b ,c))), nous n'avons aucune garantie que la fonction fun2 n'a pas été redéfinie plus tôt.

Listing 2.22 : Problème d'hygiène (Lisp)

```
(defun fun2 (x y) (+ x y))
(defmacro fun3 (a b) `(fun2 ,a ,b))

(let ((rez1 (fun3 1 2))) rez1)

(defun fun2 (x y) (* x y))

(let ((rez2 (fun3 1 3))) rez2)
```

Effectivement, une instance du problème peut être trouvée ci-dessous. Les variables rez1 et rez2 vont avoir deux valeurs différentes, alors que le programmeur pourrait s'attendre à avoir des valeurs identiques. Un tel bogue peut être difficile à repérer. Cependant, Scheme résout le problème pour nous, ainsi la macro équivalente en Scheme retourne deux valeurs identiques.

Listing 2.23 : Problème d'hygiène (Scheme)

```
1 (define-syntax fun3
2   (syntax-rules ()
3       ((_ a b) (fun2 a b))))
```

2.4 Les méta classes en Python

Les macros que nous avons vues jusqu'à présent manipulent une représentation d'un programme, que cela soit une liste d'expressions syntaxiques en Lisp ou une liste de lexèmes en C. La représentation utilisée était purement textuelle et les informations contextuelles liées au programme n'étaient pas disponibles. Par exemple, nous n'avons pas l'information liée au typage des expressions [30, p.117-133].

Python offre un système de méta classes qui nous permet de résoudre un tel problème. OpenJava nous offre un système similaire [40] permettant de créer et de modifier les classes. Nous avons décidé de nous concentrer sur les méta classes Python en raison de leur documentation plus complète, ainsi que de leur utilisation dans plusieurs projets d'envergure (SQLAlchemy, Flask, Django...).

Toutes les entités en Python sont des objets ayant des attributs. Ainsi une fonction Python telle que décrite ci-dessous possède plus de 34 attributs. Les attributs incluent : la documentation qui a été écrite par l'utilisateur, les annotations de type, et même le bytecode de la fonction compilée.

Listing 2.24 : Fonction en Python (Python)

```
def sqr(x : int) -> int:
  """retourne le carré de x"""
  return x ** 2
```

L'utilisateur a la capacité de modifier chaque attribut de l'objet *sqr*, incluant le bytecode [38], il peut aussi ajouter de nouveaux attributs.

De la même manière l'utilisateur peut créer et manipuler les classes Python, puisque les classes elles-mêmes sont des objets. Les classes sont créées par la fonction type. La fonction type prend trois arguments : le nom de la classe à créer, les classes dont elle hérite, ainsi qu'un dictionnaire d'attributs. Par exemple `A = type('A', (), )` crée une classe vide. Le code ci-dessous crée une classe `B` qui hérite de A et possède deux attributs, l'attribut `v` ayant pour valeur `42` et l'attribut `sqr` étant une fonction.

Listing 2.25 : Création d'une classe (Python)

```
B = type('B', (A,), dict(v=42, sqr=sqr))
```

Les méta classes sont des objets qui initialisent une classe, lui donnant ses attributs et ses méthodes. À la manière dont les constructeurs d'une classe sont appelés pour créer un objet, les constructeurs d'une méta classe sont appelés pour créer une classe [19].

Pour créer une méta classe en Python, il suffit de créer une classe héritant de la classe type.

Listing 2.26 : Création d'une méta classe (Python)

```
class Singleton(type):
  instance = None
  def __call__(cls, *args, **kw):
    if not cls.instance:
```

```
5            cls.instance = super(Singleton, cls).__call__(*args, **kw)
6        return cls.instance
```

L'exemple ci-dessus implémente une méta classe qui représente un *Singleton*. Un Singleton est une classe qui ne peut avoir qu'une seule instance. Notre méta classe implémente l'opérateur __call__ qui sera appelé lorsque un objet du type *cls* sera créé. La fonction vérifie l'existence d'une instance, si celle-ci existe elle est retournée, sinon le constructeur de la classe *cls* est appelé pour créer l'instance.

Pour créer une classe utilisant le motif du singleton, il nous suffit de spécifier la méta classe Singleton lorsque la classe est déclarée, comme le montre l'exemple ci-dessous. Ainsi les objets a et b sont exactement les mêmes. Il est impossible à l'utilisateur de créer plusieurs objets de type singleton. L'implémentation d'un singleton dans d'autres langages est souvent faite de façon ad hoc, l'utilisation de variables globales peut être requis. Cependant, Python offre une façon élégante d'implémenter un tel motif.

Listing 2.27 : Création d'un Singleton (Python)

```
1 class ResourceManager(metaclass=Singleton):
2   def __init__(self, ...):
3     ...
4
5 a = ResourceManager();
6 b = ResourceManager();
```

Les méta classes sont utilisées pour modéliser des abstractions de haut niveau en donnant la possibilité au programmeur de contrôler la façon dont les classes sont créées, ou encore pour générer du code à l'intérieur de la classe en lui ajoutant de nouvelles fonctions ou attributs.

2.5 Les *templates* C++

Un problème, que peut avoir un langage typé statiquement tel que C, est la définition de structures de données générales qui peuvent marcher pour n'importe quel type. De telles structures peuvent être implémentées, cependant elles manquent souvent de sûreté.

À travers les *templates* le programmeur peut définir une structure de données pour un type qui sera spécifié par la suite. L'exemple ci-dessous montre l'implémentation d'un *Point* qui est une structure de données contenant deux valeurs de même type *T*. Si le programmeur a besoin d'un point contenant deux nombres décimaux, il peut le créer en écrivant `Point<float>`.

Listing 2.28 : Point (C++)

```
1 template<typename T>
2 struct Point{
3   T x;
4   T y;
5 }
```

Le développeur n'a pas besoin d'implémenter différentes structures de données pour différents types et peut réutiliser beaucoup plus de codes grâce aux *templates*.

D'une certaine façon les *templates* peuvent être vus comme des macros, qui vont générer le code nécessaire lorsque celui-ci est utilisé. Elles permettent aussi au compilateur de pouvoir vérifier le typage de façon plus approfondie. Effectivement, dans un langage comme C, la seule façon d'écrire une structure de données aussi générique serait de déclarer les variables `x` et `y` comme étant des pointeurs `void*` ce qui empêche le compilateur de vérifier les types.

De plus, cette structure de données utilise des allocations dynamiques. Une

autre solution serait de déclarer un Point par type mais ceci n'est pas une solution générale et oblige le programmeur à dupliquer du code.

Listing 2.29 : Point (C)

```
struct Point{
  void* x;
  void* y;
}
```

Les *templates* peuvent aussi être utilisés pour faire des calculs, bien que ceux-ci n'aient pas été créés pour. Effectivement, ceux-ci sont *Turing complete*, cependant le compilateur fixe une limite au nombre de récursions qu'un template peut faire.

L'exemple ci-dessous montre comment la fonction factorielle pourrait être calculée pendant la compilation grâce aux templates [42].

Listing 2.30 : Calculs avec les templates (C++)

```
template <unsigned int n>
struct factorial {
  enum { value = n * factorial<n - 1>::value };
};

template <>
struct factorial<0> {
  enum { value = 1 };
};
```

La bibliothèque du standard c++ (STL) utilise abondamment les templates. Ceux-ci permettent de créer de nouvelles abstractions avec peu ou pas d'impact nocif à la performance, tout en simplifiant et augmentant la cohérence de la bibliothèque.

Par exemple, la STL implémente la notion d'*iterator*. Les *iterators* sont des abstractions qui permettent de traverser une structure de données quelconques. Ainsi toutes les structures de données peuvent être lues et manipulées de façon similaire permettant de réutiliser les mêmes algorithmes pour différentes structures de données.

Listing 2.31 : Déroulement des boucles (C++)

```
template<int i, typename Arg>
struct loopct{
  static void run(std::function<void(int, Arg)> action, Arg arg) {
     action(i, arg);
     loopct<i - 1, Arg>::run(action, arg);
  }
};

template<typename Arg>
struct loopct<-1, Arg>{
  static void run(std::function<void(int, Arg)>, Arg) {}
};
```

Nous pouvons utiliser les templates pour générer du code pendant la compilation. L'exemple ci-dessus va recopier l'appel à la fonction `action` `i` fois. L'utilisateur peut donc implémenter lui même des utilitaires déroulant les boucles, permettant à celui-ci de forcer l'optimisation de son code sans se reposer sur le compilateur.

Une autre façon d'augmenter la généralité du code est d'encapsuler toutes les informations spécifiques à un type à l'intérieur d'une classe.

Listing 2.32 : Déclarations d'une classe d'informations (C++)

```
template<typename T>
struct TypeInfo
```

```
3 {
4   typdef T value;
5
6   static std::string& name() {  return ""; }
7   static T add(const T& a, const T& b) { return a + b; }
8 };
9
10 template<>
11 struct TypeInfo<int>{
12   static std::string& name() {  return "int"; }
13 }
14
15 template<>
16 struct TypeInfo<string>{
17   static std::string& name() {  return "string"; }
18   static T add(const T& a, const T& b) { return a.append(b); }
19 }
```

L'exemple ci-dessus implémente une classe TypeInfo qui va sauvegarder un ensemble d'informations pour un type T. Lorsque les valeurs par défaut ne sont pas suffisantes ou que l'utilisateur désire ajouter le support pour un nouveau type, il suffit de spécialiser la classe TypeInfo comme ce qui a été fait, ci-dessus, pour les types int et string.

Par la suite nous pouvons utiliser TypeInfo pour écrire notre algorithme comme le montre le code ci-dessous. L'exemple que nous avons présenté est simpliste puisqu'il existe une façon plus courte d'arriver au même résultat. Cependant, cette approche est très utile lorsque un système complexe est construit et que la classe TypeInfo va être réutilisée de nombreuses fois. La bibliothèque Eigen utilise cette approche [13] pour implémenter des matrices pouvant même fonctionner sur des types définis par

l'utilisateur.

Listing 2.33 : Utilisation des classes d'informations (C++)

```
template<typename T>
void print(T a){
  cout << TypeInfo<T>::name() << ": " << a << std::endl;
}
```

Les classes du type TypeInfo sont utiles pour créer une façon uniforme d'accéder à des informations liées aux types, limitant ainsi les insertions de codes spécifiques à certains types. Effectivement, tout le code spécifique est regroupé à l'intérieur de la classe TypeInfo , permettant ainsi aux fonctions d'être les plus génériques possibles.

Nous pouvons noter ici que cette utilisation des templates est particulière au C++ et que les génériques de Java ne peuvent pas être utilisés de cette manière. Effectivement, les attributs statiques en Java sont partagés par toutes les classes, peu importe le type T , alors que C++ va générer de nouveaux attributs statiques pour chaque type T [28].

Les templates C++ opèrent au niveau de l'*arbre de syntaxe abstrait* (AST), effectivement ils permettent au programmeur d'ajouter des arguments à des expressions (fonctions ou objets). Ces arguments sont déduits lorsqu'ils sont utilisés. Cela oblige le programmeur à coder la logique de son méta programme à l'intérieur de nombreux objets C++ , ce qui peut rendre les templates C++ difficiles à déboguer.

2.6 Haskell

Template Haskell (TH) est un système qui permet d'écrire des meta programmes qui sont evalués durant la compilation, ceux-ci produisent un programme Haskell.

L'exemple ci-dessous génère une fonction Haskell qui permet de transformer une fonction prenant un tuple de taille ***n*** en argument, en une fonction prenant ***n*** arguments. La meta fonction curryN prend la taille ***n*** du tuple et retourne une expression de type ***Exp***, qui représente la fonction Haskell *currynn*. Par exemple, curryN 2 retourne une expression représentant la fonction curry2

Listing 2.34 : CurryN (Haskell)

```haskell
curryN :: Int -> Q Exp
curryN n = do
  f  <- newName "f"
  xs <- replicateM n (newName "x")
  let args = map VarP (f:xs)
      ntup = TupE (map VarE xs)
  return $ LamE args (AppE (VarE f) ntup)
```

Nous pouvons noter que curryN retourne une expression monadique Q Exp. Pour accéder au résultat Exp du calcul, nous pouvons utiliser l'opération *splice* ou *$* pour remplacer l'appel de la meta fonction par l'expression générée. Bien sûr, pour que l'opération *splice* puisse marcher, il faut que le résultat retourné par la monade soit un programme correct.

Template Haskell utilise un arbre de syntaxe abstrait pour représenter les programmes qu'il génère. Toutes les constructions syntaxiques du langage Haskell sont disponibles à travers leur constructeur. Par exemple, LamE est le constructeur d'une lambda expression ou encore AppE le constructeur d'un appel de fonction. Ces constructeurs peuvent être utilisés pour créer de nouveaux programmes. Ainsi le langage Haskell peut être utilisé comme langage de méta-programmation. Template Haskell opère au niveau de l'AST, De plus, celui-ci n'est pas limité au langage Haskell, effectivement TH peut être utilisé pour créer de nouveaux langages, ou intégrer

des langages existants à Haskell, en définissant un parseur pour ce langage. Par exemple, la bibliothèque *shakespeare* permet à l'utilisateur d'écrire du code HTML à l'intérieur de Haskell même.

2.7 Programmation par Aspect

Nous avons vu dans l'introduction que les programmeurs découpent leurs programmes en modules réutilisables pour augmenter la lisibilité du programme et diminuer la duplication de code. Pour ce faire, le programmeur va découper son programme en plusieurs sous-modules logiques. Chaque sous-module implémentant une fonctionnalité du programme final.

Par exemple, dans une bibliothèque mesurant la rapidité d'exécution d'un programme, un utilitaire *Horloge* mesurant le temps, et un autre utilitaire gardant les différents temps d'exécution peuvent être implémentés séparant le code des deux mécaniques.

Cependant, il est possible que la nature du paradigme à implémenter soit impossible à séparer des autres fonctionnalités.

Par exemple, lors de l'implémentation d'un journal (*log*) enregistrant les appels de fonctions exécutées. Il nous faut ajouter un appel enregistrant l'information requise par le log à chaque fonction. Ainsi, nous nous retrouvons avec des fonctions utilisant un entremêlement de fonctionnalités rendant le code moins lisible et plus dur à mettre à jour.

Listing 2.35 : Exemple d'entremêlement

```
1 void fun1(....){
2   log("fun1 was called");
3
```

```
4    // Implémentation de fun1
5
6    ....
7  }
```

Les aspects peuvent être utilisés pour résoudre ce problème. L'exemple ci-dessous créé un aspect *Logging* qui définit pour chaque fonction un *advice* , qui spécifie comment ajouter des entrées dans le journal en fonction de la fonction qui est appelée. Ici *module.fun1* et *module.fun2* sont les deux *advice* implémentés dans l'aspect *Logging*.

Listing 2.36 : Exemple d'un Aspect

```
1  void fun1(....){
2    // Implémentation de fun1
3    ....
4  }
5
6  aspect Logging {
7    void module.fun1(...)  {
8      log("fun1 was called");
9    }
10
11   void module.fun2(...){
12     ...
13   }
14 }
```

La programmation par aspect nous permet donc d'éviter que différentes fonctionnalités s'entremêlent, allégeant le code [37]. Cependant, la programmation par aspect rend le code plus difficile à comprendre, puisque nous ne pouvons pas

connaître l'entierté des actions qui sont effectuées lorsqu'une fonction est appelée, seulement en regardant le code de la dite fonction. Effectivement, il nous faut aussi regarder les aspects qui ont été définis pour cette fonction. Ainsi, les aspects séparent les fonctionnalités pour une plus grande modularité mais au coût d'obscurcir la séquence d'exécution du programme [22].

Python nous offre une manière similaire de résoudre le problème d'entremêlement des fonctionnalités à travers les décorateurs de fonction.

Listing 2.37 : Les décorateurs de fonctions (Python)

```
1  def log_it(fun):
2    """Décorateur ajoutant l'appel de la fonction fun à notre log"""
3
4    def wrapper(*args, **kwargs):
5      log("function: " + fun.__name__ + ", starts")
6      r = fun(*args, **kwargs)
7      log("function: " + fun.__name__ + ", ended")
8      return r
9
10   return wrapper
11
12 @log_it
13 def fun1(arg1, arg2):
14   pass
15
16 a = fun1(1, 2)
```

L'exemple ci-dessus implémente le logging des appels de fonction à travers un décorateur. Les décorateurs enveloppent (*wrap*) notre fonction `fun1` à l'intérieur d'une nouvelle fonction *wrapper* qui va pouvoir se charger d'ajouter l'appel de

notre fonction fun1 à notre log. De plus, les décorateurs de fonctions peuvent aussi être utilisés pour vérifier la validité des arguments passés à la fonction.

À la différence des Aspects, nous savons que la fonction fun1 utilise un décorateur grâce à l'annotation @log_it. Les décorateurs nous offrent aussi une plus grande flexibilité puisque nous pouvons choisir lorsque la fonction fun1 est appelée.

Par exemple, la bibliothèque *Numba* nous permet de compiler des fonctions Python en code machine, les rendant plus rapide à exécuter. Pour utiliser *Numba*, il suffit à l'utilisateur d'annoter les fonctions qu'il désire voir compiler avec le décorateur @jit. Le décorateur se charge de compiler la fonction, puis de l'exécuter. La fonction originale Python n'est jamais exécutée [23].

CHAPITRE 3

LES ASSISTANTS À LA PREUVE

Les assistants à la preuve permettent de vérifier et d'aider à écrire des preuves formelles. Ils sont souvent utilisés pour prouver la validité d'un programme, dont la fiabilité ou la sécurité est critique, tels que les schémas de cryptographie [2]. Il existe plusieurs assistants à la preuve Coq, F*, HOL ou encore Isabelle . Nous allons nous concentrer sur Coq puisque c'est l'assistant à la preuve le plus répandu et possédant une documentation riche.

Coq permet à l'utilisateur de spécifier des définitions, des axiomes, ou encore une proposition à prouver. À partir des théorèmes connus, l'assistant détermine les sous-propositions manquantes pour terminer la preuve. Cependant, il en revient à l'utilisateur de compléter la preuve.

L'utilisateur peut aussi utiliser des sous-routines pré-programmées pour essayer de trouver une preuve, ou plus précisément une partie de preuve, puisque Coq requiert souvent des indications de la part du programmeur pour prouver le théorème. De telles routines sont appelées *tactiques*.

3.1 Représentation d'une preuve

Pour représenter les preuves mathématiques à l'intérieur d'un langage de programmation, nous utilisons l'isomorphisme de Curry-Howard. Celui-ci permet de lier ensemble la théorie des types et la logique. Ainsi, une proposition correspond à un type, l'implication en logique correspond au type d'une fonction, et la conjonction correspond au produit de deux types. En appliquant ces principes, la preuve

d'une proposition P est le terme t de type P [34].

Par exemple la preuve de $P \supset Q$ peut être vue comme une fonction calculant une preuve de Q à partir d'une preuve de P. La fonction a donc un type $P \rightarrow Q$ [34]. Le système de typage de Coq est puissant et permet au programmeur de représenter des propositions complexes.

Ainsi pour prouver qu'un théorème est vrai, l'utilisateur doit construire une fonction représentant la preuve.

Pour écrire une preuve en Coq, le programmeur commence par définir le lemme qu'il désire prouver comme le fait l'exemple ci-dessous.

Listing 3.1 : Definition d'un Lemme (Coq)

```
1 Lemma expr: forall x y, (x + y) * (x + y) = x*x + 2*x*y + y*y.
```

Écrire une preuve à la main, à travers un assistant à la preuve, peut être fastidieux. C'est pourquoi Coq offre une sorte de système de Macro appelé *Ltac* qui va nous permettre de générer notre preuve de façon plus aisée.

Par exemple, la tactique intros x y construit le terme fun x y => _ représentant la preuve que nous allons construire.

Pour continuer notre preuve, nous allons rechercher des théorèmes connus par Coq, pour transformer successivement notre expression et pour finalement trouver le résultat à prouver.

La recherche est faite à travers la fonction SearchRewrite. La fonction prend en argument un motif qui va être utilisé pour la recherche, par exemple (_ * (_ + _)) est un motif, le symbole _ est utilisé pour représenter une expression quelconque.

Nous voulons maintenant distribuer le terme $(x+y)*(x+y)$. Pour cela nous allons rechercher le théorème de la distributivité. Les lignes commençant par In >> sont les lignes écrites par l'utilisateur, alors que les lignes commençant par Out >>

sont les réponses de l'assistant.

Listing 3.2 : Distributivités (Coq)

```
In  >> SearchRewrite (_ * (_ + _)).
Out >> mult_plus_distr_l: forall n m p : nat, n * (m + p) = n * m + n
    * p
```

La commande affiche les théorèmes correspondant au terme de recherche. Dans notre cas un seul théorème a été trouvé. Nous pouvons appliquer le théorème à travers la commande rewrite.

Listing 3.3 : Application (Coq)

```
In  >> rewrite mult_plus_distr_l.
```

L'expression résultante est $(x+y)*x+(x+y)*y$. De la même façon, le théorème de la distributivité à droite peut être trouvé et appliqué. L'expression que nous avons alors est $x*x+yx+xy+y*y$.

Nous voulons transformer $x*y$ en $y*x$ pour pouvoir les combiner ensemble par la suite. Pour cela nous allons avoir besoin de la règle de commutativité de la multiplication.

Listing 3.4 : Commutativité (Coq)

```
In  >> SearchPattern (?x * ?y = ?y * ?x).
Out >> mult_comm: forall n m : nat, n * m = m * n
In  >> rewrite mult_comm with (n:= y) (m:=x).
```

Finalement, il nous suffit d'additionner les deux membres restant pour finaliser la preuve. Pour cela, nous devons faire apparaître un 1, comme ceci $x*x+1*y*x+x*y+y*y$ et utiliser la règle de l'addition de l'opération multiplication.

Listing 3.5 : Addition (Coq)

```
In  >> SearchRewrite (S _ * _).
Out >> mult_1_l: forall n : nat, 1 * n = n
    >> mult_succ_l: forall n m : nat, S n * m = n * m + m

In  >> pattern (x * y) at 1; rewrite <- mult_1_l.
In  >> rewrite <- mult_succ_l.
In  >> Qed.
```

L'expression finale obtenue est $x*x+2*y*x+y*y$, nous avons donc réussi à prouver notre lemme. Nous pouvons noter que Coq représente les nombres entiers en utilisant un type inductif nat, ainsi le nombre 2 est en réalité vu comme S (S 0), où S n représente le nombre après n. Coq convertit automatiquement d'une notation à l'autre. Maintenant que notre lemme a été prouvé, celui-ci peut être utilisé pour prouver de nouvelles propositions. La preuve ci-dessus peut aussi être faite de façon non-interactive. Cette preuve est présentée dans le code ci-dessous. La preuve, ou fonction résultante peut être trouvée en annexe I.1.

Listing 3.6 : Preuve non-interactive (Coq)

```
intros x y.
rewrite mult_plus_distr_l.
rewrite mult_plus_distr_r.
rewrite mult_plus_distr_r.
rewrite plus_assoc.
rewrite <- plus_assoc with (n := x * x).
rewrite mult_comm with (n := y) (m := x).
pattern (x * y) at 1; rewrite <- mult_1_l.
rewrite <- mult_succ_l.
rewrite mult_assoc.
reflexivity.
Qed.
```

La ligne rewrite <- plus_assoc with (n := x * x). est utilisée pour modifier le sens de la réécriture mais n'a aucun effet sur l'expression en elle-même.

Cet exemple essaye de démontrer pourquoi les preuves par ordinateur requièrent autant d'interactions de la part des utilisateurs, ainsi que l'attrait de Ltac qui permet de réduire le travail fourni par l'utilisateur.

Il existe des tactiques plus avancées qui permettent de générer des preuves de façon quasi automatique (en fonction du type de proposition à prouver). Par exemple, dans notre cas nous aurions pu utiliser la tactique ring [6]. L'exemple ci-dessous montre l'utilisation de cette tactique. La preuve résultante peut être trouvée dans l'annexe I.2.

Listing 3.7 : Utilisation d'une tactiques (Coq)

```
1 In  >> Require Import Arith.
2     >> Require Import Ring.
3     >> Lemma expr:
4     >>     forall x y, (x + y) * (x + y) = x * x + 2 * x * y + y * y.
5     >> intros; ring.
6     >> Qed.
```

Les tactiques peuvent être vues comme des macros générant le code nécessaire afin de prouver une proposition. La commande Show Proof. peut être utilisée pour afficher la preuve qui a été générée par la tactique que nous avons utilisée.

Les tactiques ont leur propre langage appelé *Ltac*. Il permet aux utilisateurs d'écrire leur propre tactique. *Ltac* a sa propre syntaxe et a été spécifiquement créé pour permettre à ses utilisateurs de construire des preuves [41].

Par exemple, la tactique présentée ci-dessous, essaye d'appliquer la tactique f

sur chaque élément de la liste ls (tuple imbriqué) et retourne le premier appel qui réussit [10].

Listing 3.8 : Création d'une tactique (Coq)

```
1 Ltac app f ls :=
2   match ls with
3     | (?LS, ?X) => f X || app f LS || fail 1
4     | _ => f ls
5   end.
```

Nous avons vu comment les macros peuvent être utilisées pour générer du code, ou dans le cas de Coq, générer des preuves. Par la suite nous allons voir comment Typer unifie la notion de macro et de tactique en créant un langage unique à la façon de Lisp pour écrire des programmes.

CHAPITRE 4

LE LANGAGE TYPER

Typer est un langage de programmation fonctionnel, il combine un assistant à la preuve et un système de macros puissant. Typer suit la philosophie minimaliste de Scheme. À la manière de Coq, Typer utilise un système de type dépendant. Le système est appelé dépendant car le *type* d'une *valeur* peut dépendre d'une autre *valeur*.

L'exemple 4.1 montre le type d'une fonction créant un tableau de taille fixe à l'exécution. Le paramètre ***n*** représentant la taille du vecteur est une valeur dans le type du résultat. Ceci nous permet d'avoir un système de type particulièrement expressif, ce qui nous permet d'augmenter le nombre de bogues trouvés pendant la compilation, et de réduire le nombre de bogues pendant l'exécution du programme. Cependant, il revient au programmeur d'exprimer ses invariants de manière à ce que le système de type puisse les vérifier.

Listing 4.1 : Exemple de type dépendant (Typer)

```
1 make_vector : (t : Type) ⇒ (x : t) -> (n : Nat) -> Vector t n;
```

Si nous reprenons l'exemple ci-dessus, la taille du tableau est incluse dans le type du tableau. Cela va nous permettre de détecter s'il existe des problèmes d'accès pendant la compilation et non pendant l'exécution, lorsque le programme essayera de lire une zone mémoire dont il n'a pas l'accès.

De plus, le système de typage de Typer a été fait pour être capable de représenter des propositions de façon similaire à ce qu'un assistant à la preuve fait. Par la suite, nous allons voir plus en détails ce que nous espérons pouvoir accomplir grâce à un

tel système.

4.1 Motivation

Grâce aux types dépendants, Typer offre un système de types particulièrement expressif, permettant de représenter plus d'informations lors de la vérification de types. L'utilisateur peut même représenter des preuves formelles qu'il peut utiliser par la suite pour générer du code sur mesure.

Par exemple, le code du Listing 4.2 peut être grandement simplifié puisque nous savons que la variable `i` est toujours inférieure à la taille du tableau et supérieure ou égale à 0, ainsi la vérification de l'indice n'est donc pas utile. Nous voulons permettre au programmeur de faire cette optimisation à la main tout en gardant les assurances garanties par le système de types.

C'est à dire que le programmeur ne pourra faire l'optimisation seulement s'il existe une preuve que l'optimisation est sûre.

Ce genre d'optimisation est déjà réalisée dans de nombreux compilateurs (dont GCC et clang pour le langage C). Cependant, une telle optimisation est complexe et la réussite de l'analyse automatique faite par le compilateur dépend entièrement de la sophistication de l'analyse. Une analyse plus sophistiquée rend plus difficile la prédiction de son échec par le programmeur.

Listing 4.2 : Exemple

```
1 def get_item(array, idx):
2   if len(array) > idx => 0:
3     return array[idx]
4   raise Error("index out of bounds")
5
6 s = 0
7 for i in range(len(array)):
8   s = s + get_item(array, i)
```

Ainsi, en permettant au programmeur de remplacer l'analyse automatique, celui-ci peut s'assurer que l'optimisation va bel et bien avoir lieu à l'endroit où la performance est la plus critique.

4.2 Aperçu

La syntaxe de Typer est inspirée de ML et Lisp. À la manière de Lisp, le code Typer est lu comme une liste d'expressions symboliques (S-exp), puisque notre système de typage est statique, nous avons aussi la possibilité d'annoter le type de chaque variable.

Typer permet au programmeur d'utiliser certains opérateurs infixes tel que $+$, $-$ ou encore $*$, ceux-ci sont convertis par le compilateur en notation préfixe. Ainsi `a + b` est équivalent à `_+_ a b`. De la même façon, `case a | p1 => e1 | p2 => e2` est équivalent à `case_ (_|_ (_=>_ p1 e1) (_=>_ p2 e2))`. Le programmeur peut utiliser les deux notations [35].

La syntaxe est divisée en deux catégories ; les expressions et les déclarations. Un fichier contient une séquence de déclarations. Les déclarations sont utilisées pour lier une expression à une variable. Il existe plusieurs types d'expressions. Les *Datatypes* permettent de créer de nouveaux types et des constructeurs, les *Fonctions* permettent à l'utilisateur de créer et d'appeler ses propres fonctions, les *Let* sont utilisés pour créer de nouvelles déclarations à l'intérieur d'une portée, les *Var* permettent de faire une référence à une déclaration, et finalement, les constantes qui représentent des valeurs fournies par l'utilisateur.

4.3 Déclarations

$$
\begin{aligned}
\text{(declaration) } \mathit{decl} ::= {} & \mathit{id} : \mathit{exp} \\
| {} & \mathit{id} = \mathit{exp} \\
| {} & \mathit{decl} \ ; \ \mathit{decl} \\
| {} & \mathit{decl\text{-}macro\text{-}call}
\end{aligned}
$$

Il existe trois types de déclarations. L'annotation de type id : exp, le binding id = exp qui associe un nom à une expression et l'appel à une macro de déclaration exp qui sera étendue en annotations et/ou bindings pendant l'élaboration. L'élaboration est le processus qui, à travers le compilateur, transforme le code Typer écrit par le programmeur en complétant les éléments que celui-ci à omis, l'élaboration inclut l'expansion des macros. Cette étape sera vue en détails dans le prochain chapitre.

Listing 4.3 : Déclarations (Typer)

```
1 a : Int; % Annotation
2 a = 2;   % Binding
3
4 % Appel à la macro (type_)
5 type Nat
6     | zero
7     | succ Nat;
```

4.4 let

$$\text{(expressions)}\ exp ::= \cdots$$
$$\mid \text{let}\ decl\ \text{in}\ exp$$

Le let nous permet de déclarer des variables locales (decls) qui peuvent être utilisées dans leurs propres définitions ainsi qu'à l'intérieur de l'expression finale exp (le let de Typer est récursif).

Listing 4.4 : Let (Typer)

```
let a : Int;
    a = 2;
    b = 3
in a + b
```

En réalité le let ci-dessus est compris par le compilateur de la façon suivante let a = 2 in let b = 3 in a + b. Seulement les définitions récursives sont groupées à l'intérieur du même let. Les annotations de types sont en grande partie optionnelles, nous avons ajouté l'annotation sur la variable a pour montrer leur utilisation, cependant celle-ci n'est pas obligatoire, effectivement l'inférence de type peut déduire le type de a. L'inférence est vue plus en détails dans le chapitre suivant.

Pour évaluer un Let, les déclarations du let sont évaluées en premier, puis elles sont substituées à l'intérieur de l'expression finale du Let. Les règles sont présentées

$$\frac{}{\text{let}\ x = v\ \text{in}\ e \ \leadsto\ e[x \mapsto \mu\ x\ .\ v]} \qquad \frac{}{\mu\ x\ .\ e \ \leadsto\ e[x \mapsto \mu\ x\ .\ e]}$$

Figure 4.1 : Règle d'évaluation du Let

à la Fig. 4.1. Nous utilisons la notation $x[v \mapsto e]$ pour représenter la substitution de la variable x par la valeur v dans l'expression e

La règle de typage du Let est délicate. La règle naïve est présentée ci-dessous. Elle montre que le type d'un let est le type de son expression finale.

Γ représente notre contexte de typage, c'est une séquence contenant les variables et leurs types. $\Gamma \vdash e : t$ peut être lu comme l'expression e à un type t dans le contexte Γ. De nouvelles expressions peuvent être ajoutées au contexte en utilisant l'opérateur , par exemple $\Gamma, x : T$ représente de l'ajout de la variable x de type T au contexte Γ [34, p. 101].

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}$$

Cependant, Typer a un typage dépendant : le type t_2 de l'expression finale peut dépendre des déclarations présentes à l'intérieur du Let. La règle prenant en compte le typage dépendant est énoncée ci-dessous. Nous utilisons la notation $e[x \to v]$ pour représenter la substitution de x par v dans e.

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2[x \mapsto e_1]}$$

Cependant, le Let de Typer est récursif, donc en réalité, la règle ressemble plutôt à :

$$\frac{\Gamma, x{:}t_1 \vdash e_1 : t_1 \quad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2[x \mapsto \mu\ x\ .\ e_1]}$$

Les règles de typage sont délicates, la réalité est encore plus délicate puisqu'il nous faut gérer les définitions mutuellement récursives.

4.5 Arrow

$$
\begin{array}{lll}
(\textit{Arrow}) & \textit{arw} & ::= \equiv > \\
 & & \mid => \\
 & & \mid -> \\
(\text{expressions}) & \textit{exp} & ::= \cdots \\
 & & \mid (\textit{id} : \textit{exp})\ \textit{arw}\ \textit{exp} \\
 & & \mid \textit{exp}\ \textit{arw}\ \textit{exp}
\end{array}
$$

Les flèches sont utilisées pour représenter le type des fonctions. Par exemple le type Int -> Float est le type d'une fonction prenant un argument de type entier et retournant une valeur de type nombre flottant. Le type d'un argument peut avoir un nom comme ceci (a : Int) -> Float. Le nom peut ainsi être utilisé à l'intérieur du type, par exemple ((a : Type) => List a).

Dans Typer, il existe trois types d'arguments. Les arguments *erasable* ($\equiv>$), les arguments *implicites* ($\Rightarrow$) et les arguments *normaux* ($\rightarrow$).

4.5.1 Les arguments normaux

Les arguments normaux sont les arguments que le programmeur est habitué à utiliser. Si un argument normal est omis une application partielle a lieu.

Listing 4.5 : Argument Normaux (Typer)

```
1 mult : Int -> Int -> Int;
2 mult = lambda (x : Int) ->
3          lambda (y : Int) -> x * y;
4
5 % Application Partielle
```

```
6  twice : Int -> Int;
7  twice = mult 2;
8
9  tena = mult 2 5;
10 tenb = twice 5;
```

4.5.2 Les arguments implicites

Les arguments implicites sont des arguments qui peuvent être omis par le programmeur. Ces arguments vont être déduits pendant la compilation. Il est toujours possible à l'utilisateur de les spécifier. L'exemple ci-dessous retourne le premier d'une liste d'entiers, ou une valeur par défaut si la liste est vide. La valeur par défaut d'un entier peut être déduite par le compilateur et n'a donc pas besoin d'être spécifiée par le programmeur. Si celui-ci souhaite utiliser une autre valeur que celle déduite par le compilateur, celui-ci peut spécifier l'argument implicite.

La valeur par défaut d'un type est contenu dans l'attribut default. La première ligne de l'exemple ci-dessous ajoute la valeur par défaut du type entier, ici 0.

Listing 4.6 : Argument Implicite (Typer)

```
1  default = add-attribute default Int (lambda args -> integer_ 0);
2
3  hd-int : Int -> List Int -> Int;
4  hd-int = lambda (default : Int) =>
5    lambda (list : List Int) -> (
6      case list
7        | cons val _ => val
8        | nil        => default) : Int;
9
10 list = cons 2 (cons 1 nil);
```

```
11
12 two  = hd-int list; % retourne 2
13 zero = hd-int nil;  % retourne la valeur par défaut
14
15 one  = hd-int (default := 1) nil; % Spécifie d'utiliser "1"
16                                    % comme valeur par défaut
```

4.5.3 Les arguments *erasable*

Les arguments *erasable* sont un cas particulier des arguments implicites. Effectivement, les arguments erasable ne sont pas nécessaires lors de l'exécution et sont enlevés une fois que la vérification des types a été faite.

Pour qu'un argument soit erasable il faut que l'argument soit seulement utilisé dans les annotations de types, ou que la valeur de l'argument ne soit pas utilisée pendant l'évaluation. L'exemple ci-dessous implémente la fonction `hd` qui retourne le premier élément d'une liste de types quelconque `t`. L'argument `t` est seulement utilisé dans les annotations de types. Il n'est donc pas nécessaire à l'évaluation et sera supprimé après la vérification des types.

Listing 4.7 : Argument Erasable (Typer)

```
1 hd : (t : Type) ≡> List t -> Option t;
2
3 hd = lambda (t : Type) ≡>
4     lambda (list : List t) -> (
5       case list
6         | cons val _ => some val
7         | nil        => none) : Option t;
```

4.6 Lambda

$$
\begin{aligned}
(\textit{lambda})\ \ \textit{exp}\ ::= \cdots \\
&\mid \text{lambda}\ (\textit{id} : \textit{exp})\ \textit{arw}\ \textit{exp} \\
&\mid \text{lambda}\ \textit{id}\ \textit{arw}\ \textit{exp}
\end{aligned}
$$

L'exemple ci-dessous présente la façon de créer un fonction de façon générale en Typer.

Listing 4.8 : Fonction sans sucre syntaxique (Typer)

```
1 mult = lambda (x : Int) -> lambda (y : Int)  -> x * y;
```

Il existe plusieurs autres façons de déclarer des fonctions. Nous pouvons annoter la fonction avec son type, cela va nous permettre d'écrire la fonction sans spécifier le type de chaque argument.

Listing 4.9 : Fonction avec annotations(Typer)

```
1 mult : (x : Int) -> (y : Int) -> Int;
2 mult = lambda x -> lambda y  -> x * y;
```

L'utilisateur n'a en fait pas besoin de spécifier le nom des arguments dans les annotations puisque le compilateur est capable de les retrouver par la suite. Ainsi nous pouvons écrire

Listing 4.10 : Fonction (Typer)

```
1 mult : Int -> Int -> Int;
2 mult = lambda x -> lambda y  -> x * y;
```

Écrire un lambda pour chaque argument peut être fastidieux. L'utilisateur peut grouper plusieurs arguments sous le même lambda.

Listing 4.11 : Fonction avec arguments groupés (Typer)

```
1 mult : Int -> Int -> Int;
2 mult = lambda x y -> x * y;
```

Finalement, le lambda peut être complètement supprimé. Cela offre au programmeur la possibilité d'écrire des fonctions de façon simple et concise.

Listing 4.12 : Fonction sans lambda (Typer)

```
1 mult : Int -> Int -> Int;
2 mult x y = x * y;
```

Pour créer des arguments implicites, ou erasable il nous suffit de modifier l'annotation de types, modifiant ainsi le type d'argument utilisé.

Dans certains cas, les annotations de types peuvent être omises grâce à l'inférence de types. La fonction lambda x -> e est alors comprise comme lambda (x : ?) -> e, le ? représente une méta variable, celle-ci va être utilisée pour déduire le type de x.

La régle de typage d'une lambda est très explicite. Une lambda ayant un argument x de type t_1 et un corps e_2 de type t_2 a pour type $t_1 \to t_2$. Le type de flèche représente le type d'argument utilisé.

$$\frac{\Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{lambda } (x : t_1) \text{ arw } e_2 : (x : t_1) \text{ arw } t_2}$$

4.7 Les appels de fonctions

$$
\begin{aligned}
(\mathit{call}) \quad \mathit{exp} ::= & \cdots \\
& \mid \mathit{exp}\ \mathit{exp} \\
& \mid \mathit{exp}\ (\mathit{id} : \mathit{exp})
\end{aligned}
$$

Pour évaluer l'appel de fonction nous devons récupérer la définition de la fonction, puis évaluer les arguments. Finalement, nous pouvons substituer les arguments par leur valeur à l'intérieur du corps de la fonction.

$$\frac{e_1 \ \rightsquigarrow \ e_1'}{e_1\ e_2 \ \rightsquigarrow \ e_1' e_2}$$

$$\frac{e_3 \ \rightsquigarrow \ e_3'}{v\ e_3 \ \rightsquigarrow \ v\ e_3'}$$

$$\frac{}{(\text{lambda}\ x\ \mathit{arw}\ e_2)\ v \ \rightsquigarrow \ e_2[x \mapsto v]}$$

Le type d'un appel de fonction est le type du corps de la fonction.

$$\frac{\Gamma \vdash e_1 : t_2 \ \text{arw}\ t_3 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1\ e_2) : t_3[x \mapsto e_1]}$$

4.8 Annotation de type

$$
\begin{aligned}
(\mathit{has\text{-}type}) \quad \mathit{exp} ::= & \cdots \\
& \mid \mathit{exp} : \mathit{exp}
\end{aligned}
$$

Le programmeur peut annoter une expression à l'intérieur même d'une expression. Cela peut être utile pour forcer la vérification de types et empêcher ou aider l'inférence. L'exemple ci-dessous force le compilateur à vérifier le type de la variable `a` qui est de type `Float` alors qu'une variable de type `Int` était attendue. Les deux types étant différents le compilateur va signaler une erreur de type.

Listing 4.13 : Annotation de type (Typer)

```
1 a = 2.0;
2 b = (a : Int);
```

4.9 Type Inductif

$$
\begin{array}{lll}
\textit{colon} & ::= & ::: \\
 & | & :: \\
 & | & : \\
\textit{formal-arg} & ::= & \textit{id} \\
 & | & (\textit{id colon exp}) \\
\textit{field-type} & ::= & \textit{exp} \\
 & | & (\textit{id colon exp}) \\
(\textit{datatype}) \quad \textit{exp} & ::= & \cdots \\
 & | & \text{type } \textit{id formal-arg}* \ (|\ \textit{id field-type}*)*
\end{array}
$$

La création de types inductifs permet à l'utilisateur de créer ses propres structures de données. Les types inductifs peuvent être utilisés pour créer des unions ou des produits de types.

Listing 4.14 : Déclaration de types inductifs (Typer)

```
% Declaration d'un type inductif (union `cons` ou `nil`)
List : Type;
type List (t : Type)
  | cons t (List t)
  | nil;

% Produit de deux Int
type Point
  | point Int Int;
```

Il est intéressant de noter que la syntaxe ci-dessus utilise une macro type. Cette macro ajoute une nouvelle façon de déclarer des types inductifs, elle a été entièrement implémentée en Typer. Le code généré est le suivant :

Listing 4.15 : Macro *type* étendue (Typer)

```
List = typecons (List (t : Type)) (cons t (List t)) nil;
cons = datacons List cons;
nil  = datacons List nil;
```

4.10 Les constructeurs

datacons nous permet d'accéder aux constructeurs d'un type inductif. Celui-ci devrait être rarement vu par le programmeur, puisque son utilisation est cachée à travers la macro type dont nous avons parlé précédemment.

$$\frac{\Gamma, it = \text{typecons } .. \ (l \ t_1 \ ... \ t_n) \ ..}{\Gamma \vdash (\text{datacons } it \ l) : t_1 \to ... \to t_n \to t_0}$$

Par exemple, le type du constructeur point est $\text{Int} \to \text{Int} \to \text{Point}$ ou encore le type du constructeur cons est $(t : Type) \equiv> t \to \text{List } t \to \text{List } t$.

4.11 Le *case*

$$
\begin{aligned}
(\textit{pattern-arg}) \quad & ::= \textit{id} \\
& \quad \mid (\textit{id} \ := \ \textit{id}) \\
(\textit{pattern}) \quad & ::= \textit{id} \ \textit{pattern-arg}* \\
(\textit{case}) \quad \textit{exp} & ::= \cdots \\
& \quad \mid \text{case } \textit{exp} \ (\text{"|"} \ \textit{pattern} \Rightarrow \textit{exp})*
\end{aligned}
$$

Les case peuvent être utilisés de différentes façons. Ils nous permettent d'extraire les données stockées à l'intérieur des constructeurs d'un type inductif. Nous pouvons aussi les utiliser pour exécuter un code différent en fonction du constructeur ciblé. Les case fonctionnent uniquement pour les constructeurs. De plus, les *pattern* ne peuvent pas être imbriqués.

Listing 4.16 : Utilisation d'un Case (Typer)

```
1 % Utilisation d'un case
2 to-num : Nat -> Int;
3 to-num x = case x
4   | (succ y) => (1 + (to-num y))
5   |  zero    =>  0;
6
7 % Extraction de valeurs
8 get-x : Point -> Int;
9 get-x p = case p | point x y -> x;
```

L'évaluation du case est délicate car chaque branche possède son propre environnent. En premier la cible du case est résolue, celle-ci doit être un constructeur. Par la suite, le compilateur recherche la branche correspondante au constructeur

cible.

Une fois la branche trouvée, on peut maintenant substituer les variables de l'expression finale par les variables présentes dans le constructeur. Un nouvel environnent est créé. Les valeurs contenues par le constructeur sont ajoutées à cet environnement. L'expression finale peut maintenant être évaluée. Pour cela il suffit de substituer les variables

$$\frac{e_1 \rightsquigarrow e'_1}{case\ e_1|... \rightsquigarrow case\ e'_1...}$$

$$\begin{array}{lll} (case & (cons\ t_0\ label_i)\ v_0\ ...\ v_j & \\ & |(cons\ t_0\ label_0)\ x_{00}\ ...\ x_{0k} => e_0 & \rightsquigarrow\ \forall\ h \in [0; j]\ e_i[x_{ih} \mapsto v_h] \\ & |(cons\ t_0\ label_n)\ x_{m0}\ ...\ x_{ml} => e_m) & \end{array}$$

Toutes les branches d'un case doivent retourner une expression d'un même type. De plus, tous les motifs et la cible du case doivent être un constructeur appartenant au type inductif t_0.

$$\frac{\Gamma \vdash e_0 : t_0 \quad \forall i\ \Gamma,\ x_{i0}\ ...\ x_{ik} \vdash e_i : t_3}{\Gamma \vdash \begin{array}{ll} (case & (cons\ t_0\ label_i)\ v_0\ ...\ v_j \\ & |(cons\ t_0\ label_0)\ x_{00}\ ...\ x_{0k} => e_0 \quad : t_3 \\ & |(cons\ t_0\ label_n)\ x_{m0}\ ...\ x_{ml} => e_m) \end{array}}$$

4.12 Les macros

À la manière de Lisp, les macros Typer sont de simples fonctions qui prennent en argument une liste de S-exp et retournent une S-exp qui sera acceptée par l'interpréteur typer. C'est la responsabilité du programmeur de créer une macro qui

retourne une S-exp sémantiquement correcte, si celle-ci est incorrecte une erreur est retournée après l'expansion, lorsque la S-exp résultante est lue par le compilateur. L'expression retournée par la macro a un type attendu, qui est vérifié après l'expansion. La notation est lourde et le compilateur Typer ne fournit pas de sucre syntaxique. Effectivement, Typer suit une philosophie minimaliste, nous voulons que de telles aides syntaxiques soient écrites en Typer même.

Listing 4.17 : Déclaration de Macros (Typer)

```
sqr = macro (lambda (x : List Sexp) ->
  let hd = case x
    | cons hd _ => hd
    | _ => (symbol_ "erreur") in
      (node_ (symbol_ "_*_") (cons hd (cons hd nil))));

val = (sqr 2);
```

Pour créer une macro, il suffit de définir une fonction Typer ayant comme signature (List Sexp) -> Sexp. La fonction est par la suite donnée au constructeur de macro Typer macro. La valeur résultante a un type Macro et représente notre macro qui peut être appelée de façon similaire à une fonction habituelle.

L'argument d'une macro est une simple liste de S-exp. Ainsi les arguments de la macro n'ont pas besoin d'être du code Typer valide, contrairement à Template Haskell, cela permet aux programmeurs d'ajouter de nouvelles syntaxes avec aise. Typer définit 6 S-exp de base, 5 sont utilisées pour représenter la syntaxe de Typer. La S-exp Block est particulière puisqu'elle permet au programmeur d'écrire autre chose que du code Typer.

Listing 4.18 : S-exp en Typer (ML)

```
type pretoken =
```

```
  | Pretoken of string
  | Prestring of string
  | Preblock of pretoken list

type sexp =
  | Block of pretoken list
  | Symbol of string
  | String of string
  | Integer of integer
  | Float of float
  | Node of sexp * sexp list
```

Par exemple, nous pourrions définir la fonction xml-reader {<html><body></body></html>}. On peut voir que l'argument n'est pas du code Typer valide, cependant, xml-reader étant une macro, l'argument va être conservé en S-exp. Ceci nous permet d'intégrer d'autres langages à Typer.

Les macros peuvent aussi être utilisées pour définir de nouvelles syntaxes de déclarations.

La macro ci-dessous définit une nouvelle façon de déclarer une fonction. La macro est utilisée de la façon suivante defun nom-fonction arguments corps et sera transformée après l'élaboration en nom-fonction arguments = corps.

Le code de la macro type peut aussi être trouvé en annexe II.

Listing 4.19 : Macro-déclarations (Typer)

```
defun = macro  (lambda (x : List Sexp) ->

  let err = (symbol_ "error") in
  let make-decl : Sexp -> List Sexp -> Sexp;
      make-decl op body = node_ (symbol_ "_=_") (cons op body)
```

```
6
7   in case x
8     | nil => err
9     | cons name args+body => (
10      case args+body
11        | nil => err
12        | cons args body =>
13          (make-decl (node_ name (cons args nil)) body)));
14
15 sqr : Int -> Int;
16 defun sqr x (x * x);
```

4.13 Features

4.13.1 Monads

Typer est un langage fonctionnel pur. Les effets de bord sont tout de même accessibles à travers le système de monads. Les monads encapsulent les opérations impératives et nous permettent de les utiliser dans du code Typer sans compromettre le typage stricte de celui-ci. L'intérêt est d'interdire aux programmeurs d'utiliser des valeurs issues d'opérations non pures à l'intérieur du système de typage. Si de telles opérations étaient autorisées, les garanties offerte par le système de typage seraient grandement compromises.

Listing 4.20 : Exemple d'inconsistance de typage (Typer)

```
1 make_vector : (t : Type) ⇒ (x : t) -> (n : nat) -> Vector t n;
2
3 % Créé un vecteur de taille 5
4 vector5 = make_vector (t : Int) x 5
5
```

```
6 vector-stdin1 = make_vector (t : Int) x (read-nat stdin);
7 vector-stdin2 = make_vector (t : Int) x (read-nat stdin);
```

L'exemple ci-dessus montre un problème de typage lié aux calculs impurs. Dans cet exemple, le type du vecteur devrait être Vector Int (read-nat stdin), Cependant, est-ce que vector-stdin1 et vector-stdin2 ont vraiment le même type ? Nous n'avons aucune garantie que l'utilisateur va fournir deux fois le même nombre. Les deux vecteurs n'ont pas forcément la même taille. Ainsi, permettre des opérations impures réduit grandement les garanties que notre système de typage peut offrir.

4.13.2 Réarrangement des arguments

Le programmeur peut fournir les arguments d'une fonction dans n'importe quel ordre sous condition que le nom des arguments soit fourni. Plusieurs exemples peuvent être trouvés ci-dessous. L'appel résultant peut être trouvé en commentaire.

Listing 4.21 : Exemple du réarrangement des arguments (Typer)

```
1 fun = lambda (x : Int) =>
2   lambda (y : Int) ->
3     lambda (z : Int) -> x * y + z;
4
5 a = fun (x := 3) 2 1;                  % fun (x := 3) 2 1;
6 b = fun (x := 3) (z := 1) 4;           % fun (x := 3) 4 1
7 c = fun (z := 3) (y := 2) (x := 1); % fun (x := 1) 2 3
```

4.13.3 Arguments par défaut

Nous avons vu dans la section 4.2 que Typer supportait plusieurs types d'arguments. Le programmeur peut aussi ajouter des attributs aux variables Typer. Les

attributs sont des expressions Typer qui sont associées à une variable. L'exemple ci-dessous permet d'associer aux variables Typer une chaîne de caractères documentant la variable.

Listing 4.22 : Exemple d'attribut (Typer)

```
sqr x = x * x;

doc-string = new-attribute String;
doc-string = add-attribute doc-string sqr "Compute x^2";

sqr-doc = get-attribute doc-string sqr;
```

Typer utilise les attributs pour permettre aux utilisateurs de spécifier la valeur par défaut à utiliser lorsqu'un argument implicite n'est pas spécifié. L'exemple ci-dessous implémente la fonction `inc` qui ajoute par défaut 1 à l'entier fourni par l'utilisateur. L'appel `inc 2` est transformé par le compilateur en `inc (a := (get-attribute default Int)) 2` après l'expansion des macros, le code généré est `int (a := 1) 2`.

Listing 4.23 : Exemple de l'attribut *default* (Typer)

```
inc : Int => Int -> Int;
inc = lambda a =>
        lambda b -> a + b;

default = new-attribute Macro;
default = add-attribute default Int (lambda args -> integer_ 1);

a = inc 2;
```

4.13.4 Argument *erasable*

Les arguments *erasable* en combinaison avec les arguments formels des types inductifs permettent au programmeur d'implémenter des structures de données et algorithmes généraux à la manière des templates en C++.

Par exemple, Typer implémente une liste contenant une valeur de type a de la façon décrite ci-dessous.

Listing 4.24 : Implémentation d'une liste (Typer)

```
List : Type;
type List (a : Type)
  | nil
  | cons a (List a);

length : (a : Type) ≡> List a -> Int;
length = lambda a ≡>
  lambda xs ->
    case xs
      | nil => 0
      | cons hd tl => (1 + (length tl));
```

Les preuves

Pour permettre à Typer de représenter des preuves, il nous suffit d'avoir des types dépendants, puisque nous pouvons utiliser des types pour représenter des preuves, grâce à l'isomorphisme de Curry-Howard que nous avons vu à la section 3.1. Le système de typage se charge d'assurer la justesse des preuves, ou des types.

Typer ne possède pas encore les nombreuses tactiques présentes dans Coq pour permettre aux utilisateurs d'écrire des preuves de façon rapide. Cependant, il nous

est tout de même possible d'écrire de simples preuves. Typer offre la fonction Eq qui retourne la preuve que deux expressions de type t sont égales. Cette fonction peut être utilisée pour écrire des preuves plus complexes.

Listing 4.25 : Utilisation de preuve en Typer

```
type Reify (a : Type)
  | RInt (Eq (t := Type) a Int)
  | RFloat (Eq (t := Type) a Float)
  | RString (Eq (t := Type) a String);

to-int : (a : Type) > (r : (Reify a)) => (x : a) -> Int;
to-int = lambda a > lambda r => lambda x ->
  case r
    | RInt    p => Eq_cast (p := p) (f := (lambda x -> x)) x
    | RFloat  p => float-to-int (Eq_cast (p := p) (f := (lambda x -> x
   )) x)
    | RString p => string-to-int (Eq_cast (p := p) (f := (lambda x ->
   x)) x);
```

L'exemple ci-dessus implémente une sorte de surcharge des opérations, en permettant à la fonction to-int d'effectuer différentes opérations en fonction du type de l'argument x. Pour faire cela nous avons créé un type inductif représentant différents types. Chaque constructeur contient la preuve que le type a est bien du type que le constructeur représente. Par exemple, RInt contient la preuve que le type a est de type Int. Par la suite nous pouvons construire la fonction to-int en utilisant un case pour assigner l'opération à effectuer pour chaque cas.

Nous pouvons noter ici l'utilisation de la fonction Eq_cast. Cette fonction est utilisée pour convertir une variable d'un type a en une variable de type b. La fonction prend en argument la preuve que x est de type b et la fonction convertissant x en

type cible.

L'utilisation de cette fonction est nécessaire puisque l'argument x est de type a , cependant les fonctions float-to-int et string-to-int ne s'attendent pas à un tel type. Par exemple, lorsque la fonction float-to-int est appelée, x doit être de type float pour satisfaire le système de typage. Nous utilisons donc Eq_cast et la preuve contenue dans les constructeurs pour transformer le type de l'argument x.

CHAPITRE 5

L'IMPLÉMENTATION DE TYPER

Typer est un langage expérimental. Dans cette première implémentation nous utilisons le langage *Ocaml*. Notre implémentation se concentre sur les fonctionnalités de base du langage avec un soucis mineur sur les performances. Effectivement, nous voulons utiliser cette première implémentation pour pouvoir commencer à programmer le compilateur Typer en Typer même, rendant ainsi le langage *self-hosting*.

5.1 Design

Typer est avant tout un langage expérimental, c'est pourquoi nous avons décidé de découper le compilateur en différents modules bien séparés au détriment de la performance. Nous en retirerons une plus grande flexibilité lorsque nous allons faire évoluer le langage. Afin de suivre cette logique, Un fichier source de Typer passe à travers 6 étapes différentes qui sont énumérées ci-dessous.

- File → Pretoken
- Pretoken → S-exp
- S-exp → Pexp
- Pexp → Lexp
- Lexp → Elexp
- Elexp → Vexp

Le fichier est découpé en pré-token. Cette étape nous permet de supprimer les espaces blancs et séparer les blocs (le code écrit entre accolades '{' '}') du reste du

code Typer. Le code à l'intérieur des blocs n'est pas obligatoirement du code Typer, cela devrait permettre aux programmeurs d'implémenter de nombreuses extensions de Typer à l'intérieur de Typer même.

De façon similaire à Lisp, le code Typer peut être vu comme une S-exp (Symoblic Expression). Les S-exp sont les composantes de base de notre langage. La transformation de pré-token à S-exp expression se fait en deux étapes. Premièrement, les pre-token sont convertis en token par le lexer puis les token sont combinés entre eux en suivant une table de précédence [20, p. 187-194].

Un échantillon de la table de précédence peut être aperçu ci-dessous.

Symbole	Gauche	Droite
/	140	153
*	141	154
−	109	128
+	110	129
	...	
:	78	78
lambda	None	117
case	None	42
(	None	3

La transformation en Pexp (pre - expression) correspond à l'analyse syntaxique du code. C'est ici que nous identifions les constructions de Typer (lambda, case, let, etc...) et que nous vérifions que la syntaxe a été respectée.

L'étape la plus importante est l'élaboration, c'est à dire la transformation de Pexp à Lexp (lambda expression). C'est sur cette partie que la majorité de mon travail s'est concentré. La prochaine section est entièrement dédiée à cette étape.

L'avant-dernière étape (de lexp à elexp (erased lexp)) efface l'information de typage qui n'est pas utile lors de l'évaluation. Finalement, La dernière étape correspond à l'évaluation de l'expression par l'interpréteur.

5.2 L'élaboration

L'élaboration est au cœur de Typer. Elle permet au code source d'être moins verbeux, plus concis et plus simple à comprendre.

L'élaboration est le processus qui, à travers le compilateur, transforme le code source écrit par le programmeur en complétant les éléments que celui-ci a élidés, construisant ainsi une représentation explicite du programme [36].

Les types Pexp et Lexp sont décrits ci-dessous pour illustrer le travail de l'élaboration. Nous pouvons remarquer que Lexp possède de nombreux champs additionnels. Ce sont ces champs qui devront être déterminés pendant l'élaboration. Effectivement, les pexp représentent un arbre purement syntaxique.

Listing 5.1 : Pexp (ML)

```
type pexp =
  | Pimm of sexp
  | Pbuiltin of symbol
  | Pvar of pvar
  | Phastype of pexp * pexp
  | Pmetavar of pvar
  | Plet of pdecl list * pexp
  | Parrow of arg_kind * pvar option * pexp * pexp
  | Plambda of arg_kind * pvar * pexp option * pexp
  | Pcall of pexp * sexp list
  | Pcase of pexp * (ppat * pexp) list
```

Listing 5.2 : Lexp (ML)

```
type ltype = lexp
 and subst = lexp S.subst
 and lexp =
   | Imm of sexp
   | SortLevel of sort_level
   | Sort of sort
   | Builtin of vname * ltype
   | Var of vref
   | Susp of lexp * subst
   | Let of (vname * lexp * ltype) list * lexp
   | Arrow of arg_kind * vname option * ltype * lexp
   | Lambda of arg_kind * vname * ltype * lexp
   | Call of lexp * (arg_kind * lexp) list (* Curried call.  *)
   | Inductive of label
                  * ((arg_kind * vname * ltype) list) (* formal Args
   *)
                  * ((arg_kind * vname option * ltype) list) SMap.t
   | Cons of lexp * symbol
   | Case of lexp * ltype
             * ((arg_kind * vname option) list * lexp) SMap.t
             * (vname option * lexp) option
   | Metavar of int * subst * vname * ltype
```

L'élaboration dans Typer effectue les transformations suivantes :

- Propagation, inférences et vérification des types
- Inférences des arguments implicites
- Exécutes les macros
- Vérification de la portée (Scope)

L'élaboration est la dernière phase du front-end. C'est à dire que n'importe quelle erreur, qui a lieu par la suite, est une erreur interne au compilateur et non issue du code passé au compilateur par l'utilisateur.

5.2.1 Les index de De Bruijn

Typer utilise les index de De Bruijn pour représenter les variables. Les noms des variables sont remplacés par la distance entre la variable utilisée et la dernière variable déclarée (indice inverse). Les variables sont donc définies par leur distance relative et non par leur nom. Cette distance s'appelle *l'index de De Bruijn*.

Un tel système nous permet d'hériter d'une propriété intéressante. Effectivement, vérifier que deux morceaux de code sont équivalents, peu importe le nom donné aux variables (*α-equivalence*), devient similaire à vérifier l'équivalence syntaxique d'une expression.

Cependant, il peut rendre le débogage un peu plus ardu. Effectivement, chaque expression est traitée dans un contexte différent (avec un nombre de variables différentes). Ainsi l'index pour une variable `a` est fonction de la taille du contexte. Voici un exemple de code Typer avec l'annotation des index associés à chaque variable.

Listing 5.3 : Index de DeBruijn (Typer)

```
1 length : (a : Type[57]) ≡> ((List[25] a[0]) -> Int[63]);
2 length = lambda (a : Type[58]) ≡>
3   lambda (xs : (List[26] a[0])) ->
4     case xs[0]
5       | cons hd tl => 1 +[42] (length[4] ?a[17] tl[0])
6       | nil => 0;
```

Nous pouvons remarquer que l'index de Type augmente au fur et à mesure que le contexte augmente. Si nous nous penchons un peu plus sur la ligne 1 + (length tl). Le contexte à cet endroit est décrit à la Fig. 5.1. Nous pouvons remarquer que la variable length doit être dans le contexte puisque la fonction est récursive.

Il existe plusieurs cas différents qui requièrent différents types d'ajustements. Ces cas peuvent se combiner, ce qui peut créer un problème d'efficacité si chaque ajustement est appliqué de façon avide, effectivement chaque ajustement va nous obliger à traverser l'arbre de syntaxe. C'est pourquoi nous allons nous munir d'une manière de combiner les transformations et de les appliquer de façon paresseuse au fur et à mesure que l'arbre de syntaxe est traversé. Le système que nous utilisons est un dérivé du calcul $\lambda\sigma$ [12].

Un exemple d'ajustement qui est fait pour chaque déclaration est un *Shift* de 1. Effectivement, toutes nos déclarations sont supposées récursives. Elles doivent ainsi avoir accès à elles-mêmes. Pour cela tous les index des variables libres doivent être augmentés de 1 puisque la fonction elle même est maintenant dans le contexte.

De la même manière, puisque nous utilisons un typage dépendant, le type de l'expression retournée par un *Let* peut faire référence à une ou des déclarations présentes à l'intérieur du *Let*. C'est pourquoi il faut ajuster le type en fonction des déclarations.

Index	Variable
4	length
3	a
2	xs
1	hd
0	tl

Figure 5.1 : Environnent

5.2.2 Le calcul suspendu

Le calcul suspendu est le biais par lequel nous allons garder à jour les index de Debruijn d'une expression, lorsque celle-ci est déplacée dans un nouveau contexte. Le calcul est fait de façon paresseuse pour éviter de traverser l'arbre de syntaxe plus d'une seule fois. Le calcul que nous avons implémenté est une variante du calcul $\lambda\sigma$ [27], celui-ci est décrit en détails dans [3].

Le calcul est implémenté à travers trois opérations :

- Identity
- Cons
- Shift

L'identité (*Identity*) ne modifie pas l'index, le *Shift* modifie de n l'index de De Bruijn, finalement le *Cons* permet de combiner plusieurs substitutions et de les appliquer de façon paresseuse au fur et à mesure que l'arbre est traversé.

Le $\lambda\sigma$ est implémenté de façon à ce qu'il soit pratiquement invisible. Effectivement, l'environnement se charge entièrement des calculs d'index. Lorsqu'une déclaration est recherchée, l'environnement retourne une Susp qui représente la série de transformations à effectuer sur l'expression pour mettre à jour les index. Les transformations ne seront alors effectuées que si elles sont nécessaires.

5.2.3 Le typage dépendant

L'utilisation d'un typage dépendant nous permet d'utiliser n'importe quelle construction du langage incluant des *lambda*, des appels de fonctions ainsi que des *case* à l'intérieur des types. L'exemple ci-dessous représente un vecteur d'entier ayant une taille déterminée par la valeur de x. Si x est premier alors le vecteur sera de taille 5 sinon il sera de taille 6 [32].

Listing 5.4 : Type utilisant un case (Typer)

```
1 Vector Int (case prime? x | true -> 5 | false -> 6)
```

Par conséquence la notion d'égalité entre types revient à la notion d'égalité entre expressions. De plus, un type peut être écrit de plusieurs façons (*confluence*), rendant la vérification des types plus complexe.

Pour pouvoir vérifier l'égalité de deux types, deux fonctions existent lexp_whnf et conv_p. lexp_whnf permet de mettre une expression sous la forme *weak head normal*.

C'est une sorte de forme canonique dont seulement la tête de l'expression est canonique. De façon concrète, lexp_whnf enlève les β réductions des expressions *head* [21]. Une β réduction est le processus par lequel le résultat d'un appel de fonctions est calculé. Ainsi lexp_whnf remplace les appels de fonctions par le corps de la fonction avec ses arguments substitués.

Par la suite, nous pouvons utiliser la fonction conv_p pour tester l'égalité de deux expressions.

5.2.4 Le contexte d'élaboration

Le contexte d'élaboration est composé de deux structures de données, un dictionnaire senv associant à chaque nom de variable un nombre représentant la taille de l'environnent au moment de la création de cette variable et une liste lexp-context contenant les valeurs et type de chaque variables.

Le dictionnaire est utilisé pour calculer l'index de De Bruijn de chaque variable. Par exemple, lorsqu'une variable a $= 2$ est ajoutée à un contexte d'élaboration ayant une taille n, nous ajoutons l'entrée $a \rightarrow n$ au dictionnaire senv. Puis, nous ajoutons, l'entrée (2, Int) à la liste lexp-context.

Pour calculer l'index de la variable a il nous suffit de rechercher la taille de l'environnement i lorsque a a été créée et de retourner n - i - 1 où n représente la taille de l'environnement, nous utilisons la fonction senv_lookup pour effectuer cette opération.

Listing 5.5 : Calcul des index de De Bruijn (ML)

```
let rec senv_lookup (name: string) (ctx: elab_context): int =
  let ((n, map), _) = ctx in
    n - (SMap.find name map) - 1
```

Par exemple, si la variable a vient d'être créée alors son index de De Bruijn est zéro, Pour accéder à l'expressions ou au type de la variable, il nous suffit de récupérer le premier élément de la liste lexp-context.

Comme nous l'avons vu à la section précédente, certaines expressions peuvent nécessiter des ajustements, c'est pourquoi en réalité nous retournons une Susp représentant la série de transformation à effectuer sur l'expression plutôt que l'expression en elle même.

5.2.5 L'inférence de types

Typer est un langage typé statiquement, ainsi le type d'une variable ou d'une expression doit être disponible lors de la vérification des types. Cependant, forcer le programmeur à fournir cette information peut rendre le code verbeux, notamment dans les cas ou le type d'une expression peut être déterminé de façon triviale. C'est pourquoi Typer utilise un système d'inférence pour trouver les types manquants.

Présentement Typer utilise deux systèmes d'inférence de types. Le premier système permet d'inférer localement les types, alors que le second système est plus complexe et utilise l'entierté du code source pour inférer les types.

Nous utilisons un système de *vérification de type bidirectionnel*. L'élaboration se passe à l'intérieur de deux fonctions principales, infer et check

La fonction infer

Listing 5.6 : Fonction infer de Typer (ML)

```
let rec infer (p : pexp) (ctx : elab_context): lexp * ltype
```

Les expressions ayant un type pouvant être inféré, sont passées à la fonction infer. La fonction infer prend en argument une expression pexp et le contexte d'élaboration puis renvoie deux lexp ; l'expression après élaboration et son type. Les expressions gérées par infer sont le Let, l'expression Arrow, l' appel de fonction Call, et les valeurs (entier, nombres décimaux, chaîne de caractères).

Le type de chacune de ces expressions peut être déterminé par le type des expressions qu'elles contiennent.

Valeur immédiate Les valeurs immédiates sont des expressions qui représentent les valeurs de base de Typer (S-exp, entier, chaîne de caractères), leurs types sont donc des builtin de Typer que nous connaissons, nous pouvons donc de façon simple reconnaître le type de valeur contenu par l'expression et retourner sa valeur et son type. De la même façon les builtin ont une valeur et un type donnés par leur définition même, il nous suffit donc de les rechercher dans notre table des builtin et d'en retourner les valeurs.

Listing 5.7 : Inférence des valeurs (ML)

```
let rec infer_var epxr ctx =
  match expr
    | Pimm(value) ->
```

```
4       let value_type = match value
5         | Integer _ -> Builtin.Int
6             ...
7         | String _ -> Builting.String in
8       (make_value (value)), value_type
9
10 let rec infer_builtin expr ctx =
11   match expr
12     | Pbuiltin(name) => look_up_builtin name
```

Les variables Lors de l'élaboration nous devons assigner aux variables leur index de De Bruijn, pour cela nous pouvons calculer l'index en utilisant la fonction senv_lookup décrite dans la section 5.2.4. Grâce à l'index nous pouvons rechercher son type dans le lexp-context.

Listing 5.8 : Inférence des variables (ML)

```
1 let rec infer_var epxr ctx =
2   match expr
3     | Pvar(nom) ->
4         let index = senv_lookup nom ctx in
5         let var_type = env_lookup_type ctx idx in
6                 (make_var (name, index)), var_type
```

Let Chaque déclaration du Let est passée à la fonction lexp_p_decls qui va appliquer la fonction check aux déclarations ayant une annotation de type, et appliquer la fonction infer sinon, elle s'occupe aussi d'ajouter les déclarations dans le contexte. Nous verrons plus en détails l'expansion de macros dans la section 5.2.6. une fois les déclarations ajoutées au contexte nous pouvons inférer le corps du Let pour

déterminer son type.

Listing 5.9 : Inférence du type Let (ML)

```
let rec infer_let epxr ctx =
  match expr
    | Plet(declarations, corps) ->
        let declarations, ctx = lexp_p_decls declarations ctx in
        let corps, corps_type = infer corps, ctx in
          (make_let (declarations corps)), corps_type
```

Arrow Les flèches, ou *Arrow* sont des expressions utilisées pour représenter le type d'une fonction, son type est toujours de type *Type*. Les flèches $(x : a) \to b$ sont composées de deux expressions $(x : a)$ et b, comme nous utilisons le typage dépendant, nous devons ajouter une variable x de type a au contexte pour que l'expression b puisse l'utiliser. Lorsque le type a n'est pas utilisé par b, nous n'avons pas besoin de donner un nom au type a et celui-ci n'a pas besoin d'être ajouté au contexte.

L'élaboration d'une flèche suit la procédure suivante : le type a est inféré puis ajouté au contexte, par la suite b est inférée.

Listing 5.10 : Inférence du type flèche (ML)

```
let rec infer_arrow epxr ctx =
  match expr
    | Parrow(None, a, b) ->
      let a_type, _ = infer a ctx in
      let b_type, _ = infer b ctx in
        (make_arrow(a_type, b_type)), Builtin.Type
    | Parrow(Some var, a, b) ->
      let a_type, _ = infer a ctx in
```

```
9         let ctx = env_extend var a ctx in
10        let b_type, _ = infer b ctx in
11          (make_arrow(a_type, b_type)), Builtin.Type
```

Call Les appels de fonction sont composés de deux parties, la fonction à appeler fun et la liste d'arguments. En premier, nous inférons le type de fun, cela va nous permettre de différencier les appels à une fonction des appels à une macro. De plus, le type de la fonction est nécessaire pour déterminer le type retourné par l'appel de la fonction.

Le type d'une fonction suit le format Arrow(a, b), où *a* est le type de l'argument et *b* est le type de la valeur retournée. Une fonction ayant deux arguments à un type qui suit le format Arrow(a, Arrow(b, c)). Pour trouver le type retourné par notre appel de fonction nous devons itérer de façon simultanée enter le type de la fonction et ses arguments.

Par exemple, imaginons que nous avons un appel à une fonction de type Arrow(a , Arrow(b, c)) avec les arguments suivants (arg1 arg2), nous devons en premier vérifier que le type de arg1 est compatible avec le type a, ajouter l'argument arg1 au contexte, puis faire de même avec les arguments suivants, une fois que tous les arguments ont été vérifiés et ajoutés au contexte, nous avons trouvé le type que l'appel retourne. Dans notre cas ce type est l'expression c.

Si notre fonction avait seulement eu un seul argument, l'appel aurait été de type Arrow(b, c). L'algorithme est décrit ci-dessous. Typer utilise un algorithme similaire, cependant, celui-ci gère aussi le réarrangement des arguments.

Le réarrangement des arguments permet au programmeur de donner les arguments d'une fonction dans un ordre quelconque. Ainsi le programmeur n'a pas

besoin de se souvenir de l'ordre des arguments. L'exemple 5.12 illustre la façon dont le programmeur peut utiliser cette fonctionnalité.

L'implémentation de cette fonctionnalité est directe, si le nom de l'argument attendu n'est pas égal au nom de l'argument donné par l'utilisateur, l'argument est ajouté dans une table, nous recherchons dans notre table d'arguments, pour vérifier si l'argument a été donné précédemment, sinon nous continuons la recherche de l'argument dans la liste d'arguments fournis par le programmeur. Une fois l'argument trouvé, nous recommençons le processus pour le second argument.

Ici nous utilisons la fonction check, c'est une fonction qui effectue l'élaboration sur une expression e à la différence de la fonction infer le type t de l'expression e est connu et nous vérifions que le type de e est bien le même que le type attendu t. La fonction sera présentée plus en détails par la suite.

Listing 5.11 : Inférence de l'appel de fonction (ML)

```
let rec infer_call fun args ctx =
  let fun_expr, fun_type = infer fun ctx in

  if fun_type is macro then
    expand_macro ...
  else
    let rec read_args fun_type args ctx acc =
      match fun_type, args with
        | Parrow(a, b), arg::args ->

            let arg_type = check arg a ctx in
            let ctx = env_extend a ctx in
                reag_args b args ctx (arg::acc)

        | _, [] -> fun_type, List.rev acc in
```

```
16
17      let return_type, args = read_args fun_type args ctx [] in
18          (make_call(fun_expr, args)), return_type
```

Listing 5.12 : Réarrangement (Typer)

```
1 fun = lambda (x : Int) =>
2   lambda (y : Int) ->
3     lambda (z : Int) -> x * y + z;
4
5 a = fun 3 2 1;
6 b = fun (z := 1) (y := 2) (x := 3);
```

Annotations de type Le programmeur peut annoter le type de n'importe quelle expression de la façon suivante `e : t` où `e` est l'expression que nous désirons annoter et `t` est le type de l'expression attendue.

Lorsqu'une expression a une annotation de type, l'expression `t` de l'annotation est inférée, puis nous vérifions avec la fonction *check* que le type de l'expression `e` est bien un type compatible avec l'annotation `t`.

Listing 5.13 : Inférence de l'annotation de type (ML)

```
1 let rec infer_annotations expr ctx =
2   match expr
3     | Phastype(expr, annotations) ->
4       let ann_type, _ = infer annotations ctx in
5         (check expr ann_type ctx), ann_type
```

La fonction check

Listing 5.14 : Fonction check de Typer (ML)

```
let rec check (p : pexp) (t : ltype) (ctx : elab_context): lexp
```

Les expressions dont le type attendu est connu en avance sont passées à la fonction check vérifiant que le type de l'expression et le type attendu sont cohérents. check prend en argument une expression pexp, une lexp représentant le type attendu ainsi que le contexte d'élaboration, puis renvoie une lexp représentant l'expression après élaboration. Les expressions gérées par check sont le Lambda, le case, et les appels de macros.

Lambda Pour effectuer, la vérification du *lambda* nous regardons si le type de l'argument est disponible, si oui il suffit d'effectuer l'inférence sur l'expression fournie, sinon nous extrayons le type de l'argument à partir de l'annotation de la fonction.

Une fois le type de l'argument déterminé nous pouvons ajouter l'argument à l'environnement, et effectuer une vérification *check* sur le corps de la *lambda*.

Listing 5.15 : Vérification du Lambda (ML)

```
let rec check_lambda expr type ctx =
  match expr
    | Plambda(var, Option Var_type, corps) =>
       let arg_type = match var_type
         | Some exp -> Some (infer var_type ctx)
         | None -> None in

       let arg_type, corps_type = match type, arg_type
         | Arrow(arg_type, _, corps_type), None =>
           arg_type, corps_type
         | Arrow(arg_type2, _, corps_type), Some arg_type =>
```

```
12            (check arg_type2 arg_type ctx), corps_type in
13
14        let ctx = env_extend ctx var arg_type in
15
16        let corps = check corps corps_type ctx in
17          make_lambda (var, arg_type, corps)
```

Case Le type d'une case peut être inféré si et seulement si au moins une des branches peut être inférée. En utilisant check nous évitons cette contrainte. Ainsi chaque branche peut contenir d'autres expressions nécessitant check incluant des case ou ou des lambda. De plus, utiliser check pour chaque branche d'un case ne requiert généralement pas d'annotation de la part de l'utilisateur. Effectivement, le type de chaque branche est habituellement donné par les expressions parents. Ainsi l'utilisation de check n'a en pratique pas de désavantage.

Pour vérifier le *case*, nous inférons le type de l'expression cible, grâce à son type, nous pouvons trouver la définition du type inductif dont il est issu. Cela nous permet d'accéder à un tableau contenant tous les constructeurs disponibles pour ce type.

Pour chaque branche du *case* nous vérifions que le constructeur demandé existe bel et bien dans le type inductif. Chaque branche possède une liste de motifs identifiant quels arguments du constructeur sont nécessaires. Il existe deux motifs possibles PatternAny (ou _) qui représente un argument qui n'est pas nécessaire et PatternSym qui représente le nom de la variable qui va contenir la valeur de l'argument du constructeur. Cette variable doit être ajoutée à l'environnement. PatternAny est ajoutée à l'environnement mais sous la forme d'une variable anonyme. Une fois la liste des motifs parcourue et que les variables nécessaires ont été ajoutées au

contexte, nous pouvons vérifier que l'expression associée à cette branche est bien du type attendu. Chaque branche ainsi traitée est ajoutée à une table représentant les différents cas du *case*.

Un aperçu de l'algorithme peut être trouvé en annexe III.

Appels de macro L'expansion de macro est vue en détails dans la section 5.2.6

5.2.5.1 Inférence

Le programmeur n'a pas besoin d'annoter chaque expression qui nécessite la fonction check. Dans l'exemple ci-dessous, ni le lambda, ni le case n'ont besoin d'être annotés car le compilateur va pouvoir utiliser l'annotation de la déclaration pour vérifier le Lambda et le case. Ainsi, peu d'annotations de types sont nécessaires en pratique. Le programmeur peut forcer la vérification ***check*** en annotant les expressions qu'il désire vérifier. De plus, par la suite nous allons voir que l'inférence de types par unification nous permet de déduire le type des expressions sans annotations dans beaucoup de cas.

Listing 5.16 : Annotations et inférence (Typer)

```
1 fun : a -> b;
2 fun = lambda xs -> case xs
3   | constructor1 => v1;
4   | constructor2 => v2
```

Nous avons choisi d'implémenter un tel système car il nous permet de minimiser les annotations de types que le programmeur devait fournir [14]. Le système présenté ci-dessus peut être considéré de façon plus appropriée comme une propagation des types plus que de l'inférence en elle même.

Effectivement, l'inférence est locale, nous utilisons le type des expressions enfants pour déterminer le type de l'expression parent et vice-versa.

La fonction g a un type de la forme $a \rightarrow ?b$, en traversant le corps de la fonction, nous allons pouvoir déterminer le type $?b$ retourné par la fonction. Effectivement, nous allons pouvoir inférer le type de la fonction add et le type qu'elle renvoie. Dans notre cas celle-ci retourne un entier Int. À partir de cette information nous pouvons déterminer que la fonction a le type $a \rightarrow Int$.

Listing 5.17 : Inférence Locale (Typer)

```
1 g = lambda (xs : a) -> (add xs 1);
```

Un tel système reste très simple et n'utilise pas toute l'information disponible. Effectivement, nous pourrions utiliser les appels d'une fonction pour déduire le type des arguments. Les premières versions de l'interpréteur n'utilisaient que le système d'inférence local décrit ci-dessus, ce n'est que par la suite qu'un système d'inférence plus complexe a été ajouté.

Ce système crée des méta-variables pour les types manquants. Les méta-variables sont des variables représentant un type manquant. Par exemple, lorsque nous écrivons foo = lambda x -> ... le type de x est inconnu, une méta-variable $?a$ est donc créée pour représenter son type [9].

Au fur et à mesure que le code source est lu, des contraintes vont être ajoutées sur chaque méta-variable. Par exemple, l'appel foo (lambda y -> ...) nous permet de déduire une partie du type $?a$. Effectivement, $?a$ doit être de la forme $?b \rightarrow ?c$ avec $?b$ étant la méta-variable représentant le type de y et $?c$ le type retourné par la lambda. Nous venons d'ajouter une contrainte sur $?a$.

Petit à petit, le type d'une expression va pouvoir être déterminé à partir de ses contraintes lors de *l'unification*.

Un tel système est particulièrement utile pour déduire les arguments *erasable* qui ne sont pas fournis par le programmeur, mais qui sont nécessaires lors de la vérification des types. Par exemple, avant que l'unification ne soit implémentée, nous devions fournir tous les arguments quels que soient leurs types. Ainsi une simple liste telle que cons 1 (cons 2 nil) était écrite de la façon suivante cons (a := Int) 1 (cons (a := Int) 2 (nil (a := Int))).

La superposition des deux systèmes nous permet de réduire l'utilisation du second système. Effectivement, les cas triviaux sont inférés par le premier système cela nous permet d'éviter la création de méta-variables et ainsi de réduire le travail fait pendant l'unification.

5.2.6 L'expansion de macros

L'expansion de macros est l'étape où les appels de macros sont remplacés par le résultat de leurs évaluations. Les macros manipulent des S-exp et retournent une S-exp. Celles-ci ne portent aucune information de type et pourtant le vérificateur de type doit pouvoir garantir la justesse du typage.

Ainsi l'expansion doit avoir lieu avant la vérification des types pour que le compilateur puisse avoir toute l'information nécessaire. Par exemple, si une macro ajoute de nouvelles déclarations, celles-ci doivent être disponibles pour que le compilateur puisse vérifier le typage des expressions qui les utilisent.

Cependant, la vérification des types doit aussi avoir lieu avant l'expansion des macros. Effectivement, lorsqu'un appel $(f args)$ est trouvé le type de f est utilisé pour distinguer un appel de fonction d'un appel de macro. Pour résoudre ces deux contraintes contradictoires, l'expansion des macros et la vérification de type s'entremêlent.

Un appel de macro a la forme suivante (f arg), dont l'expression f doit avoir un type *Macro* , ou être une expression retournant une macro.

Une fois l'appel reconnu, nous utilisons la fonction handle_macro_call ctx m args pour étendre la macro. Cette fonction prend en argument, le contexte d'élaboration, la macro, l'argument de la macro. Dans l'exemple ci dessous, la macro est m, l'argument est une liste de S-exp ayant un seul élément _.

Listing 5.18 : Exemple de macro (Typer)

```
m = macro (lambda x -> integer_ 2);
quatre = 2 + (m _);
```

Listing 5.19 : Vérification des Macro (ML)

```
let handle_macro_call ctx func args type =
  let macro_expand = get_predef "expand_macro_" ctx in
  let macro_expand = lexp_eval macro_expand ctx

  let func = lexp_eval func ctx in
  let args = ocamllist_to_typerlist args in

  let sxp = match eval_call macro_expand [func; args] with
    | Vsexp (sxp) -> sxp  in

  check (pexp_parse sxp) type ctx
```

La fonction handle_macro_call recherche la fonction d'expansion de macro dans le contexte d'élaboration (5.20). typer-expand est évalué retournant ainsi une fermeture attendant deux arguments. L'expression func représente la macro, celle-ci est évaluée, cela nous permet d'obtenir la définition même de la macro func dans le cas ou func aurait été une expression retournant la définition et non la définition

elle même. Nous pouvons remarquer que nous utilisons la fonction lexp_eval pour évaluer les expressions. Cette fonction se charge de transformer le contexte d'élaboration en contexte d'évaluation, s'assurant ainsi que la définition de la macro peut être effectivement utilisée.

Par la suite, notre liste d'arguments args est transformée d'une liste Ocaml en une liste compréhensible par l'interpréteur de Typer. Il nous suffit maintenant d'évaluer l'appel de la fonction d'expansion (5.20) avec comme argument notre macro et les arguments de la macro. L'évaluateur nous retourne une S-exp que nous pouvons *parser* en pexp. Finalement, nous appelons check pxp ltype pour effectuer l'élaboration sur le code généré, tout en vérifiant que celui-ci est de type ltype.

Listing 5.20 : Fonction d'expansion de macro (Typer)

```
1 expand_macro : Macro -> List Sexp -> Sexp;
2 expand_macro m args = case m
3   | macro f => (f args);
```

Les macros sont des morceaux de codes qui sont exécutés pendant la compilation, cela veut dire que celles-ci doivent être compilées avant le reste du code source, cela ajoute des restrictions sur l'utilisation des macros.

Par exemple, dans certains langages, une macro ne peut pas être utilisée dans le fichier où celle-ci a été définie. Scala est un de ces langages. Effectivement, pour que la macro puisse être compilée en avance, celle-ci doit être dans un fichier à part [7]. D'autres langages ne permettent pas de définir des macros localement. Par exemple, toutes les macros C sont définies globalement et ne peuvent pas être limitées à une portée.

Typer ne possède pas ces restrictions. Effectivement, il possède un interpréteur qui lui permet d'évaluer les macros au fur et à mesure qu'un fichier source est lu.

De plus, les macros peuvent être définies à l'intérieur d'une portée.

Cependant, une restriction demeure. Toutes les valeurs nécessaires à l'expansion des macros doivent être disponibles pendant la compilation, sinon le compilateur ne pourra pas finir l'expansion.

Listing 5.21 : Exemple d'usage non-supporté (Typer)

```
1 call-macro : (List Sexp -> Sexp) -> List Sexp -> Int;
2 call-macro m args = let my-macro = macro m
3   in my-macro 5 6;
```

L'exemple 5.21 illustre un tel cas. Il implémente une fonction `call-macro` qui va construire une macro à l'intérieur même de la fonction puis appeler cette même macro. Lors de la lecture du code, le compilateur va essayer d'évaluer l'appel de fonction suivant `m (cons (cons (integer 6) nil))`. Cependant, la variable `m` ne possède pas encore de valeur et va amener le compilateur à signaler une erreur.

5.3 L'évaluation

Nous avons implémenté un interpréteur basé sur l'arbre de syntaxe même [33]. Un tel interpréteur est nécessaire pour l'expansion des macros, nous utilisons le même interpréteur pour évaluer les fichiers Typer. Nous voulons implémenter le compilateur Typer en Typer avant de nous pencher sur la performance d'exécution.

5.3.1 L'effaçage des Types

Avant d'évaluer nos expressions nous transformons l'arbre de syntaxe original en une version allégée en supprimant toutes les informations de typage. Effectivement, cette information n'est plus nécessaire une fois que la vérification des types a été effectuée. Simplifier l'arbre de syntaxe nous permet ainsi d'alléger notre interpréteur

qui n'a plus accès à des informations superflues.

Une importante modification est tout de même faite pendant l'effaçage des types. Nous avons vu à la section 4.5.3, qu'il existait des arguments *erasable* qui n'étaient pas utiles lors de l'évaluation, puisque leurs valeurs n'étaient pas utilisées dans des calculs. C'est pourquoi de tels arguments sont enlevés pendant l'effaçage des types. Cependant, puisque le nombre d'arguments est modifié il nous faut ajuster l'index des variables présentes à l'intérieur du corps de la fonction, puisque l'environnement sera plus petit.

5.3.2 L'évaluation

L'évaluation est l'étape par laquelle une expression est réduite en une valeur. La liste des valeurs existantes peut être trouvée dans le listing 5.22. Vout et Vin sont des containers permettant de faire des opérations d'entrée/sortie. Closure représente l'évaluation partielle d'une fonction.

Listing 5.22 : Valeur retournée par l'évaluateur)

```
type value_type =
  | Vint of int
  | Vstring of string
  | Vcons of symbol * value_type list
  | Vbuiltin of string
  | Vfloat of float
  | Closure of string * elexp * runtime_env
  | Vsexp of sexp
  | Vundefined
  | Vdummy
  | Vin of in_channel
  | Vout of out_channel
```

```
13    | Vcommand of (unit -> value_type)
```

Pour évaluer une expression elxp il nous suffit d'appeler l'interpréteur de la façon suivante eval elxp ctx trace où ctx est le contexte d'évaluation et trace est l'historique des appels passés, une liste vide peut être passée si aucun historique n'est disponible, La fonction va retourner une valeur de type value_type

Les expressions Imm, Inductive, Cons, Lambda et Builtin ont des correspondances directes avec des éléments de value_type. Aucun travail particulier n'est effectué sur les expressions en elles mêmes. Elles sont converties de façon immédiate en leur contrepartie.

Listing 5.23 : Évaluation des feuilles (ML)

```
1 let eval_leaf expr ctx =
2   match lxp with
3     | Imm(Integer i)      -> Vint(i)
4     | Imm(String s)       -> Vstring(s)
5     | Imm(sxp)            -> Vsexp(sxp)
6     | Inductive           -> Vinductive
7     | Cons label          -> Vcons (label, [])
8     | Lambda (n, lxp)     -> Closure(n, lxp, ctx)
9     | Builtin str         -> Vbuiltin(str)
```

L'évaluation des variables est aussi très directe. Effectivement, il suffit de rechercher l'index de la variable dans l'environnement d'évaluation et de retourner le résultat de la recherche.

Listing 5.24 : Évaluation des variables (ML)

```
1 let eval_var expr ctx =
2   match lxp with
3     | Var (index) -> get_rte_variable index ctx
```

Let

Listing 5.25 : Expression Let

```
type value_type =
  ...
  | Let of (vdef * elexp) list * elexp
```

Notre let est composé de deux parties ; une liste de déclarations et une expression qui est retournée par le Let.

Nous devons nous rappeler que notre let est récursif par défaut. Il nous faut donc ajouter les variables à l'environnement avant d'évaluer les déclarations pour que l'environnement soit de la bonne taille lors de l'évaluation.

Une fois les variables ajoutées nous pouvons évaluer les déclarations une par une et remplacer la valeur par défaut présente dans l'environnement par la valeur calculée par l'interpréteur.

Finalement, nous évaluons l'expression finale du let et retournons le résultat.

Listing 5.26 : Évaluation des Let (ML)

```
let eval_let expr ctx =
  match lxp with
    | Let (declarations, corps) ->
      let n = (length decls) - 1 in

      let ctx = Lfold_left (fun ctx (name, declaration) ->
        add_variable (Some name) Vdummy ctx) ctx declarations in

      iteri (fun idx ((_, name), lxp) ->
       let v = eval lxp ctx in
       let offset = n - idx in
```

```
12          ignore (set__variable offset (Some name) v ctx)) ctx
    declarations;
13
14        eval corps ctx
```

Call

Avant de nous intéresser à l'évaluation du call, nous devons parler de l'évaluation des lambda. Lorsque l'évaluateur rencontre une lambda, une fermeture (*closure*) est créée, celle-ci possède tous les attributs d'une lambda mais elle sauvegarde en plus le contexte d'évaluation dans laquelle elle apparaît. Ainsi lorsque la fermeture est appelée celle-ci est évaluée dans son propre contexte.

Listing 5.27 : Expression Call

```
1 type value_type =
2   ...
3   | Call of elexp * elexp list
```

Les call sont composés de deux expressions, en premier l'expression appelée puis une liste d'arguments. L'appel de fonction est évalué différemment en fonction du type d'expression qui est appelé. En Typer, il existe trois types d'expressions qui peuvent être appelés, une fermeture, un builtin et un constructeur.

Lorsqu'un constructeur est appelé, l'évaluateur va simplement retourner un nouveau constructeur avec les arguments de l'appel. Par exemple, l'expression suivante ((cons a ... b) c ... d) est transformée en (cons a ... b c ... d), i.e les arguments de l'appel c ... d ont été ajoutés aux arguments du constructeur.

Lorsqu'un builtin est appelé, le nom du builtin est recherché dans la table des constructions par défaut. La table retourne alors l'implémentation de la fonction-

nalité en plus du nombre d'arguments nécessaires. Il nous suffit alors de passer les arguments à la fonction builtin que nous avons recherchée.

Finalement, lorsqu'une fermeture est appelée son contexte est extrait et chaque argument y est ajouté. Si le nombre d'arguments est insuffisant une nouvelle fermeture est retournée possédant le contexte nouvellement étendu, sinon le corps de la fonction est évalué et le résultat est retourné.

Listing 5.28 : Évaluation des appels de fonction (ML)

```
let eval_call expr ctx =
  match lxp with
    | Call(f, args) ->
      let f = eval f ctx in
      let args = map (fun e -> eval e ctx) args in

      let eval_call' f args ctx =
        match f, args
          | _, [] -> f
          | Vcons (name, fields), _ ->
            Vcons (n, List.append fields args)

          | Vbuiltin (name), args ->
                (find_builtin name) args

          | Closure (x, corps, ctx), arg::args ->
            let ctx = add_variable (Some x) args ctx in
            let rec bind_args corps args ctx =
              match (corps, args)
                | Lambda (x, corps), arg::args ->
                  let ctx = add_variable (Some x) arg ctx in
                    bind_args corps args ctx
```

```
23                  | _, [] -> eval corps ctx
24                  | _ -> eval_call' (eval e ctx) args in
25                bind_args corps args ctx in
26
27            eval_call' f args ctx
```

Case

Listing 5.29 : Expression Case

```
1 type value_type =
2   ...
3   | Case of elexp * ((vdef option) list * elexp) SMap.t
```

Un case est composé de deux parties. Une expression cible représentant l'expression qui doit être associée à une des branches possibles. L'expression cible doit obligatoirement être un constructeur. Typer ne supporte pas d'autres expressions. La deuxième partie est une table contenant toutes les branches possibles du case.

Pour évaluer le case nous devons en premier extraire le nom et les arguments du constructeur. Grâce au nom du constructeur nous pouvons rechercher l'expression à exécuter à l'intérieur de la table. La recherche retourne une liste de variables à déclarer ainsi que l'expression à évaluer b.

La liste de variables à déclarer est de la même taille que la liste d'arguments du constructeur cible et représente les noms assignés à chaque argument du constructeur cible. Chaque argument du constructeur cible est ajouté au contexte. Finalement, l'expression b est évaluée et retournée.

Dans l'exemple ci-dessous la liste de variables à déclarer serait (None (Some "a") None (Some "b") None), Ainsi seulement a = 2 et b = 4 sont ajoutées au contexte.

Listing 5.30 : Expression Case

```
case (my-cons 1 2 3 4 5)
  | my-cons _ a _ b _ => a + b
  | a-cons b => b;
```

Listing 5.31 : Évaluation d'un case (ML)

```
let eval_case expr ctx =
  match lxp with
    | Case(target, patterns) ->
      let cname, args = match eval target ctx
        | Vcons(name, args) -> cname, args in

      let pattern, expr = find_branch cname patterns in

      let fold2 ctx pattern args =
        match pattern, args
          | (Some var)::pattern, arg::args ->
                fold2 (add_variable (Some var) arg ctx) pattern args
          | _::pattern, arg::args -> fold2 ctx pattern, args
          | _ -> ctx in

       eval expr (fold2 ctx pattern args)
```

5.4 Processus de Développement

Nous avons choisi de commencer par implémenter un sous-ensemble de Typer minimaliste sur lequel nous avons petit à petit ajouté les fonctionnalités manquantes.

Une telle approche nous a permis de commencer à expérimenter avec Typer

pendant le développement du langage, rendant le débogage plus facile.

De plus, cela a permis à plusieurs personnes de travailler sur le code de façon indépendante, sans que l'avancement d'une personne n'affecte l'avancement des autres.

Par exemple, le calcul $\lambda\sigma$ [3], n'a été implémenté qu'après la sortie de la première version de l'interpréteur. Par la suite l'évaluateur n'a nécessité que très peu de modifications pour prendre en compte le $\lambda\sigma$.

Cependant, cette approche nous a aussi obligé à réviser et ré-implémenter certaines parties de Typer plusieurs fois au fur et à mesure que des fonctionnalités étaient ajoutées.

Un exemple serait la lecture des déclarations top-level. Au début, les déclarations top-level étaient implicitement déclarées dans un let récursif (l'implémentation du calcul $\lambda\sigma$ n'était pas encore prête).

Cependant, lorsque nous avons ajouté la possibilité à l'utilisateur de programmer des macros pour générer de nouvelles déclarations, cette approche n'était plus valable (nous ne pouvions pas savoir le nombre de déclarations que la macro allait ajouter).

De plus, la macro devait avoir accès aux déclarations précédentes pour pouvoir être étendue). Nous avons dû donc ré-implémenter cette partie, pour que chaque déclaration soit lue une à une (le calcul $\lambda\sigma$ a été utilisé pour calculer les index de De Bruijn).

5.5 Les utilitaires de débogage

Nous avons essayé de fournir le plus d'informations nécessaires aux programmeurs pour pouvoir déboguer avec facilité leurs applications. Après tout, les pro-

grammeurs passent plus de temps à déboguer qu'à programmer, il est donc important que ce processus soit aussi simple que possible.

Afin d'aider au plus le programmeur un REPL est fourni avec lequel plusieurs options sont disponibles pour aider le débogage.

Listing 5.32 : Typer REPL

```
1  ========== TYPER REPL ==========
2     Typer 0.0.0 - Interpreter - (c) 2016
3
4     %quit          (%q) : leave REPL
5     %help          (%h) : print help
6   ----------------------------------------------------------------
7  In[ 1] >> %help
8     %quit          (%q) : leave REPL
9     %who           (%w) : print runtime environment
10    %info          (%i) : print elaboration environment
11    %calltrace     (%ct): print call trace of last call
12    %readfile           : read a Typer file
13    %help          (%h) : print help
14 In[ 2] >> 2 + 2;
15    Out[ 2] >> 4
16 In[ 3] >>
```

5.5.1 Call Trace

Listing 5.33 : Typer calltrace

```
1 In[ 1] >> %readfile samples/nat.typer
2 In[ 2] >> (plus two three);
3    Out[ 2] >> (succ (succ (succ (succ (succ zero)))))
4 In[ 3] >> %ct
```

```
================ EVAL TRACE ================================
     size = 26 max_printed = 50
------------------------------------------------------------
[ln   1, cl   2] |+- Call: (plus two three)
[ln   1, cl   2] |:+- Var: plus
[ln   1, cl   7] |:+- Var: two
[ln   1, cl  11] |:+- Var: three
[ln  19, cl  13] |:+- Lambda: lambda y -> case x ....
[ln  22, cl  40] |:+- Case: case x | succ  z => ...
[ln  20, cl  14] |:|+- Var: x
[ln  22, cl  25] |:|+- Call: (succ (plus z y))
[ln  22, cl  25] |:|:+- Var: succ
[ln  22, cl  31] |:|:+- Call: (plus z y)
[ln  22, cl  31] |:|:|+- Var: plus
[ln  22, cl  36] |:|:|+- Var: z
[ln  22, cl  38] |:|:|+- Var: y
[ln  19, cl  13] |:|:|+- ...
============================================================
```

5.5.2 Context Dump

Les contextes d'exécution et d'élaboration sont aussi imprimables.

Listing 5.34 : Contexte d'exécution

```
In[  4] >> %who
================ ENVIRONMENT ================================
| INDEX | VARIABLE NAME | VALUE                             |
------------------------------------------------------------
|     47 | Type          |  Type
|     46 | Int           |  Int
|     45 | Float         |  Float
```

```
|     44 | String          |  String
|     43 | Sexp            |  Sexp
                  .
                  .
                  .
```

Listing 5.35 : Contexte d'élaboration

```
In[ 4] >> %info
  ========== LEXP CONTEXT ==========================================
  | NAME        | INDEX    | NAME        | OFF   | VALUE:TYPE     |
  ------------------------------------------------------------------
  | one         | 6        | one         | 1     | (succ zero): Nat
  | two         | 5        | two         | 1     | (succ one): Nat
  | three       | 4        | three       | 1     | (succ two): Nat
  | plus        | 3        | plus        | 1     | lambda (x : Nat) ...
  | odd         | 2        | odd         | 2     | lambda (n : Nat) ...
  | even        | 1        | even        | 1     | lambda (n : Nat) ..
  | o1          | 0        | o1          | 1     | (even two): Int
  ==================================================================
```

5.5.3 AST Dump

Un autre programme de débogage est aussi fourni, celui-ci permet d'imprimer chaque étape de la compilation. Il peut aussi re-générer du code Typer à partir d'un fichier, le code généré sera indenté de façon correcte et aura toutes les macros étendues.

Listing 5.36 : Lexp AST

```
lambda          [ln  18, cl   1] :plus : Nat -> Nat -> Nat;
FILE: ./samples/nat.typer        :plus = lambda (x : Nat) ->
```

```
3                                   :  lambda (y : Nat) ->
4                                   :    case x
5                                   :      | succ z => (succ (plus z y))
6                                   :      | zero => y;
7 Call              [ln  43, cl   1] :o1 : Int;
8 FILE: ./samples/nat.typer         :o1 = (even two);
```

5.6 Évaluation de l'implémentation

Typer est un langage expérimental, toutes les fonctionnalités que le langage doit posséder ne sont pas encore entièrement définies. C'est pourquoi nous nous sommes concentrés sur l'implémentation d'un compilateur flexible qui peut être aisément modifié. Ainsi nous avons porté une attention moindre à la performance de celui-ci.

De plus, nous désirons que Typer puisse devenir *self-hosting*, ainsi seule une base minimale du langage est définie puisque nous allons par la suite ré implémenter Typer en Typer même. Il est donc inutile de polir la présente implémentation au dela du strict nécessaire.

Afin de nous assurer que l'implémentation réponde à nos attentes et que le compilateur fonctionne correctement, j'ai implémenté une petite bibliothèque permettant d'ajouter des tests unitaires. Nos tests sont rangés en sections, chaque section possède une série de tests à effectuer. Nous pouvons filtrer par section et par test. Cela nous permet de pouvoir vérifier un test en particulier sans perdre de temps à exécuter toute la batterie de tests.

Le listing 5.37 présente le résultat des tests pour la section *Sexp*. Cette section vérifie le bon fonctionnement du *parser*.

Listing 5.37 : Test Typer

```
[RUN      ] SEXP
[       OK] SEXP - lambda x -> x + x
[       OK] SEXP - x * x * x
[       OK] SEXP - ((a) ((1.00)))
[       OK] SEXP - (x + y)
[       OK] SEXP - (x := y)
[       OK] SEXP - if A then B else C -> D
[       OK] SEXP - A : B -> C
[       OK] SEXP - f __\; y
[       OK] SEXP - case e | p1 => e1 | p2 => e2
[       OK] SEXP - a\b\c
[       OK] SEXP - (a;b)
[       OK] SEXP - a.b.c . (.)
    ( 2/ 9) sexp ..................................................OK
```

Le code du premier test *lambda* $x \rightarrow x + x$ peut être trouvé ci-dessous. Ce test vérifie que la précédence entre l'opérateur *lambda* et l'opérateur + est correcte. Pour ce faire nous inspectons l'arbre que le *parser* a retourné.

Listing 5.38 : Test Typer : parsing Lambda

```
let _ = test_sexp_add "lambda x -> x + x" (fun ret ->
        match ret with
    | [Node(Symbol(_, "lambda_->_"),
        [Symbol(_, "x");
         Node(Symbol(_, "_+_"), [Symbol(_, "x"); Symbol(_, "x")])])]
          -> success ()
    | _ -> failure ()
)
```

Inspecter un arbre de syntaxe peut être complexe et surtout verbeux. C'est

pourquoi, nous utilisons le parseur pour générer deux arbres puis comparons les résultats en utilisant une fonction d'égalité.

La fonction test_eval_eqv_named prend quatre arguments ; le nom du test, le code Typer déclarant les variables nécessaires au test, l'expression que nous désirons tester et finalement, le dernier argument est la valeur attendue. Les déclarations sont évaluées en premier puis les deux derniers arguments. Les résultats des deux évaluations sont comparés pour vérifier le comportement de l'évaluateur.

Listing 5.39 : Fonction de test

```
let test_eval_eqv_named name decl run res =
  add_test "EVAL" name (fun () ->
    let rctx, ectx = eval_decl_str decl ectx rctx in

    let erun = eval_expr_str run ectx rctx in (* evaluated run expr *)
    let eres = eval_expr_str res ectx rctx in (* evaluated res expr *)

      if value_eq_list erun eres
      then success ()
      else (
        value_print (List.hd erun);
        value_print (List.hd eres);
        failure ()))
```

Une telle fonction nous permet d'écrire des tests rapidement, de façon concise et lisible. Par exemple, il nous suffit d'écrire test_eval_eqv_named "2 + 2" "" "2 + 2" "4" pour ajouter un test vérifiant que l'addition fonctionne.

Nos tests sont répartis dans 9 sections, vérifiant différentes facettes du compilateur incluant ; le parseur, le système de macro, l'évaluateur et la vérification de type.

5.6.1 Builtin Typer Library

Nous avons commencé à écrire la bibliothèque de fonctions offertes par défaut par Typer a fin de pouvoir écrire des tests plus complexes et surtout commencer à ajouter les primitives de base pour nous permettre par la suite d'implémenter le compilateur Typer en Typer.

Une des fonctions les plus intéressante est la macro type dont nous avons parlé précédemment (le code de la macro peut être trouvé dans l'annexe II.1). Celle-ci est utilisée pour simplifier la déclaration de nouveaux types. Un exemple de l'utilisation de la macro peut être trouvé ci-dessous.

Listing 5.40 : La macro type (Typer)

```
1 Nat : Type;
2 type Nat
3  | zero
4  | succ Nat;
```

Dans cet exemple, l'argument passé à la macro type est la S-exp suivante (_|_ Nat zero (succ Nat)). La macro extrait le nom du type défini (Nat) et les constructeurs (zero et succ Nat) pour générer la déclaration du type Nat = typecons (dNat) (zero) (succ Nat);. Par la suite, des alias aux constructeurs de Nat sont aussi ajoutés cela nous permet par exemple d'utiliser la variable zero comme constructeur et non l'expression datacons Nat zero. Le code final généré par la macro type peut être trouvé ci-dessous.

Listing 5.41 : Code généré par la macro type (Typer)

```
1 Nat = typecons (dNat) (zero) (succ Nat);
2 zero = datacons Nat zero;
3 succ = datacons Nat succ;
```

La macro est modérément complexe et longue (90 lignes de code) et est utilisée pour vérifier le bon comportement de notre système de macros. Nous allons voir par la suite que nous utilisons aussi cette macro pour vérifier les performances de l'élaboration.

5.6.2 Performance

Nous avons porté peu d'attention à la performance d'exécution, cependant, nous avons tout de même fait attention pour que l'élaboration soit relativement efficace, par exemple nous utilisons une liste de Myers lorsque les éléments sont souvent recherchés [31].

Pour mesurer la performance de compilation nous avons recopié plusieurs fois dans un fichier l'implémentation de la macro type (qui peut être trouvée en annexe) afin de générer des fichiers de grande taille. Les temps d'exécution peuvent être trouvés dans la Fig. 5.6.2. La colonne *Référence* représente le temps estimé que le compilateur devrait prendre si le temps de traitement augmente de façon linéaire par rapport à la taille du fichier. Nous pouvons remarquer que le temps de traitement semble augmenter de façon linéaire par rapport à la taille du code source. La performance du compilateur reste donc satisfaisante pour notre utilisation présente.

Temps d'exécution				
Taille du fichier (ko)	LOC	Réel (s)	Utilisateur (s)	Référence (s)
3	98	0,12	0,06	0,12
6	195	0,16	0,11	0,24
12	389	0,24	0,19	0,48
23	777	0,40	0,36	0,96
45	1553	0,98	0,94	1,92
90	3105	3,02	2,92	3,84
180	6209	6,81	6,70	7,68
359	12417	12,24	12,18	15,36

5.7 Difficultés rencontrées

5.7.1 Déclarations récursives et macros de déclarations

Lors des premières itérations du compilateur, les macros de déclarations n'étaient pas présentes. Il nous suffisait donc de parcourir la liste des déclarations une première fois pour ajouter leurs noms au contexte puis une deuxième fois pour lier le nom de la variable à l'expression. Ainsi, chaque expression avait la possibilité d'accéder à n'importe quelle autre déclaration sans considération pour l'ordre des déclarations. L'utilisateur n'avait donc pas besoin d'utiliser de déclarations avancées (*forward*).

Listing 5.42 : Déclarations récursives

```
1 a : Type -> Something;
2 b : Type -> Something;
3 a = lambda x ->  case x | ... => (b 1) ...;
4 e = lambda x -> ...;
```

```
5 b = lambda x -> case x | ... => (a 2) ...;
6 f = lambda x -> ...;
7 type C
8         | ...;
9 d = ...;
```

Cependant, une fois les macros de déclarations ajoutées cette approche n'était plus valable. Effectivement, lorsque le compilateur arrive à la macro type C, il faut que les expressions des déclarations précédentes soient disponibles, puisque la macro peut très bien nécessiter une déclaration antérieure pendant l'expansion. De plus, nous ne connaissons pas le nombre de déclarations que la macro va créer après l'expansion, nous avons donc l'obligation d'étendre la macro de déclaration au moment même où celle-ci est repérée.

Ce changement nous oblige à détecter les déclarations récursives. Nous forçons les utilisateurs à utiliser les déclarations *forward* pour les déclarations récursives. Cela nous permet d'éviter d'avoir à inspecter l'expression. Lorsqu'une déclaration *forward* est rencontrée nous supposons que toutes les déclarations venant par la suite sont récursives, jusqu'à ce que la dernière déclaration forward soit remplie par son expression. Dans l'exemple ci-dessus, les déclarations a, e, b sont considérées comme récursives et auront accès à chacune d'entre elles. Mais elles n'auront pas accès à f.

Ainsi les let ont leurs expressions redécoupées en groupe d'expressions récursives. Dans l'exemple ci-dessus a, e, b seront compris dans le même let alors que f, d et toutes les déclarations générées par la macro type seront seules dans leur propre let.

Cette partie a nécessité plusieurs réécritures et de nombreux débogages. Ef-

fectivement, les contraintes énoncées ci-dessus couplées avec les index de De Bruijn rendaient la source des problèmes difficile à identifier.

5.7.2 Les index de De Bruijn

Comme nous l'avons vu dans une section antérieure, nous avons utilisé les index de De Bruijn pour représenter nos variables. Un tel système nous permet de simplifier certaines opérations telle que la comparaison d'expressions. Cependant, les index de De Bruijn viennent avec un coût qu'il ne faut pas sous-estimer. Effectivement, il faut faire particulièrement attention au contexte d'origine et au contexte de destination lorsque les expressions de Typer sont manipulées et surtout il ne faut pas oublier d'ajuster les index au besoin. Les erreurs sont faciles à faire et délicates à déboguer.

CHAPITRE 6

CONCLUSION

Nous avons vu comment Typer permet à l'utilisateur d'élider certains détails lors de l'écriture de son programme grâce à l'élaboration. Cette étape est capable de déduire les informations manquantes en complétant ainsi l'arbre de syntaxe, rendant ainsi l'écriture de code plus simple de la part du programmeur.

Les types dépendants permettent déjà à Typer d'être utilisé pour représenter des preuves formelles, cependant Typer manque encore de nombreuses fonctions de base pour rendre l'écriture de preuves plus plaisante. Nous espérons améliorer l'inférence de type, et permettre la complétion automatique de certaines preuves.

Le système de macro, quant à lui, peut déjà être utilisé pour créer de nouvelles extensions au langage, comme la macro type fournie par la bibliothèque par défaut de Typer.

L'évaluateur Typer est basique et n'a été écrit seulement en vue de l'écriture d'un nouveau compilateur écrit en Typer.

Grâce à Typer nous espérons augmenter la productivité des programmeurs tout en améliorant la fiabilité des programmes écrits grâce à son système de types qui démontre déjà sa versatilité.

6.1 Travaux futurs

Par la suite, nous aimerions ajouter la gestion de modules pour permettre au programmeur d'importer des déclarations issues de différents fichiers sources. C'est une fonctionnalité importante qui devrait par la suite nous permettre d'implémen-

ter le prochain compilateur Typer avec plus d'aise.

Nous espérons pouvoir commencer à implémenter le compilateur Typer en Typer par la suite avant de continuer à ajouter de nouvelles fonctionnalités.

Finalement, nous aimerions affiner l'inférence pour permettre à l'utilisateur d'omettre les preuves triviales et les tautologies, permettant ainsi au programmeur de se concentrer sur la programmation plus que sur l'écriture de preuves.

BIBLIOGRAPHIE

[1] Harold Abelson et Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996.

[2] Andrew W. Appel. Verification of a cryptographic primitive : Sha-256 (abstract). *SIGPLAN Not.*, 50(6) :153–153, juin 2015. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2813885.2774972.

[3] Vincent Archambault-Bouffard et Stefan Monnier. Implementation of explicit substitutions : from $\lambda\sigma$ to the suspension calculus. *HOR*, 2016.

[4] Sara Baase. *A Gift of Fire : Social, Legal, and Ethical Issues for Computing Technology*. Pearson Prentice Hall, 2008.

[5] Gabriele Jost Barbara Chapman et Ruud van der Pas. *Using OpenMP*. The MIT Press.

[6] Yves Bertot. Coq in a hurry. *HAL*, 2010.

[7] Eugene Burmako. Scala macros : Let our powers combine !: On how rich syntax and static types work with metaprogramming. Dans *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3 :1–3 :10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. URL http://doi.acm.org/10.1145/2489837.2489840.

[8] Chris Casinghino, Vilhelm Sjöberg et Stephanie Weirich. Combining proofs and programs in a dependently typed language. *SIGPLAN Not.*, 49(1) :33–45, janvier 2014. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2578855.2535883.

[9] James Cheney. Relating nominal and higher-order pattern unification. Dans *Proceedings of UNIF 2005*, pages 104–119, 2005.

[10] Adam Chlipala. *Certified Programming with Dependent Types : A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

[11] William Clinger. Macros in scheme. *SIGPLAN Lisp Pointers*, IV(4) :17–23, octobre 1991. ISSN 1045-3563. URL http://doi.acm.org/10.1145/1317265.1317268.

[12] Pierre-Louis Curien, Thérèse Hardin et Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2) : 362–397, mars 1996. ISSN 0004-5411. URL http://doi.acm.org/10.1145/226643.226675.

[13] Eigen development Team. Eigen3 developer's manual, 2016. URL https://eigen.tuxfamily.org/dox-devel/TopicCustomizing_CustomScalar.html. [Online ; accessed 19-février-2017].

[14] Joshua Dunfield et Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. Dans *Int'l Conf. Functional Programming*, septembre 2013. arXiv:1306.6032[cs.PL].

[15] Expression templates. Expression templates - Wikipedia, 2016. URL https://en.wikipedia.org/wiki/Expression_templates. [Online ; accessed 15-Février-2017].

[16] Culpepper R. Darais D. Flatt, M. et R. B. Findler. Macors that work together. *Journal of functionnal Programming*, 2012.

[17] Richard P. Gabriel, Jon L. White et Daniel G. Bobrow. Clos : Integrating object-oriented and functional programming. *Commun. ACM*, 34(9) :29–38, septembre 1991. ISSN 0001-0782. URL http://doi.acm.org/10.1145/114669.114671.

[18] Paul Gazzillo et Robert Grimm. Superc : Parsing all of c by taming the preprocessor. *SIGPLAN Not.*, 47(6) :323–334, juin 2012. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2345156.2254103.

[19] Leonardo Giordani. Python 3 oop part 5 - metaclasses, 2014. URL http://blog.thedigitalcatonline.com/blog/2014/09/01/python-3-oop-part-5-metaclasses/. [Online ; accessed 19-février-2017].

[20] Dick Grune et Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood, 1990.

[21] Denis Howe. The free on-line dictionary of computing, 2016. URL http://foldoc.org/. [Online ; accessed 30-Décembre-2016].

[22] Yuliyan Kiryakov et John Galletly. Aspect-oriented programming : Case study experiences. Dans *Proceedings of the 4th International Conference Conference on Computer Systems and Technologies : E-Learning*, CompSysTech '03, pages 184–189, New York, NY, USA, 2003. ACM. ISBN 954-9641-33-3. URL http://doi.acm.org/10.1145/973620.973651.

[23] Siu Kwan Lam, Antoine Pitrou et Stanley Seibert. Numba : A llvm-based python jit compiler. Dans *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7 :1–7 :6, New York, NY,

USA, 2015. ACM. ISBN 978-1-4503-4005-2. URL http://doi.acm.org/10.1145/2833157.2833162.

[24] Dictionnaire Larousse. Dictionnaire larousse, 2017. URL http://www.larousse.fr/. [Online ; accessed 19-février-2017].

[25] GNU libgomp. Gnu libgomp. URL https://gcc.gnu.org/onlinedocs/libgomp/. [Online ; accessed 15-Février-2017].

[26] Vlad Dumitrel Lucian Radu Teodorescu et Rodica Potolea. Moving computations from run-time to compile-time : hyper-metaprogramming in practice. Dans *Proceedings of the 11th ACM Conference on Computing Frontiers Article No. 17*, mai 2014.

[27] P.-L. Curien M. Abadi, L. Cardelli et J.-J. Lévey. Explicit substitutions. *Journal of Functionnal Programming*, 1991.

[28] Gayle Laakmann McDowell. *Cracking the coding interview*. CareerCup, 2014.

[29] Scott Meyers. *Effective C++*. Addison-Wesley, 2005.

[30] Marc-Olivier Killijian Michiaki Tatsubori, Shigeru Chiba et Kozo Itano. Openjava : A class-based macro system for java. Dans *Reflection and Software Engineering*. Springer-Verlag, 2000.

[31] Eugene W. Myers. An applicative random-access stack, 1983.

[32] Ulf Norell. Interactive programming with dependent types. *SIGPLAN Not.*, 48(9) :1–2, septembre 2013. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2544174.2500610.

[33] Terence Parr. *Language Implementation Patterns*. The Pragmatic Programmers, 2010.

[34] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[35] Vincent Archambault-Bouffard Pierre Delaunay et Stefan Monnier. Typer : An infix statically typed lisp. 2017.

[36] François Pottier. Hindley-milner elaboration in applicative style : Functional pearl. *SIGPLAN Not.*, 49(9) :203–212, août 2014. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2692915.2628145.

[37] Nathan V. Roberts, Eunjee Song et Paul C. Grabow. Model interfaces for two-way obliviousness. Dans *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 488–495, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. URL http://doi.acm.org/10.1145/1529282.1529386.

[38] Joe Jevnik Scott Sanderson. Bytecode transformers for cpython, 2016. URL https://github.com/llllllllll/codetransformer. [Online ; accessed 19-février-2017].

[39] Tim Sheard et Simon Peyton Jones. Template meta-programming for haskell. Dans *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. URL http://doi.acm.org/10.1145/581690.581691.

[40] Michiaki Tatsubori. An extension mechanism for the java language. 1999.

[41] The Coq Development Team. *Coq Reference Manual*. URL https://coq.inria.fr/refman/.

[42] Todd L. Veldhuizen. C++ templates are turing complete. Rapport technique, 2003.

[43] Richard Wyatt. Lisp macros. *J. Comput. Sci. Coll.*, 29(2) :86–93, décembre 2013. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=2535418.2535431.

[44] Joel Yliluoma. Guide into openmp : Easy multithreading programming for c++, 2007. URL http://bisqwit.iki.fi/story/howto/openmp/. [Online ; accessed 19-février-2017].

Annexe I

Preuve générée par une tactique

Listing I.1 : Preuve écrite par l'utilisateur

```
(fun x y : nat =>
 eq_ind_r (fun n : nat => n = x * x + 2 * x * y + y * y)
   (eq_ind_r (fun n : nat => n + (x + y) * y = x * x + 2 * x * y + y *
    y)
      (eq_ind_r (fun n : nat => x * x + y * x + n = x * x + 2 * x * y
   + y * y)
         (eq_ind_r (fun n : nat => n = x * x + 2 * x * y + y * y)
            (eq_ind (x * x + (y * x + x * y)) (fun n : nat => n + y *
   y = x * x + 2 * x * y + y * y)
               (eq_ind_r (fun n : nat => x * x + (n + x * y) + y * y =
    x * x + 2 * x * y + y * y)
                  (eq_ind (1 * (x * y))
                     (fun n : nat => x * x + (n + x * y) + y * y = x *
    x + 2 * x * y + y * y)
                     (eq_ind (2 * (x * y)) (fun n : nat => x * x + n +
    y * y = x * x + 2 * x * y + y * y)
                        (eq_ind_r (fun n : nat => x * x + n + y * y =
   x * x + 2 * x * y + y * y) eq_refl
                           (Nat.mul_assoc 2 x y)) (1 * (x * y) + x * y
   ) (Nat.mul_succ_l 1 (x * y)))
                     (x * y) (Nat.mul_1_l (x * y))) (Nat.mul_comm y x)
   ) (x * x + y * x + x * y)
               (Nat.add_assoc (x * x) (y * x) (x * y))) (Nat.add_assoc
    (x * x + y * x) (x * y) (y * y)))
```

```
15          (Nat.mul_add_distr_r x y y)) (Nat.mul_add_distr_r x y x)) (
    Nat.mul_add_distr_l (x + y) x y))
```

Listing I.2 : Preuve générée par une tactique

```
1  (fun x y : nat =>
2   let hyp_list := nil in
3   let fv_list := (x :: y :: nil)%list in
4   natr_ring_lemma1 ring_subst_niter fv_list hyp_list
5     (Ring_polynom.PEmul (Ring_polynom.PEadd (Ring_polynom.PEX BinNums.N
       1) (Ring_polynom.PEX BinNums.N 2))
6        (Ring_polynom.PEadd (Ring_polynom.PEX BinNums.N 1) (Ring_polynom
     .PEX BinNums.N 2)))
7     (Ring_polynom.PEadd
8        (Ring_polynom.PEadd (Ring_polynom.PEmul (Ring_polynom.PEX
     BinNums.N 1) (Ring_polynom.PEX BinNums.N 1))
9           (Ring_polynom.PEmul
10             (Ring_polynom.PEmul (Ring_polynom.PEc (BinNat.N.of_nat 2))
       (Ring_polynom.PEX BinNums.N 1))
11             (Ring_polynom.PEX BinNums.N 2)))
12       (Ring_polynom.PEmul (Ring_polynom.PEX BinNums.N 2) (Ring_polynom
     .PEX BinNums.N 2))) I
13    (eq_refl
14     <:
15     (let lmp :=
16        Ring_polynom.mk_monpol_list BinNums.N0 (BinNums.Npos 1) BinNat.
     N.add BinNat.N.mul BinNat.N.add
17           (fun x0 : BinNums.N => x0) BinNat.N.eqb BinNat.N.div_eucl
     hyp_list in
18       Ring_polynom.Peq BinNat.N.eqb
19         (Ring_polynom.norm_subst BinNums.N0 (BinNums.Npos 1) BinNat.N.
```

```
    add BinNat.N.mul BinNat.N.add
20          (fun x0 : BinNums.N => x0) BinNat.N.eqb BinNat.N.div_eucl
    ring_subst_niter lmp
21          (Ring_polynom.PEmul (Ring_polynom.PEadd (Ring_polynom.PEX
    BinNums.N 1) (Ring_polynom.PEX BinNums.N 2))
22              (Ring_polynom.PEadd (Ring_polynom.PEX BinNums.N 1) (
    Ring_polynom.PEX BinNums.N 2))))
23      (Ring_polynom.norm_subst BinNums.N0 (BinNums.Npos 1) BinNat.N.
    add BinNat.N.mul BinNat.N.add
24          (fun x0 : BinNums.N => x0) BinNat.N.eqb BinNat.N.div_eucl
    ring_subst_niter lmp
25          (Ring_polynom.PEadd
26              (Ring_polynom.PEadd (Ring_polynom.PEmul (Ring_polynom.PEX
     BinNums.N 1) (Ring_polynom.PEX BinNums.N 1))
27                  (Ring_polynom.PEmul
28                      (Ring_polynom.PEmul (Ring_polynom.PEc (BinNat.N.
    of_nat 2)) (Ring_polynom.PEX BinNums.N 1))
29                      (Ring_polynom.PEX BinNums.N 2)))
30              (Ring_polynom.PEmul (Ring_polynom.PEX BinNums.N 2) (
    Ring_polynom.PEX BinNums.N 2))))) = true))
```

Annexe II

La macro *type* de Typer

Listing II.1 : La macro *type* de Typer

```
% build a declaration
% var-name = value-expr;
make-decl : Sexp -> Sexp -> Sexp;
make-decl var-name value-expr =
  node_ (symbol_ "_=_")
    (cons var-name
      (cons value-expr nil));

chain-decl : Sexp -> Sexp -> Sexp;
chain-decl a b =
    node_ (symbol_ "_;_") (cons a (cons b nil));

% build datacons
% ctor-name = datacons type-name ctor-name;
make-cons : Sexp -> Sexp -> Sexp;
make-cons ctor-name type-name =
  make-decl ctor-name
    (node_ (symbol_ "datacons")
      (cons type-name
        (cons ctor-name nil)));

% buil type annotation
% var-name : type-expr;
make-ann : Sexp -> Sexp -> Sexp;
make-ann var-name type-expr =
```

```
26    node_ (symbol_ "_:_")
27      (cons var-name
28        (cons type-expr nil));
29
30  type-impl = lambda (x : List Sexp) ->
31    % x follow the mask -> (_|_ Nat zero (succ Nat))
32    %                Type name  --^     ^------^ constructors
33
34    % Return a list contained inside a node sexp
35    let get-list : Sexp -> List Sexp;
36        get-list node = sexp_dispatch_ (a := List Sexp) node
37          (lambda op lst -> lst)    % Nodes
38          (lambda _ -> nil)     % Symbol
39          (lambda _ -> nil)     % String
40          (lambda _ -> nil)     % Integer
41          (lambda _ -> nil)     % Float
42          (lambda _ -> nil) in % List of Sexp
43
44    % Get a name from a sexp
45    %   - (name t) -> name
46    %   - name -> name
47    let get-name : Sexp -> Sexp;
48        get-name sxp =
49          sexp_dispatch_ (a := Sexp) sxp
50            (lambda op lst -> get-name op)   % Nodes
51            (lambda str    -> symbol_ str)   % Symbol
52            (lambda _ -> symbol_ "error")    % String
53            (lambda _ -> symbol_ "error")    % Integer
54            (lambda _ -> symbol_ "error")    % Float
55            (lambda _ -> symbol_ "error") in % List of Sexp
```

```
56
57  let get-head : List Sexp -> Sexp;
58      get-head x = case x
59        | cons hd _ => hd
60        | nil => symbol_ "error" in
61
62  % Get expression
63  let expr = get-head x in
64
65  % Expression is  node_ (symbol_ "|") (list)
66  % we only care about the list bit
67  let lst = get-list expr in
68
69  % head is (node_ type-name (arg list))
70  let name = get-head lst;
71      ctor = tail lst in
72
73  let type-name = get-name name in
74
75  % Create the inductive type definition
76  let inductive = node_ (symbol_ "typecons")
77    (cons name ctor) in
78
79  let decl  = make-decl type-name inductive in
80
81  % Add constructors
82  let ctors =
83   let for-each : List Sexp -> Sexp -> Sexp;
84       for-each ctr acc = case ctr
85        | cons hd tl => (
```

```
86          let acc2 = chain-decl (make-cons (get-name hd) type-name)
    acc in
87            for-each tl acc2)
88        | nil => acc
89      in for-each ctor (node_ (symbol_ "_;_") nil) in
90
91  % return decls
92    (chain-decl decl       % inductive type declaration
93                ctors);    % constructor declarations
```

Annexe III

Algorithme de vérification du Case

Listing III.1 : Vérification du Case (ML)

```
let rec check_case expr type ctx =
  match expr
    | Pcase(target, branches) ->
      let target, target_type = infer target ctx in

      let constructeurs = match target_type
        | Inductive (..., constructeurs) -> constructeurs in

      let rec process_branches branches ctx lbranches =
        match b
          | (nom-cons, patterns, expr)::branches ->
            let args_cons = lookup_arguments nom constructeur in

            let rec read_patterns patterns arg_cons ctx acc =
               match args_cons, patterns
                 | PatternAny::pats, arg::args ->
                        read_patterns pats args ctx (None::acc)
                 | PatternSym var::pats, (_, arg_type)::args ->
                   let ctx = env_extend ctx var arg_type in
                     read_patterns pats args ctx ((Some var)::acc)
                 | [], [] ->
                   let expr = check expr type ctx in
                      (List.rev acc), expr in

            let v = read_patterns pattenrs args_cons ctx [] in
```

```
26                process_branches branches ctx (StringMap.add nom-cons
    v acc)
27
28          | [] -> lbranches in
29
30       let lbranches = process_branches branches ctx StringMap.t in
31         make_case (target, type, lbranches)
```