THESE

présentée à

L'UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

Spécialité

MATHEMATIQUES

par

Christèle Faure

Quelques aspects de la Simplification en Calcul Formel

Soutenue le 25 Avril 1992, devant le jury suivant :

Daniel Lazard, président
James Davenport, rapporteur
Dominique Duval, rapporteur
André Galligo, directeur de thèse
Jacques Morgenstern, examinateur
Loïc Pottier, examinateur.

Mes remerciements vont à

André Galligo pour sa patience et son libéralisme face à mes idées même les plus saugrenues,

Dominique Duval pour sa lecture attentive de mon mémoire et ses remarques sévères qui m'ont amenée à le restructurer,

James Davenport qui malgré un emploi du temps "serré" a trouvé le temps de lire mon mémoire,

Michael Rusinowitch qui a fait des remarques constructives sur mon mémoire et permis ainsi quelques clarifications,

les membres de mon jury pour avoir accepter d'y participer, surtout un samedi matin,

Yves et Nicole, mes collègues de bureau, pour leur discrétion attentive,

Marc, Loïc, Stéphane, Thierry, Jacques, José qui m'ont aidée par leurs remarques, leurs critiques durant ces trois années,

tous ceux, qu'ils soient parents, amis ou collègues qui m'ont encouragée, soutenue et même stimulée quand j'en avais besoin.

Contents

Introduction

Les ordinateurs ont été initialement utilisés pour faire du calcul numérique, c'est à dire du calcul sur des nombres et éviter ainsi des dépenses de temps considérables pour calculer puis vérifier les calculs à la main.
Ensuite, le calcul formel a été introduit pour manipuler des objets mathématiques de haut niveau tels que les polynômes, les matrices, les nombres algébriques ... et non plus uniquement des nombres. Puis, le calcul symbolique est venue s'y ajouter pour que le calcul formel se rapproche du calcul algébrique effectué à la main et est devenu l'une de ses caractéristiques principales

Calcul numérique :

$$\int_{\pi/4}^{\pi/2} \frac{1}{sin(x)} dx = 0.88137359$$

Calcul formel :

$$\int \frac{1}{sin(x)} dx = ln \left| tan(\frac{x}{2}) \right|$$

$$ln \left| tan(\frac{\pi/2}{2}) \right| - ln \left| tan(\frac{\pi/4}{2}) \right| = 0.88137358701954302524$$

Chaque système de calcul formel contient un interprète qui lit une expression, la calcule et imprime le résultat. Nous décrivons dans la figure suivante les manipulations effectuées sur les expressions (le sens des flèches indique l'ordre chronologique des traitements).

Dans cette thèse, nous prenons le parti de considérer que **le point de vue de l'utilisateur est primordial**. La "personnalité" d'un système de calcul formel se caractérise du point de vue utilisateur par les transformations effectuées sur les expressions entre la lecture et l'impression. Nous étudions

Figure 1: Boucle de “top level”

donc les modes de calcul qu'ils utilisent pour mieux satisfaire la demande de l'utilisateur.

Dans l'avant propos, nous passons en revue les différentes formes de déclarations proposées à l'utilisateur pour caractériser les calculs qu'il veut voir effectués par le système. Nous étudions quatre systèmes que nous qualifions de classiques : Macsyma, Reduce, Maple, Mathematica et un cinquième plus original : Scratchpad/Axiom.
Traditionnellement, on distingue deux mécanismes de calcul dans les systèmes de calcul formel : la simplification qui gère règles de transformation et propriétés et l'évaluation qui gère les fonctions de calcul.
L'évaluation, application d'une fonction à des arguments, est un mécanisme général des langages de programmation (notamment des langages applicatifs tels que Lisp).
Tandis que la simplification est une particularité des systèmes de calcul formel qui reproduit, ou tout du moins essaie de reproduire, les manipulations algébriques effectuées “à la main”. Nous soulignons par des exemples deux défauts existant dans les systèmes classiques qui nous paraissent importants et ont motivé notre réflexion : la mauvaise utilisation de certaines informations connues par le système (Problème I) et l'impossibilité de déclarer des informations “fines” dont a besoin tout utilisateur exigeant (Problème II).

Dans la première partie de cette thèse, nous introduisons la notion de **calcul équationnel** pour analyser les processus de simplification et d'évaluation.
La simplification introduite dans les systèmes classiques de calcul formel prend en compte des propriétés et des règles de transformation déclarées par l'utilisateur. Nous unifions et clarifions ces deux formes de simplification en les considérant comme deux formes de réécriture équationnelle. Ceci précise l'action du simplificateur ainsi que les informations qu'il utilise (propriétés

et règles).
Une fois la simplification ainsi délimitée, on définit l'évaluation comme le mécanisme complémentaire : l'évaluation effectue les transformations que ne réalise pas la simplification. De plus, nous considérons que l'évaluateur d'un système de calcul formel applique les fonctions modulo les équations d'associativité et de commutativité déjà utilisées lors de la simplification. Ces spécifications de la simplification et de l'évaluation nous permettent d'apporter une solution élégante au Problème I.

La seconde partie de cette thèse introduit une nouvelle catégorie d'informations : **les types formels**, qui affine les informations véhiculées par les règles, les propriétés et les fonctions de calcul.
On peut remarquer que lorsqu'on calcule "à la main", on sait dans quel domaine mathématique on évolue et on peut réaliser des transformations spécifiques à ce domaine. Cette classe d'informations n'est (essentiellement) utilisée dans les systèmes de calcul formel que par le biais de la construction d'objets mathématiques complexes (matrices, polynômes ...), accompagnés d'opérateurs qui leur sont généralement associés pour permettre une manipulation automatique. Par exemple, lorsque l'utilisateur précise une expression en la construisant comme une matrice (donnée par ses lignes), le système sait utiliser cette information et effectue une addition de matrices.
Nous avons essayé d'abstraire toutes ces informations en les attachant à des types pour contraindre la manipulation des expressions, conformément à ce qui est habituellement fait à la main. Nous montrons également comment la simplification par règles et par propriétés se trouve affinée quand les expressions sont préalablement typées. Ce typage définit le champ d'action de l'évaluation par similitude avec l'interprétation utilisée en calcul des modèles (logique). Grâce à ce formalisme enrichi, le Problème II est entièrement résolu.

Dans la troisième partie de cette thèse, nous proposons un modèle de calculateur entièrement paramétrable pour privilégier "le point de vue de l'utilisateur". Nous décrivons de façon détaillée les choix d'implémentation mis en oeuvre dans le prototype Ulysse pour se conformer aux principes dégagés dans les deux premières parties. Nous différencions les mécanismes de calcul qui forment le **moteur** des informations utilisées lors des calculs qui sont mémorisées dans une **base**.
Nous illustrons les possibilités de calcul décrites dans les deux parties précédentes par la retranscription d'une courte session d'Ulysse.

En annexe nous décrivons une extension d'Ulysse appelée Athéna [CAJL] développée par Loïc Pottier. Les notations, les définitions ainsi que les choix que nous avons été amenés à préciser durant la thèse sont regroupés dans une autre annexe. Chacune des trois parties de cette thèse débute par une courte introduction.

Avant propos : Mode déclaratif des systèmes de calcul formel existants

Lorsqu'une expression est soumise à un système de calcul formel, celui-ci la lit, la calcule et imprime sa forme calculée. Le calcul est conditionné par les opérateurs qui le constituent, nous voulons décrire comment ce conditionnement est implémenté dans les systèmes de calcul formel et préciser les composants élémentaires du calcul.

Les systèmes savent effectuer des calculs par défaut, car ils possèdent des opérateurs prédéfinis considérés comme standards dans le calcul formel et qui sont associés à des calculateurs dédiés.
Les opérateurs prédéfinis sont les opérateurs algébriques les plus couramment utilisés : $+, *, -, /, abs, log, \sqrt{}, \cdot$. Soit f un opérateur prédéfini, le calcul de l'expression $f(x_1, \cdots, x_n)$ est effectué par une fonction fct dédiée à f. L'utilisateur ne peut pas intervenir sur cette fonction, par contre il peut appeler explicitement des fonctions qui ont d'autres point de vue sur cet opérateur et modifie son action :

Maple : `evalb` ([CGGW88] pages 148-152), `evalc, evalm, evalf`, `simplify/trig` ([CGGW88] pages 191,198-202), `simplify/exp, radsimp`,

Macsyma : `trigexpand` ([MIT83] chapitre 5 pages 75-77), `trigreduce, trigsimp`,

Axiom : `Simplify` ([Gro89] pages 210-212), `removeCos, simplifyLog, ratform`.

Comme il nous est impossible d'analyser ces fonctions qui sont dans le coeur des systèmes, nous allons étudier les possibilités offertes à l'utilisateur pour contrôler les calculs. Et par ce biais nous allons définir les composants de base du calcul formel. L'utilisateur modifie le comportement d'un système de calcul formel en déclarant des règles de transformation, des propriétés, ou en définissant des fonctions ou des procédures.

Dans les sections suivantes, nous allons décrire ces moyens d'action offerts à l'utilisateur afin de préciser ce que sont les mécanismes de base du calcul formel. Puis nous allons analyser les différences entre le calcul formel existant et ce que pourrait être un calcul mathématique automatisé.

Déclaration de propriétés

Pour modifier le comportement du simplificateur dans les systèmes classiques, il est possible d'associer des propriétés aux opérateurs. Le simplificateur teste alors ces propriétés et applique la fonction de simplification correspondante.

Les fonctions `define` ([CGGW88] pages 141-143) de Maple, `declare` ([MIT83] chapitre 8 pages 6-7) de Macsyma, `SetAttributes` ([Wol88] pages 214-126) de Mathematica permettent de déclarer des propriétés.
Les propriétés algébriques acceptées sont :

1. Maple : `associative, commutative, identity=`*Id*, `inverse=`*Op*, `zero=`*Zero*, `group, linear`
2. Macsyma : `additive, outative, linear, multiplicative, lassociative, rassociative, commutative`, `symmetric`, `antisymmetric`,
3. Mathematica : `Orderless, Flat, OneIdentity, Listable`
4. Reduce : `linear, noncom, symmetric, antisymmetric` ([Hea85] pages 40-43)

Remarque 1 *Lorsque nous parlons de propriétés, nous sous-entendons propriétés algébriques. En effet il y d'autres formes de propriétés utilisées dans les systèmes. Par exemple en Macsyma, il existe des propriétés de typage :* `integer, noninteger,`

`even, odd, rational, irrational, real, imaginary, complex, analytic` *; des propriétés syntaxiques :* `infix, prefix` *([Hea85] pages 40-43),* `nary, binary` *([MIT83] chapitre 8 pages 6-7), et d'autres :* `increasing, decreasing, oddfun, evenfun, posfun`.

Exemple 2 *Déclaration de quelques propriétés*[1] *:*

```
>define(f,inverse=g);
>f(x,g(x),y);
                 f(y)

In[1]:= SetAttributes[f,Flat]

In[2]:= f[x,f[x,y],z]
Out[2]= f[x, x, y, z]
```

```
(C1) declare(f,multiplicative);
(D1)                    DONE
(C2) f(x*y);
(D2)                 F(X) F(Y)
psl? linear f;
psl? f(x+y,z);
f(x,z)+f(y,z)
```

Déclaration de règles

Nous appellerons règle de transformation toute égalité utilisée dans un seul sens pour transformer les expressions. Une règle est alors composée d'un motif ou *pattern* (sa partie gauche) et d'une expression de remplacement (sa partie droite). Les règles peuvent être plus ou moins compliquées suivant la structure de leur *pattern*, nous allons décrire des déclarations avec des *patterns* de plus en plus compliqués.

Patterns sans variables

Les règles sans variables peuvent être définies par `define` en Maple, par `tellsimp, tellsimp-after` ([MIT83] chapitre 9 pages 2-4) en Macsyma, par `let` ([Hea85] pages 67-73) en Reduce et par `:=` ([Wol88] pages 193-195).

Exemple 3 *Déclaration globale de règles :*

[1] Nous différencieront par la suite les systèmes par leurs prompts standard : `>` pour Maple, `C1,D1` pour Macsyma, `In[1],Out[1]` pour Mathematica et `psl?` pour Reduce (ce dernier n'est pas le prompt standard).

```
>define(f,f(x,y)=3);
>f(3,4);
             f(3, 4)
>f(x,y);
                3
```

```
In[1]:= f[x,y]:=3

In[2]:= f[3,4]
Out[2]= f[3, 4]

In[3]:= f[x,y]
Out[3]= 3
```

```
(C1) tellsimp(f(x,y),3);
(D1)     [FRULE1, FALSE]
(C2) f(3,4);
(D2)         F(3, 4)
(C3) f(x,y);
(D3)            3
```

```
psl? let(f(x,y)=3);
Declare f operator ?  (Y or N)
psl? y

psl? f(3,4);
f(3,4)

psl? f(x,y);
3
```

Patterns **avec variables**

Dans les exemples précédents les règles de simplification sont utilisées par le système comme des substitutions puisqu'il n'y a pas de variables dans les *patterns*.
Il est nécessaire de pouvoir définir des règles s'appliquant pour plusieurs valeurs, il faut alors considérer certains symboles comme des variables du pattern. Ces symboles doivent donc être déclarés comme des variables quand ils sont utilisés dans un *pattern*. En Macsyma les déclarations se font globalement par `matchdeclare` ([MIT83] chapitre 9 pages 2-3), en Maple (respectivement Reduce) elles se font localement à une règle par `forall` (respectivement `for all` ([Hea85] pages 68-69)) et en Mathematica les déclarations sont "syntaxiques" (c'est à dire que les variables de *pattern* sont lexicalement différentes des autres) les variables de pattern sont de la forme `_, x_` ([Wol88] pages 227-228).

Exemple 4 *Application de règles dont le pattern contient des variables :*

```
>define(g, forall(x,g(x,y)=3));
> g(k(x,y,z),y);
                          3
> g(3,4);
                     g(3, 4)
```

```
(C8) matchdeclare(x,true);
(D8)              DONE
(C9) tellsimp(g(x,y),3);
(D9)        [GRULE1, FALSE]
(C11) g(k(x,y,z),y);
(D11)              3
(C13) g(3,4);
(D13)           G(3, 4)
```

```
In[22]:= g[x_,y]:=3

In[24]:= g[k[x,y,z],y]
Out[24]= 3

In[25]:= g[3,4]
Out[25]= g[3, 4]
```

```
psl? for all x let g(x,y)=3;
Declare g operator ?  (Y or N)
psl? y

psl? g(k(x,y,z),y);
3

psl? g(3,4);
g(3,4)
```

Il faut noter que Maple n'accepte que les *patterns* linéaires, c'est à dire contenant une seule fois chaque variable.

```
> define(f,associative, forall(x, f(x,g(x))=Id));
Error, (in define/forall) forall variables should be used
exactly once in lhs
```

Mathematica propose beaucoup de formes de variables de *pattern* différentes : `_,x_,x__,x___,x..,` `x...` ([Wol88] pages 572-574).

Patterns avec variables contraintes

Les déclarations précédentes permettent l'application des règles quelque soit la valeur de leurs variables, mais dans certains cas il est nécessaire de restreindre ce champs de valeurs. Pour cela il est possible de spécifier que les valeurs des variables doivent vérifier un prédicat ou/et avoir un type donné.

Maple, Macsyma, Mathematica définissent un ordre sur les règles utilisées lors de la recherche d'une règle applicable. Les règles sont ordonnées par

leur *patterns* par exemple les *patterns* sans variables (clos) sont plus petits que les autres. Ainsi, pour définir la fonction fibonacci dans ces systèmes, il n'est pas nécessaire de spécifier que $n > 2$ pour que le calcul se termine. Par contre Reduce ne fait pas cette distinction et il faut dire que `n` est supérieur à 2 sinon la réécriture est infinie.

Exemple 5 *L'argument de la fonction fibonacci doit être un entier :*

```
>define(fibo, forall(integer(x),fibo(x)=fibo(x-1)+fibo(x-2)),
              fibo(1)=1,fibo(2)=1);
>fibo(10);
                    55
```

```
(C1) tellsimp(fibo(1),1);
(D1)                          [FIBORULE1, FALSE]
(C2) tellsimp(fibo(2),1);
(D2)                      [FIBORULE2, FIBORULE1, FALSE]
(C3) matchdeclare(n,integerp);
(D3)                                 DONE
(C4) tellsimp(fibo(n),fibo(n-2)+fibo(n-1));
(D4)                [FIBORULE3, FIBORULE2, FIBORULE1, FALSE]
(C5) fibo(10);
(D5)                                  55
```

```
In[13]:= fibo[n_Integer]:=fibo[n-1]+fibo[n-2]
In[14]:= fibo[1]=fibo[2]=1

In[15]:= fibo[10]
Out[15]:= 55
```

```
psl? let fibo(1)=1;
psl? let fibo(2)=1;
psl? for all n such that integerp n and n>2
     let fibo(n)=fibo(n-1)+fibo(n-2);
```

```
psl? fibo(10);
55
```

Gestion des règles

Dans tous les systèmes, les règles sont automatiquement associées aux opérateurs de tête du *pattern*, sauf en Maple où il faut le faire explicitement[2] lors de la déclaration de la règle.

Les systèmes Reduce et Macsyma modifient les règles qui ont pour opérateur $+$ ou $*$ de manière à éviter de faire du *pattern-matching* associatif. Ils font passer l'opérateur à droite de la règle, par exemple $l+m=n$ est transformé en $l = m-n$, $x/2 = y$ en $x = 2*y$, $a-b = c$ en $a = b+c$ ... Les règles mémorisées par le système sont donc différentes de celles données par l'utilisateur, et la forme des expressions calculées ne correspondent pas à celle qu'il attend.

La stratégie d'application des règles est la stratégie de simplification, c'est à dire que les sous-expressions sont calculées avant l'expression qui les contient.

Mais il est possible de diriger l'application d'une règle, c'est à dire indiquer si elle doit être appliquée tant que possible ou une seule fois ...
En Macsyma par exemple, les règles peuvent être nommées par `defrule` ([MIT83] chapitre 9 pages 5) et utilisées par `apply1,apply2,applyb1` ([MIT83] chapitre 9 pages 5) suivant la stratégie désirée.

Définition de procédures et de fonctions

Un système évalue une expression d'opérateur f si une fonction ou une procédure de nom f a été définie.
Bien sûr, tous les systèmes permettent la déclaration de fonctions.

[2] La déclaration `>define(f,g(x,y)=3)` n'a aucun effet car `f` est différent de `g`.

Dans les systèmes classiques

On peut définir un opérateur formellement par des propriétés ou des règles, mais il est souvent important de lui associer une procédure de calcul.

Pour cela, il faut associer au nom de l'opérateur un objet procédure, par `proc` ([CGGW88] pages 74-75) en Maple, par `block` ([MIT83] chapitre 2 pages 12) en Macsyma, par `Block` ([Wol88] pages 252-253) en Mathematica, par `procedure` ([Hea85] pages 81-86) en Reduce.

Exemple 6 *Définition de procédures :*

```
> f := proc(x) local y; y:=x+3; y end;

> f(y);
                    y + 3
> f(4);
                      7

(C8) f(x) := block([y],y:x+3,y);
(D8) F(X) := BLOCK([Y], Y : X + 3, Y)
(C9) f(y)
(D9)               Y + 3
(C10) f(4)
(D10)                7

In[29]:=  f[x_] := Block[{y}, y=x+3 ; y]
In[30]:= f[y]
Out[30]= 3 + y
In[31]:= f[4]
Out[31]= 7

psl? procedure f(x);
       begin y; y:= x+3; return y; end;
f
psl? f(y)
y + 3
psl? f(4)
7
```

En Axiom

Axiom contient deux modes d'utilisation, le mode interprété et le mode compilé. Des fonctions peuvent être définies dans ces deux modes en utilisant l'opérateur == ([Gro89] pages 127).

La bibliothèque de fonctions d'Axiom peut être complétée grâce à un langage de programmation fortement typé : pour être compilées les fonctions doivent être entièrement typées. Axiom autorise non seulement la définition de fonctions, mais aussi celle de nouveaux types, par l'intermédiaire de définitions de `domaines, categories` ([Dav91]).

Exemple 7 *Voici la description de la catégorie* `Ring` *donnée par Axiom :*

```
->)show Ring
 Ring is a category constructor.
 Abbreviation for Ring is RING
 This constructor is exposed in this frame.
 Issue )edit xcatdef.spad to see algebra source code for RING

                                Operations

* : ($,$) -> $                        * : (Integer,$) -> $
+ : ($,$) -> $                        - : $ -> $
- : ($,$) -> $                        = : ($,$) -> Boolean
1 : () -> $                           0 : () -> $
coerce : $ -> $                       coerce : Integer -> $
coerce : $ -> Expression              recip : $ -> Union($,"failed")
** : ($,NonNegativeInteger) -> $
characteristic : () -> NonNegativeInteger
```

Voici la description du domaine des entiers `I` *:*

```
                                Operations

* : (I,I) -> I                        * : (I,I) -> I
+ : (I,I) -> I                        - : I -> I
```

```
- : (I,I) -> I                          < : (I,I) -> Boolean
= : (I,I) -> Boolean                    1 : () -> I
SqFr : I -> FactoredForm I              0 : () -> I
abs : I -> I                            associates? : (I,I) -> Boolean
coerce : I -> I                         coerce : Integer -> I
coerce : I -> Expression                deriv : I -> I
factor : I -> FactoredForm I            gcd : (I,I) -> I
lcm : (I,I) -> I                        max : (I,I) -> I
min : (I,I) -> I                        numberOfDigits : (I,I) -> I
oddp : I -> Boolean                     prime? : I -> Boolean
quo : (I,I) -> I                        random : () -> I
recip : I -> Union(I,"failed")          rem : (I,I) -> I
sizelp : (I,I) -> Boolean               unit? : I -> Boolean
** : (I,NonNegativeInteger) -> I
characteristic : () -> NonNegativeInteger
coerce : Expression -> Union(I,"failed")
deriv : (I,NonNegativeI) -> I
div : (I,I) -> Record(quotient: I,remainder: I)
expressIdealElt :
      (I,I,I) -> Union(notInIdeal,Record(coef1: I,coef2: I))
exquo : (I,I) -> Union(I,"failed")
princIdeal : (I,I) -> Record(coef1: I,coef2: I,generator: I)
unitNormal : I -> Record(unit: I,coef: I,associate: I)
```

Par contre, les fonctions définies à l'interprète ne sont pas nécessairement entièrement typées. L'interprète "connaît" les conversions autorisées ainsi que les différents types de chaque opérateur et peut donc calculer le type des fonctions. S'il reste des ambiguïtés dans le type, la fonction n'est alors pas typée automatiquement, et l'utilisateur peut alors lever les ambiguïtés en forçant des types grâce à l'opérateur : ([Gro89] pages 228).
Lors de l'appel d'une fonction, l'interprète type sa définition, la compile et utilise ce code compilé pour évaluer l'expression. Si la définition ne peut pas être typée, le système signale qu'il ne peut effectuer le calcul.

Exemple 8 *Nous reprenons l'exemple défini dans les autres systèmes :*

```
->f(x) == x+3
                                        Type: VOID
```

```
->f(4)
   Compiling function f with type PositiveInteger -> PositiveInteger

   (2)  7
                                Type: PositiveInteger

->f(y)
   Compiling function f with type Variable y -> Polynomial Integer

   (3)  y + 3
                                Type: Polynomial Integer

-> f(x:Integer):Integer == x+3
   Function declaration f : Integer -> Integer has been added to
   workspace.
   Compiled code for f has been cleared.
   1 old definition(s) deleted for function or rule f
                                Type: Void

-> f(9)
   Compiling function f with type Integer -> Integer

   (9)  12
                            Type: PositiveInteger

-> f(y)
   Coercion failed in the compiled user function f.

   Cannot coerce from type Variable y to Integer for value y.
```

Description de problèmes qui ont influencé notre travail

Nous avons énuméré les différentes formes de déclarations réalisables concernant un opérateur, nous allons maintenant définir les mécanismes qui les gèrent.

On peut considérer que les règles de transformations et les propriétés sont gérées par des mécanismes de simplification alors que les procédures et les fonctions sont elles prises en compte par un évaluateur. Mais si on regarde ce qui se passe dans chacun des systèmes étudiés, on constate que ce n'est pas complètement vrai.

Pour Maple, la déclaration de règles ou de propriétés n'est qu'une manière concise de définir une procédure. En effet Maple transforme les déclarations de règles et de propriétés d'un opérateur en une procédure qui est bien sûr gérée par l'évaluateur. Donc en Maple, l'utilisateur ne peut en réalité que définir des procédures.
En Mathematica, les seules déclarations possibles sont les propriétés et les règles de transformation (même les procédures sont considérées comme une simple séquence d'évaluation), le système n'effectue donc que de la simplification.
Dans l'interprète d'Axiom, l'utilisateur ne peut définir que des fonctions, celui-ci ne contient donc qu'un évaluateur. Il existe néanmoins des modules spécialisés de manipulation d'expression tels que "pattern-matching".
En Reduce et Macsyma, par contre les trois sortes d'information sont gérées différemment mais l'évaluation est privilégiée.

Dans les systèmes que nous avons décrits, des points de vue différents concernant les informations associées à un opérateur ont été adoptés. Excepté en Axiom, les informations définissant les opérateurs peuvent être exprimées sous trois formes : les règles, les propriétés et les procédures. Mais il est clair que les procédures seules suffisent à tout définir (voir ce qui se passe en Maple), mais pour des raisons de concision d'expression, propriétés et règles sont importantes.
Il nous semble que ces trois types d'informations servent naturellement à exprimer des renseignements eux aussi différents. Nous allons dans cette thèse essayer de spécifier cette différence, et par la même redéfinir les simplificateurs (par propriétés et par règles) et l'évaluateur.
En essayant d'éclaircir ces notions, nous allons résoudre les deux problèmes décrits ci dessous.

Problème I
L'associativité est une propriété très souvent utilisée en calcul formel. Les opérateurs associatifs sont binaires, mais souvent notés de façon n-aire dans les systèmes pour des raisons d'efficacité. En effet la notation n-aire permet de ramener les tests d'égalité modulo associativité à un test d'égalité

trivial. Voici un exemple des difficultés de programmation attachées à cette approche.

Exemple 9 *On souhaite définir un opérateur noté* `mult` *qui est associatif, a pour élément neutre* `1` *et pour opposé* `inv` *et qui se contente d'effectuer la multiplication de ses arguments entiers, ainsi* `mult(2,mult(3,x,y),5)` *est calculé en* `mult(6,x,y,5)`.

On pense naturellement pouvoir définir `mult` *par des propriétés et des règles comme ceci en Maple :*

```
> define(mult,associative,inverse=`inv`,identity=1,
>             forall([integer(x),integer(y)],mult(x,y)=x*y));
```

Ce qui revient à définir la procédure `mult`*, qui prend un nombre variable d'arguments et modifie la liste de ces arguments, comme ceci :*

```
proc()
local _AA,i,j,Sign;
options remember;
    _AA := [args];
    _AA := map(proc(x) if x <> 1 then x fi end,_AA);
    _AA := map(proc(x,pn)
                  if type(x,function) and (op(0,x) = pn)
                  then op(x)
                  else x
                  fi
              end,
        _AA,procname);
    for i from 2 while i <= nops(_AA) do
        if
           type(_AA[i],function) and (op(0,_AA[i]) = inv) and
           (nops(_AA[i]) = 1) and (_AA[i-1] = op(1,_AA[i]))
        or type(_AA[i-1],function) and (op(0,_AA[i-1]) = inv) and
           (nops(_AA[i-1]) = 1) and (_AA[i] = op(1,_AA[i-1]))
        then _AA := subsop(i-1 = NULL,i = NULL,_AA);
             if i = 2 then i := 1 else i := i-2 fi
        fi
    od;
```

```
    if _AA = [] then RETURN(1) fi;
    if type(_AA,[integer,integer]) then RETURN(_AA[1]*_AA[2]) fi;
    'procname'(op(_AA))
end
```

Dans le code de cette procédure, la variable `_AA` *qui contient la liste des arguments est modifiée successivement par des instructions qui correspondent à la gestion des propriétés* `identity=1`, `associative`, *et* `inverse=‘inv‘` *ainsi que de la règle* `forall([integer(x),integer(y)]`
A la fin, l'expression est reconstruite à partir de l'opérateur qui se trouve dans la variable `procname` *et des arguments qui se trouvent dans la liste* `_AA`.

Avec cette définition de `mult`, *Maple simplifie correctement, mais n'effectue pas correctement les calculs sur les entiers.*

```
> mult(2,inv(2),x,y,5,2);
                                  mult(x, y, 5, 2)
```

Pour obtenir le comportement désiré, il faut remplacer la ligne :

```
if type(_AA,[integer,integer]) then RETURN(_AA[1]*_AA[2]) fi;
```

par :

```
    for i from 2 while i <= nops(_AA) do
        if
          type(_AA[i],integer) and type(_AA[i-1],integer)
        then _AA := subsop(i-1 = NULL,i = _AA[i] * _AA[i-1],_AA);
            if i = 2 then i := 1 else i := i-2 fi
        fi
    od;
```

Les calculs sont alors effectués correctement.

```
> mult(x,1,2,3,y,inv(y),4);
                          mult(x,24)
```

Comme `mult` *est associatif, la déclaration de la règle binaire* `forall([integer(x),integer(y)],mult(x,y)=x*y)`

devrait suffire pour que Maple construise automatiquement le morceau de procédure voulu.

Si on traite cet exemple en Mathematica, on remarque le même problème : il faut prendre en compte explicitement l'associativité de l'opérateur pour que les calculs se fassent correctement.

La déclaration suivante ne suffit pas :

```
In[1]:= SetAttributes[mult,Flat]
In[2]:= mult[X_Integer,Y_Integer]:= X*Y
In[3]:= mult[X_,inv[X_]]:= 1
In[4]:= mult[inv[X_],X_]:=1
In[5]:= mult[1,Y_]:=Y
In[6]:= mult[Y,1]:=Y

In[7]:= mult[2,inv[2],x,y,5,2]
Out[7]= mult[2, inv[2], x, y, 5, 2]

In[8]:= mult[2,inv[2],x,1,y,5,2]
Out[8]= mult[2, inv[2], x, y, 5, 2]
```

Il faut déclarer des règles de la forme :

```
In[9]:= mult[W___,X_integer,Y_integer,Z___]:= mult[W,X*Y,Z]
```

Le **calcul équationnel** nous semble permettre la résolution de ce problème, nous allons donc dans la première partie de cette thèse introduire cette forme de calcul et voir ce qu'elle apporte à la simplification mais aussi à l'évaluation.

Problème II

Dans les systèmes de calcul formel on peut voir deux classes d'objets correspondants aux objets mathématiques, les objets qui ont une représentation se rapportant à l'objet mathématique qu'elles représentent et les objets dont la représentation est tellement générale qu'elle peut correspondre à n'importe quel objet mathématique. Par exemple un entier peut être `1,2,...` ou `n,m ...`, une matrice peut être en Mathematica tout simplement `m` ou `m*determinant[m]`, mais aussi `{{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0},} { {0, 0, 0, 1}}`.
Quand l'utilisateur veut définir une règle en contraignant les variables du *pattern* à être tel ou tel objet mathématique il doit tester différemment ces deux formes d'objets.

Exemple 10 *On souhaite définir un opérateur* `pui` *qui correspond à la puissance entière, donc à l'itération du produit* `mult` *dont on connaît des règles de transformation. On utilise Mathematica comme système d'expérimentation, les variables peuvent donc être contraintes par* `X_Integer` *qui signifie que* `X` *ne pourra correspondre qu'à un* `Integer`*. On peut donc déclarer les règles :*

```
In[7]:= pui[pui[X_,N_Integer],M_Integer]:=pui[X,N*M]
In[8]:= mult[pui[X_,N_Integer],pui[X_,M_Integer]] := pui[X,N+M]
```

On obtient bien alors :

```
In[9]:= mult[pui[y,2],pui[y,4]]
Out[9]= pui[y, 6]
```

mais par contre :

```
In[10]:= mult[pui[y,n],pui[y,4]]
Out[10]= mult[pui[y, n], pui[y, 4]].
```

Pour introduire une règle qui ne s'applique que pour les entiers "formels", il faut les caractériser. Pour cela on introduit un opérateur servant de type fictif.

```
In[11]:= pui[pui[X_,N_entier],M_entier]:=pui[X,N*M]
In[12]:= mult[pui[X_,N_entier],pui[X_,M_entier]] := pui[X,N+M]
```

On obtient bien alors :

```
In[13]:= pui[pui[y,entier[n]],entier[m]]
Out[13]= pui[y,entier[n]*entier[m]]
```

mais par contre :

```
In[14]:= pui[pui[y,2],entier[m]]
Out[14]= pui[pui[y,2],entier[m]]
```

Il faut donc déclarer :

```
In[15]:= pui[pui[X_,N_entier],M_Integer]:=pui[X,N*M]
In[16]:= pui[pui[X_,N_Integer],M_entier]:=pui[X,N*M]
In[17]:= mult[pui[X_,N_Integer],pui[X_,M_entier]] := pui[X,N+M]
In[18]:= mult[pui[X_,N_entier],pui[X_,M_Integer]] := pui[X,N+M]
```

et ainsi

```
In[14]:= pui[pui[y,2],entier[m]]
Out[14]= pui[y,2*entier[m]]
```

Mais par contre comme le système ne peut pas deviner qu'une somme d'entiers est un entier

```
In[15]:= pui[pui[pui[y,2],entier[m]],4]
Out[15]= pui[pui[y,2*entier[m]],4]
```

Il faut donc définir une multitude de règles pour simuler une forme de calcul de type afin que Mathematica comprenne que la somme de deux entiers, le produit de deux entiers ... sont des entiers.

Pour éviter cela, nous allons dans la deuxième partie de cette thèse introduire un mécanisme de **typage "formel"** qui permette d'associer à `2` et à `n` le même type, et ainsi de déclarer une unique règle `pui[pui[X_,M_entier],N_entier]`. Ce typage permettra aussi de calculer le type de `2*entier[m]`, donc la règle s'applique pour toute expression entière : `1,2,n,m,n+m,1+m ...`.

Remarque 11 *On peut remarquer qu'en Macsyma ce genre de types est utilisé de façon tout à fait embryonnaire.*
Il est possible par exemple de déclarer qu'un objet `x` *est un scalaire par :*
`declare(x,scalar)`*, ou que ce n'est pas un scalaire par :*
`declare(x,nonscalar)` *([MIT83] chapitre 8 pages 6-7).*
Ces déclarations sont utilisées lors de la simplification.

```
(C1) declare([n1,n2],scalar,[m1,m2],nonscalar);
(D1)                                  DONE
(C2) (n2 * m1) . (n1 * m2);
(D2)                             N2 M1 . N1 M2
(C3) (n2 * m1) . (n1 * m2), dotscrules;
(D3)                            N1 N2 (M1 . M2)
```

Part I

Calcul formel et réécriture équationnelle

Figure 1.2: Transformation de t en t' par remplacement d'égaux par des égaux.

Le calcul équationnel intervient tout naturellement dans les calculs mathématiques, en effet les transformations successives effectuées sur une expression sont le plus souvent dirigées par des équations. Par exemple calculer l'expression $0 + sin(x)$ en $sin(x)$ c'est calculer $0 + sin(x)$ modulo l'équation $0 + X = X$.
Le calcul équationnel est fondé sur le remplacement d'*égaux par des égaux* (voir Figure page 25).

Le calcul équationnel est fondé sur la manipulation d'expressions, nous introduisons la définition des expressions utilisée en général.
On peut trouver dans [DJ90], la définition suivante des expressions en calcul équationnel :
"Une expression est un arbre dont les noeuds sont étiquetés par des symboles d'opérateurs et les feuilles par des symboles de constantes ou des symboles de variables."

Exemple 1 *On peut modéliser les entiers naturels en considérant qu'ils sont construits à partir des symboles* 0 *(zéro) et* S *(successeur). Dans cette modélisation, 2 est représenté par* $S(S(0))$. *La fonction d'addition sur les entiers s'exprime à l'aide des deux équations notées* (1) *et* (2) *:*

$$\begin{array}{lrcl} (1) & x + 0 & = & x \\ (2) & x + S(y) & = & S(x + y) \end{array}$$

On peut remarquer que les symboles x, y *sont des variables alors que* 0 *est une constante. Calculer* $S(S(0)+S(S(0)))+S(S(S(0)))$

modulo ces équations revient à faire les transformations successives suivantes :

$$
\begin{array}{ll}
 & S(S(0)+S(S(0)))+S(S(S(0))) \\
(2) & S(S(S(0)+S(0)))+S(S(S(0))) \\
(2) & S(S(S(S(0)+0)))+S(S(S(0))) \\
(1) & S(S(S(S(0))))+S(S(S(0))) \\
(2) & S(S(S(S(S(0)))+S(S(0)))) \\
(2) & S(S(S(S(S(S(0)))+S(0)))) \\
(2) & S(S(S(S(S(S(S(0)))+0)))) \\
(1) & S(S(S(S(S(S(S(0)))))))
\end{array}
$$

Dans cette partie, nous allons regarder la simplification par règles de transformations ou par propriétés et l'évaluation comme des calculs équationnels. Le premier chapitre présente brièvement la réécriture équationnelle. C'est un mécanisme de calcul dérivé du calcul équationnel par orientation de certaines équations en règles de réécriture . A notre avis, ce mécanisme peut avantageusement remplacer la simplification par règles de transformation généralement utilisée dans les systèmes classiques.
Certaines règles de réécriture sont utilisée extrêmement souvent en calcul formel, nous avons choisi de les définir sous forme de propriétés. Ainsi, nous avons "factorisé" la réécriture par ces règles en des algorithmes de simplification. Le second chapitre défini la simplification par propriétés utilisée dans les systèmes classiques de façon précise.
Enfin, si on considère que les besoins en transformation par des équations sont satisfaits par la réécriture et la simplification (par propriétés), il n'est plus nécessaire que les procédures fassent ce type de calcul. Dans le troisième chapitre, nous définissons l'évaluation comme un processus "faisant ce que la simplification ne fait pas". Il est alors possible d'affiner le mécanisme d'évaluation ainsi que les procédures à définir.

Chapter 1

Réécriture équationnelle

Pour calculer une expression modulo des équations, il faut choisir à chaque étape une bonne équation et l'appliquer dans le bon sens à la bonne sous-expression. Le calcul équationnel peut être facilement automatisé si les équations sont orientées.
La simplification par règles de transformation utilisée dans les systèmes classiques est une forme restreinte d'automatisation du calcul par des équations orientées. Nous allons décrire une forme d'automatisation plus étendue appelée : réécriture équationnelle.

> **Remarque importante 1** *La définition des expressions généralement utilisée ne peut être directement étendue au calcul formel. En effet, les expressions manipulées par les systèmes de calcul formel ne sont pas uniquement composées de symboles, et ne contiennent aucune variable au sens de [DJ90]. Certaines constantes ne sont pas des symboles mais des objets informatiques complexes (listes, vecteurs ...) car elles représentent des objets mathématiques.*

Les expressions manipulées par le calcul formel peuvent être définies comme suit :

Définition 2 *Une* **expression** *encore appelée* **terme** *est un arbre dont les noeuds sont étiquetés par des symboles d'opérateur et les feuilles par des*

constantes symboliques ou non symboliques.
"Le symbole qui étiquette la racine de l'arbre s'appelle **racine** *du terme (ou* **opérateur de tête***). A un sous-arbre correspond alors un* **sous-terme***."*

Notation *On note $\mathcal{O}$ l'ensemble des symboles d'opérateur contenus dans les termes et $\mathcal{C}$ l'ensemble des constantes.*

Dans ce chapitre nous décrivons succinctement un mécanisme général de transformation de termes par des équations orientées : la réécriture équationnelle. Pour en trouver un exposé plus précis et plus complet, le lecteur peut se référer à [DJ90].

1.1 Réécriture

La réécriture est fondée sur la notion de règle de réécriture que nous définissons :

Définition 3 *On distingue un symbole unaire de $\mathcal{O}$ noté V, tel que les termes de racine V sont appelés* **variables de réécriture***.*
Une **règle de réécriture** *est un couple de termes (t_1, t_2) noté $t_1 \to t_2$ tels que les variables de t_2 sont des variables de t_1.*

Exemple 2 *Dans l'exemple précédent, les équations sont orientées*[1] *en deux règles :*

$$\begin{array}{lrcl} (1) & V(x) + 0 & \to & V(x) \\ (2) & V(x) + S(V(y)) & \to & S(V(x) + V(y)) \end{array}$$

où x, y sont des variables de réécriture.

Notation *On note $\mathcal{R}$ l'ensemble des variables de réécriture.*

[1] Voir description de l'ordre plus loin.

Définition 4 *Un ensemble de règles de réécriture est appelé* **système de réécriture***, et le remplacement d'égaux par des égaux en utilisant des règles de réécriture (de la gauche vers la droite) s'appelle* **la réécriture***.*

"**Réécrire** un terme t consiste à choisir un sous terme t' de t et une règle de réécriture *gauche* $\rightarrow$ *droite*, à vérifier que t' est une instance[2] de *gauche* (on dit aussi que *gauche* filtre t') puis à remplacer t' dans t par l'instance correspondance de *droite*."

Dans ce contexte, calculer un terme t consiste à le réécrire jusqu'à l'obtention d'un terme **irréductible** (qui ne peut être réécrit par aucune règle du système) appelé **forme normale** de t. La réécriture est un processus non déterministe : un terme pourra (en général) avoir plusieurs formes normales selon la succession de réécritures effectuées. Si le résultat de la réécriture d'un terme est indépendant du choix de la règle appliquée, le système est dit **confluent**.

De plus il est possible que certains calculs ne terminent pas; si tous les calculs sont finis, on dit que le système possède la propriété de **terminaison**. Les équations sont orientées grâce à un ordre sur les termes[3] pour assurer la terminaison. Une équation de la forme $t_1 = t_2$ où t_1 et t_2 sont des termes est orientée en la règle $t_1 \rightarrow t_2$ si t_1 est "plus grand" que t_2. Intuitivement, si une réécriture augmente la complexité des termes (donc est mal orientée), les calculs peuvent ne pas terminer. Lorsqu'un système termine et est confluent, on a une et une seule forme normale pour chaque terme dite **forme canonique**, et le système est alors dit **convergent** ou **canonique**.

1.2 Réécriture équationnelle

Dans la section précédente, on a supposé toutes les équations orientables, ce n'est bien sûr pas le cas en général. Parmi les équations usuelles, la commutativité $f(x, y) = f(y, x)$ est non orientable, de plus si f est commutatif alors l'équation signifiant l'associativité de f est elle aussi non orientable. Pour prendre en compte ces équations, il faut effectuer de la **réécriture**

[2] Instancier *gauche* ou *droite* consiste à remplacer chaque variable de $\mathcal{R}$ par un terme (sa valeur).

[3] Il existe différents ordres sur les termes que nous ne décrirons pas ici (voir [JL86]).

Figure 1.1: Réécriture équationnelle de t en t'.

équationnelle. Pour plus de précision concernant cette notion de réécriture équationnelle et les problèmes qui lui sont associés, on peut se référer à [PS81].

Soit C une classe d'équivalence de termes modulo les équations non orientées $e_1...e_n$, un terme t de la classe C est **réécrit modulo** $e_1...e_n$ en t' de la classe C' s'il existe un terme de C qui se réécrit en un terme de C' (voir Figure page 30).

La notion de remplacement *d'égaux par des égaux* est différente en réécriture équationnelle de ce qu'elle est en réécriture standard. Pour spécifier cette différence, nous avons définie la notion de **sous-terme généralisé**.

Définition 5 *Soient $e_1...e_n$ les équations modulo lesquelles est calculée la réécriture. Soit t un terme appartenant à une classe d'équivalence C modulo $e_1...e_n$, tout sous-terme d'un terme appartenant à C (donc équivalent à t) est un* **sous-terme généralisé** *de t.*

En effet, il ne suffit plus de réécrire chaque sous-terme modulo chaque règle, mais il faut réécrire chaque sous-terme généralisé du terme modulo les équations par chaque règle.

Remarque 3 *Les équations usuelles modulo lesquelles se fait la réécriture sont l'associativité et la commutativité.*

Exemple 4 *Si on rajoute l'équation de commutativité de l'opérateur $+$ au système de l'exemple précédent :*

$$\begin{array}{lrcl} (1) & x+0 & \rightarrow & x \\ (2) & x+S(y) & \rightarrow & S(x+y) \\ (3) & x+y & = & y+x \end{array}$$

le terme $0+(x+y)$ se réécrit en $x+y$, en effet $0+(x+y) = (0+x)+y$ modulo associativité, $0+x$ est donc un sous-terme généralisé de $0+(x+y)$, or $0+x = x+0$ modulo commutativité donc (1) s'applique.

1.3 Réécriture équationnelle et simplification par règles

Dans les systèmes de calcul formel, la simplification par règles de transformation est limitée d'une part parce qu'elle utilise la même stratégie que la simplification et que l'évaluation : "en profondeur d'abord[4]". Et d'autre part, parce que les *patterns* des règles sont parfois soumis à des restrictions (linéaires en Maple).

Nous avons choisi de remplacer ce mécanisme par un mécanisme de réécriture équationnelle qui est fondé lui sur des connaissances théoriques précises et dont le champ d'action est nettement plus étendu que celui de la simplification par règles. En effet la réécriture équationnelle peut utiliser différentes stratégies et ne limite pas les *patterns* de règles. De plus comme la réécriture est définie modulo des équations, il n'est pas nécessaire de gérer de façon artificielle ces équations dans les règles en introduisant des variables tampon (`X___` en Mathematica), et en définissant plusieurs règles à la place d'une seule.

[4]Ce qui signifie que les sous-termes sont simplifiés (ou évalués) avant le terme qui les contient.

Chapter 2

Spécification de la simplification (par propriétés)

Nous avons remplacé la simplification par règles de transformation par un mécanisme de réécriture équationnelle, il n'existe donc plus qu'une forme de simplification : la simplification par propriétés. Par la suite nous utiliserons le mot simplification au lieu de "simplification par propriétés".

Bien qu'ayant des actions fondamentalement similaires, les deux formes de simplification utilisées dans les systèmes de calcul formel sont nécessaires car elles n'ont pas même but. La simplification par propriétés a pour vocation de faire les transformations standards efficacement, alors que la simplification par règles fait les calculs non standards.

Nous allons utiliser l'ossature théorique solide de la réécriture pour préciser la simplification, car ce mécanisme est peu clair dans les systèmes courants. Nous considérons la simplification comme une forme de "compilation" de la réécriture équationnelle et allons la définir dans cette optique.

Ce chapitre présente les concepts nécessaires à la mise en oeuvre pratique de cette approche, cependant l'exposé de certains détails est renvoyée dans l'annexe A.1 page 105.

2.1 Stratégie, équations et propriétés

Pour compiler la réécriture équationnelle en simplification, il nous faut choisir une stratégie ainsi que des équations et des règles de réécriture particulières.

2.1.1 Stratégie : Stgd

Le but de la simplification est de rendre les termes "plus simples à l'oeil" et non de les mettre en forme canonique, il n'est donc pas aisé à priori de spécifier de façon déductive (comme en réécriture) une stratégie de simplification qui permet d'obtenir la forme voulue dans la plupart des cas.

Afin de perpétuer les habitudes, nous avons choisi d'utiliser la même stratégie que dans les systèmes classiques : **en profondeur d'abord** ce qui signifie que les sous-termes sont simplifiés avant le terme qui les contient. De ce fait, nous supposons que les sous-termes sont simplifiés, la simplification n'est donc plus fondée sur le calcul de sous-termes généralisés, mais de **sous-termes généralisés directs** que nous allons définir.

Définition 6 *On appelle* **sous-terme direct** *d'un terme, un des fils de la racine de l'arbre qui le représente.*

Définition 7 *On appelle* **sous-terme généralisé direct** *(en abrégé Stgd) d'un terme t tout sous-terme direct d'un terme équivalent à t.*

> **Exemple 5** *Si $t = a + (b + (c + d))$, $a + c$ est un sous-terme généralisé direct de t modulo Associativité/Commutativité car $a + (b + (c + d)) =_{AC} (a + c) + (b + d)$ et $a + c$ est un sous-terme direct de $(a + c) + (b + d)$.*

On peut alors dire que simplifier un terme t sachant que ses sous-termes directs sont simplifiés correspond à simplifier l'un de ses sous-termes généralisés directs s_1 en s_2, puis à remplacer s_1 par s_2 dans t et à simplifier le terme ainsi obtenu.

2.1.2 Equations

Pour bien spécifier notre approche, nous dégageons le choix suivant :

Choix I *Nous proposons de considérer la simplification comme un calcul modulo associativité et commutativité.*
On peut justifier ce choix en disant que ce sont les équations les plus couramment utilisées et que les algorithmes de filtrage modulo associativité et commutativité ont été bien définis pour la réécriture.

Proposition 8 *En prenant comme forme normale d'un terme modulo associativité la forme "aplatie", et modulo commutativité la forme "triée" (ses sous-termes directs sont "triés"), l'énoncé suivant définit* **constructivement** *la forme d'un* **sous-terme généralisé direct**.

Soit $op(a_1, \cdots, a_n)$ *le terme initial, l'opérateur d'un sous-terme généralisé direct est op et ses arguments sont :*

- *dans un* **contexte** *associatif/commutatif : un sous-ensemble ordonné de* $\{a_1...a_n\}$
  (par exemple $a + c$ *est un sous-terme généralisé direct de* $c + b + a$*),*

- *dans un contexte associatif : une sous-séquence de* $a_1...a_n$
  (par exemple $m_1 \cdot m_2$ *est un sous-terme généralisé direct de* $m_1 \cdot m_2 \cdot m_3$*),*

- *sinon :* $a_1...a_n$.

Les opérateurs associatifs sont binaire bien que les termes associatifs soient souvent représentés de façon n-aire.

Choix II *Nous choisissons de ne considérer que les sous-termes généralisés directs binaires des termes associatifs.*
Car, par similitude avec la réécriture, simplifier un terme associatif n-aire revient à itérer la simplification binaire.

2.1.3 Propriétés

Lorsqu'on décrit les opérateurs courants par des règles de réécriture, on constate que certains modèles de règles sont souvent utilisés.
Par exemple les règles $+(X, zero) \rightarrow X$ et $*(X, un) \rightarrow X$ sont bâties sur le modèle $f(X, neutre) \rightarrow X$.

Une règle de réécriture est formée de deux termes construits à partir de symboles d'opérateur $\mathcal{O}$, de constantes $\mathcal{C}$ et de variables de réécriture $\mathcal{R}$.
On calcule le **modèle** d'une règle par **abstraction**, c'est à dire par remplacement d'un sous-terme de la partie gauche de la règle par une variable d'un nouveau type : les **variables d'instanciation**.

Notation *On note $\mathcal{I}$ l'ensemble des variables d'instanciation.*

> **Exemple 6** *La règle $+(X, zero) \rightarrow X$ dans laquelle X est une variable de R, zero est une constante de C et $+$ est un opérateur de O, a pour modèle $f(X, neutre) \rightarrow X$ où X est une variable de R et $f, neutre$ sont des variables de $\mathcal{I}$.*

Définition 9 *Nous appellerons* **propriété** *un couple de termes $(f(x_1, \cdots, x_n), g)$ noté $f(x_1, \cdots, x_n) \rightarrow g$ dans lequel $f \in \mathcal{I}$ et les termes $x_1 ... x_n, g$ sont construits à partir de variables de $\mathcal{R}$ ou de $\mathcal{I}$, de symboles d'opérateurs et de constantes.*

> **Remarque 7** *Il est bien évident que plusieurs modèles peuvent correspondre à une même règle, car il y a en général plusieurs façons d'abstraire une règle.*

Si on suppose un ensemble de propriétés défini, les règles de réécriture sont construites à partir de ces propriétés par **instanciation**. Une propriété est instanciée par remplacement de **toutes** les variables de $\mathcal{I}$ par des termes construits à partir de $(\mathcal{O}, \mathcal{C}, \mathcal{R})$. En effet, il faut que le résultat de l'instanciation soit bien une règle de réécriture, il ne doit donc contenir aucune variable de $\mathcal{I}$.

Exemple 8 *Pour construire une règle à partir du modèle* $f(X, neutre) \rightarrow X$*, dont* $f, neutre \in \mathcal{I}$ *et* $X \in \mathcal{R}$*, on peut par exemple instancier* f *par* $+$ *et neutre par zero et obtenir ainsi la règle* $+(X, zero) \rightarrow X$*.*

Lorsqu'on construit une règle de réécriture à partir d'une propriété, on peut remarquer que l'instanciation de la variable notée f joue un rôle particulier. Dans le langage usuel, on utilise le mot propriété pour désigner un modèle d'équation mais aussi les propriétés d'un opérateur. C'est un abus de langage car dans le premier cas il s'agit effectivement d'une propriété, dans l'autre cas il s'agit d'instances de propriétés. Nous perpétuons cet abus de langage par commodité.

Définition 10 *On dit qu'on* **associe la propriété** P **à l'opérateur** *op lorsqu'on instancie* $P = f(x_1, \cdots, x_n) \rightarrow g$ *en donnant à* f *la valeur op. La propriété d'un opérateur est parfaitement définie si on connaît le nom de l'opérateur op, la propriété* P *et les valeurs associées aux variables de* $\mathcal{I} \setminus \{f\}$ *appelées* **paramètres** *(puisqu'elles doivent toutes être instanciées).*

Remarque 9 *Dans les systèmes classiques, les propriétés peuvent représenter plusieurs règles de transformation, les fonctions de simplifications doivent prendre en compte plusieurs propriétés et sont donc parfois obscures.*

2.2 Simplification par une propriété

La réécriture d'un terme par une règle r_1 de modèle P est similaire à celle du terme par une règle r_2 du même modèle P. Au lieu de recalculer dynamiquement la réécriture d'un terme par une règle de modèle P, on a envie de la compiler sous la forme d'un algorithme de simplification par P dont les paramètres sont les variables de $\mathcal{I}$.
Les algorithmes qui représentent la "compilation" de la réécriture d'un terme par une propriété suivant la stratégie "en profondeur d'abord" sont appelés **algorithmes de simplification**.

Nous avons défini la stratégie de simplification, il nous faut donc considérer que la simplification d'un terme par une propriété est équivalente à la réécriture de ce terme à la racine jusqu'à ce qu'il devienne irréductible, donc tant qu'un sous-terme généralisé direct peut être réécrit.

Dans cette section, nous allons décrire la simplification d'un sous-terme généralisé direct du terme t modulo la propriété p en trois étapes :

étape 1 isolement du sous-terme généralisé direct st_0 de t qui peut être réécrit par r,

étape 2 réécriture de st_0 par r en st_1,

étape 3 reconstruction du terme équivalent à t dans lequel st_0 a été remplacé par st_1.

2.2.1 Etape 1

Cette étape a pour but de mettre syntaxiquement en évidence un sous-terme généralisé direct réductible.
Pour chercher un sous-terme généralisé réductible la réécriture utilise des algorithmes de pattern-matching équationnels. Comme nous définissons un algorithme par propriété (donc par règle), le pattern matching est très spécialisé et consiste à rechercher N sous-termes directs du terme ayant des formes particulières.

Si $N = 1$, la partie gauche de la règle est de la forme :

$$\begin{array}{ll} f(X_1...X_{I-1}, Y, X_{I+1}...X_n) & \\ où \ \{X_i\}_1^n \setminus \{X_I\} & \textit{sont libres} \\ \quad X_I = Y & \textit{a une forme définie,} \end{array}$$

pour réécrire : $f(x_1, \cdots, x_n)$, il faut chercher un x_i de la forme Y et mettre en évidence le sous-terme généralisé direct réductible.
Si x_i correspond au modèle Y, le terme $f(x_1, \cdots, x_n)$ est transformé :

en contexte associatif :
en $f(x_1...x_{i-1}, \mathbf{f(x_i, x_{i+1})}...x_n)$ si $I = 1$,
en $f(x_1...\mathbf{f(x_{i-1}, x_i)}, x_{i+1}...x_n)$ si $I = 2$ (uniquement si $i > 1$),

en contexte associatif/commutatif :
en $f(x_1...x_{i-1}, \mathbf{f(x_i, x_{i+1})}...x_n)$,

en contexte standard :
en $\mathbf{f(x_1...x_i...x_n)}$ où $i = I$,

en contexte commutatif :
en $\mathbf{f(x_1...x_i...x_n)}$ où $i \in [1..n]$.

Si $N = 2$, la partie gauche de la règle est de la forme :

$$f(X_1...X_{I-1}, Y, X_{I+1}...X_{J-1}, Z, X_{J+1}...X_n)$$
$$où \begin{array}{ll} \{X_i\}_1^n \setminus \{X_I, X_J\} & \textit{sont libres} \\ X_I = Y \text{ et } X_J = Z & \textit{ont des formes définies,} \end{array}$$

pour réécrire : $f(x_1, \cdots, x_n)$,il faut chercher un x_i de la forme Y et un x_j de la forme Z.
Si x_i correspond au modèle Y, x_j au modèle Z, le terme $f(x_1, \cdots, x_n)$ est alors transformé en :

en contexte associatif, (uniquement si $i < j$)
$f(x_1...x_{i-1}, \mathbf{f(x_i, x_j)}, x_{i+2}...x_n))$ car $j - i = 1$,

en contexte associatif/commutatif,
$f(\mathbf{f(x_i, x_j)}, x_1...x_{i-1}, x_{i+1}...x_{j-1}, x_{j+1}...x_n)$ si $i < j$
$f(\mathbf{f(x_i, x_j)}, x_1...x_{j-1}, x_{j+1}...x_{i-1}, x_{i+1}...x_n)$ si $i > j$

en contexte standard :
$\mathbf{f(x_1...x_i...x_j...x_n)}$ où $i = I$ et $j = J$,

en contexte commutatif :
$\mathbf{f(x_1...x_n)}$ où $i = I$ et $j = J$ ou $i = J$ et $j = I$.

D'après le choix II page 35, les règles associées à un opérateur associatif sont nécessairement binaires. Donc à partir de $N = 3$ il n'y a plus que les cas non associatifs à regarder. Comme dans ces cas le traitement est évident, nous n'allons pas les décrire ici.

2.2.2 Etape 2

Une fois que le sous-terme généralisé st_0 de la bonne forme est rendu apparent, il est réécrit par la règle en st_1 si la partie droite de la règle est $g(Y_1...Y_m)$ et $st_1 = g(y_1...y_m)$.

Il faut alors que st_1 soit simplifié en st_2 pour pouvoir l'injecter dans le terme initial parmi les autres sous-termes qui sont eux simplifiés. En général st_1 n'a aucune raison d'être simplifié. Mais de la même manière qu'on a considéré la partie gauche des règles pour trouver st_0, nous allons examiner leur partie droite pour avoir des informations sur st_1.

Tout d'abord, on peut dire que les sous-termes de st_1 ne sont pas tous simplifiés. Pour simplifier st_1, il faut le destructurer jusqu'à trouver les x_i qui eux sont simplifiés, et simplifier à nouveau à partir de là. Pour chaque règle, on peut préciser la profondeur jusqu'à laquelle il faut simplifier st_1, c'est :

$$S = \max_i \mathit{Profondeur}(X_i, g(Y_1...Y_m))$$
$$\text{où} \quad \mathit{Profondeur}(X_i, g(Y_1...Y_m)) = 0 \quad \text{si} \quad X_i \notin g(Y_1...Y_m).$$

2.2.3 Etape 3

Cette étape a pour objet de construire un nouveau terme en remplaçant dans t, st_0 par st_2.
Tout d'abord, on remplace textuellement st_0 par st_2, comme le résultat doit être réduit modulo Associativité/Commutativité, il faut simplifier l'expression obtenue modulo ces équations.

2.3 Algorithmes de simplification

L'algorithme de simplification par une propriété p est parfaitement défini lorsqu'on connaît non seulement les paramètres N et S de la règle équivalente à p, mais aussi la fonction qui recherche les N termes correspondant aux N patterns, la fonction de construction du résultat de la réécriture et les équations de l'opérateur de tête de sa partie gauche.

Nous allons montrer par deux exemples, comment il est possible de systéma-

tiser la construction de ces algorithmes.

Exemple 10 *Soient $f(X, g(X)) \to zero$ une règle où X, Y sont des variables, zero une constante, les paramètres de l'algorithme sont $N = 2$, et $S = 1$. On se donne* `pattern?` *la fonction qui teste si deux termes correspondent aux patterns X et $g(X)$, l'algorithme de simplification d'un terme t par rapport à cette règle quand f est associatif/commutatif est le suivant :*

`si` $t = f(x_1...x_n)$ `et pattern?`$(x_1..x_n) = (i, j, ())$
`alors`

1. $st_0 = f(x_i, g(x_i))$
2. `construire` $st_1 = zero$
3. `simplifier` $zero$ `à` $N = 0$ `niveaux en` $st_2 = zero$
4. `simplifier` $f(st_2, x_1...x_{k-1}, x_{k+1}...x_{l-1}, x_{l+1}, ...x_n)$ `modulo commutativité en` t'
5. `repartir en 1.` `avec pour terme` $t = t'$

`sinon` $f(x_1, \cdots, x_n)$

Exemple 11 *Considérons la règle $f(X, g(a, Y)) \to h(a, f(X, Y))$ où X, Y sont des variables de pattern, a une constante, les paramètres de l'algorithme sont $N = 1$, et $S = 1$.*
La fonction `pattern?` *teste si un sous-terme direct du terme est de la forme $g(a, y)$ et calcule la substitution. L'algorithme de simplification d'un terme t par rapport à cette règle quand f est associatif est le suivant :*

`si` $t = f(x_1...x_n)$ `et pattern?`$(x_1..x_n) = (i, ((Y, y)))$
`alors`

1. $st_0 = f(x_{i-1}, g(a, y))$
2. `construire` $st_1 = h(a, f(x_{i-1}, y))$
3. `simplifier` $h(a, f(x_{i-1}, y))$ `à` $N = 1$ `niveau en` st_2
4. `si` $st_2 = f(y_1...y_m)$
   `alors repartir en 1.` `avec` $t = f(x_1...x_{i-2}, y_1...y_m, x_{i+1}...x_n)$
   `sinon repartir en 1.` `avec` $t = f(x_1...x_{i-2}, st_2, x_{i+1}...x_n)$

`sinon` $f(x_1, \cdots, x_n)$

Une fois l'ensemble $\mathcal{P}$ des $n_{\mathcal{P}}$ propriétés choisi, la simplification d'un terme par une propriété est parfaitement définie. Cependant un opérateur peut être muni de plusieurs propriétés, il faut donc définir la simplification par plusieurs propriétés. Pour simplifier un terme par plusieurs propriétés, il faut composer ces algorithmes élémentaires.

Soit $simp_1, ..., simp_{n_{\mathcal{P}}}$ l'ensemble des algorithmes de simplification par les propriétés de $\mathcal{P}$, on veut définir un ordre d'application de ces algorithmes qui soit général à tous les termes. On sait que les termes seront simplifiés différemment suivant cet ordre puisqu'en général le système de règles équivalent aux propriétés d'un ensemble d'opérateur n'est pas convergent.

Mais comme la simplification n'a pas pour but de mettre les termes en forme canonique mais sous une forme "plus simple", il ne semble pas intéressant de compléter systématiquement ce système de règles. En effet, lors de la complétion on ajoute des règles dont on contrôle mal l'esthétique, la forme simplifiée avec ces nouvelles règles a alors peu de chances de rester plus simple à l'oeil.

Nous avons donc choisi de considérer la simplification comme une composition d'algorithmes élémentaires telle que, lors de la simplification d'un terme, chaque algorithme n'est utilisé qu'une seule fois. Pour cela, il nous faut définir un ordre sur les propriétés vérifiant :

$$p_1 < .. < p_n \Rightarrow simp(t, \{p_1...p_n\}) = simp_n(simp_{n-1}(...simp_1(t)))).$$

Cet ordre dépend de l'ensemble des propriétés $\mathcal{P}$ et ne peut être défini en toute généralité.
Nous donnons un exemple de choix des propriétés et de définition de l'ordre entre ces propriétés dans l'annexe A.1 page 105.

Chapter 3

Spécification de l'évaluation

Dans les deux chapitres précédents, nous avons spécifié les mécanismes de calcul les mieux adaptés pour faire des transformations pouvant être schématisées par des règles de réécriture.
Nous supposons donc à partir de maintenant que l'évaluation n'a plus besoin de faire ce type de calcul, nous allons donc spécifier les fonctions de calcul ou procédure.

Nous avons choisi de voir la simplification et la réécriture comme des mécanismes équationnels. Par extension on peut aussi considérer l'évaluation comme un mécanisme de remplacement d'égaux par des égaux. Nous allons donc définir le processus d'application de fonctions comme un processus équationnel.

3.1 Fonctions de calcul ou procédures

L'évaluation d'un terme $f(x_1, \cdots, x_n)$ dans les systèmes de calcul formel classiques consiste en l'appel d'une fonction de calcul de nom f qui peut avoir l'un des trois comportements suivants :

1. envoie d'une erreur,

2. transformation descriptible par des règles de réécriture,

3. transformation non descriptible par des règles de réécriture.

Pour illustrer ce propos, considérons un système de calcul formel fictif dans lequel les fonctions de calcul sont écrites en Le-Lisp, la fonction binaire de calcul associée à l'opérateur + s'écrit :

```
(de add (e1 e2)
    (selectq (type-of e1)
             (integer
              (selectq (type-of e2)
                       (integer (add-integer e1 e2))
                       (polynomial (add-int-poly e1 e2))
                       (matrix (error "Je ne peux additionner un
                                       entier et une matrice" ()))
                       (t (if (equal e1 0)
                              e2
              (make-term + e1 e2)))))
             (polynomial ... )
             (matrix ... )
             (t (cond ((equal e2 0) e1)
                      (...
                      (t (make-term + e1 e2))))))
```

Afin que le champ d'action de l'évaluation ne recouvre pas celui de la réécriture (donc celui de la simplification), nous imposons la restriction suivante :

Choix III *Les fonctions de calcul ne doivent pas effectuer de transformations descriptibles par des règles.*
Ce choix se justifie si on considère que la réécriture est le mécanisme le plus à même de faire des transformations par des règles de réécriture.

L'action des fonctions est donc de :

1. renvoyer une erreur,
2. effectuer des transformations non descriptibles par des règles de réécriture.

La fonction binaire de calcul associée à l'opérateur + dans l'exemple précédent doit être modifiée pour correspondre à cette contrainte.
Les morceaux suivants du code précédent sont remplacés par `(make-term + e1 e2)` :

```
(if (equal e1 0)
```

```
    e2
    (make-term '+ e1 e2))


(cond ((equal e2 0) e1)
      ((and (equal (op e1) '+) (equal (op e2) '+))
       (make-term '+ (args e1) (args e2)))
      ((equal (op e1) '+) (make-terme '+ (args e1) e2))
      (...
      (t (make-term '+ e1 e2)))
```

3.2 Evaluation

Nous venons de spécifier le comportement des fonctions de calcul impliquées dans l'évaluation, il nous reste à préciser le mécanisme équationnel d'appel de ces fonctions.

L'évaluation d'un terme une fois ses sous-termes directs évalués consiste en :

1. l'évaluation de l'un de ses sous-termes généralisés directs,
2. la construction du terme dans lequel le sous-terme calculé est remplacé par sa valeur,
3. l'évaluation de ce nouveau terme,

jusqu'à ce que tous les Stgd s'évaluent en eux même.

Dans les contextes non associatifs, un terme n'est composé que d'un seul sous-terme généralisé direct : lui-même, l'évaluation est donc réalisée en une seule étape.
Par contre en associatif, l'évaluation d'un terme se fait en plusieurs étapes. On peut remarquer que si l'évaluation est accomplie modulo associativité, les fonctions de calcul associées à un opérateur associatif sont binaires (choix II page 35).

Proposition 11 *L'évaluation d'un sous-terme généralisé d'un terme est décrite par les séquents suivants en fonction des équations que satisfait son opérateur de tête :*

(ENA) en contexte non associatif

$$\frac{f \mapsto \tilde{f}}{f(x_1, \cdots, x_n) \rightarrow_{eval} \begin{cases} erreur \\ ou\ f(x_1, \cdots, x_n) \\ \tilde{f}(x_1, \cdots, x_n) \end{cases}}$$

(EA) en contexte associatif

$$\frac{\begin{array}{ll} \forall k \in [1..i-1] & y_k = x_k \\ \forall k \in [i+2..n] & y_{k-1} = x_k \\ k = i & y_k = \tilde{f}(x_i, x_{i+1}) \end{array}}{f(x_1...x_i, x_{i+1}...x_n) \rightarrow_{eval_A} \begin{cases} erreur \\ ou\ f(x_1, \cdots, x_n) \\ f(y_1, \cdots, y_{n-1}) \end{cases}}$$

(EAC) en contexte associatif commutatif

$$\frac{\begin{array}{ll} \forall k \in [1..n-2] & y_k \in \{x_1...x_n\} \setminus \{x_i, x_j\} \\ k = n-1 & y_k = \tilde{f}(x_i, x_j) \end{array}}{f(x_1...x_i...x_j...x_n) \rightarrow_{eval_{AC}} \begin{cases} erreur \\ ou\ f(x_1, \cdots, x_n) \\ f(y_1, \cdots, y_{n-1}) \end{cases}}$$

Exemple 12 *Si on reprend les hypothèses de l'exemple du Problème I page 16, l'évaluation de*

$$mult(x, inv(x), 2, 3, z)$$

consiste en la suite d'évaluations dans laquelle $\rightarrow_A$ dénote l'évaluation

d'un Stgd, et $\longrightarrow_A$ celle du terme complet suivante :

$$
\begin{array}{lll}
 & mult(x, inv(x), 2, 3, z) & \\
 & mult(\widetilde{mult}(x, inv(x)), 2, 3, z) & \\
\rightarrow_A & mult(x, inv(x), 2, 3, z) & car \quad \widetilde{mult}(x, inv(x)) = mult(x, inv(x)) \\
 & mult(x, \widetilde{mult}(inv(x), 2), 3, z) & \\
\rightarrow_A & mult(x, inv(x), 2, 3, z) & car \quad \widetilde{mult}(inv(x), 2) = mult(inv(x), 2) \\
 & mult(x, inv(x), \widetilde{mult}(2, 3), z) & \\
\rightarrow_A & mult(x, inv(x), 6, z) & car \quad \widetilde{mult}(2, 3) = 6
\end{array}
$$

$$
\begin{array}{lll}
 & mult(x, inv(x), 6, z) & \\
 & mult(\widetilde{mult}(x, inv(x)), 6, z) & \\
\rightarrow_A & mult(x, inv(x), 6, z) & car \quad \widetilde{mult}(x, inv(x)) = mult(x, inv(x)) \\
 & mult(x, \widetilde{mult}(inv(x), 6), z) & \\
\rightarrow_A & mult(x, inv(x), 6, z) & car \quad \widetilde{mult}(inv(x), 6) = mult(inv(x), 6) \\
 & mult(x, inv(x), \widetilde{mult}(6, z)) & \\
\rightarrow_A & mult(x, inv(x), 6, z) & car \quad \widetilde{mult}(6, z) = mult(6, z) \\
\longrightarrow_A & mult(x, inv(x), 6, z) &
\end{array}
$$

On peut remarquer que $\widetilde{mult}$ n'effectue le calcul que si ses deux arguments sont des entiers, il serait donc plus direct de dire que les seuls Stgd à calculer sont ceux qui contiennent des entiers.

Un évaluateur se compose donc naturellement de trois évaluateurs équationnels :

1. un évaluateur associatif/commutatif,
2. un évaluateur associatif,
3. un évaluateur standard (non associatif).

En adoptant ce point de vue, le mécanisme d'évaluation modulo A, C est "factorisé" dans l'évaluateur au lieu d'être dupliqué dans toutes les fonctions d'opérateur associatif, associatif/commutatif. On peut alors être sûr que l'évaluation respectera l'associativité et la commutativité. De plus l'écriture d'une fonction dont l'opérateur est associatif est grandement facilitée, puisqu'il suffit d'écrire une fonction binaire.

3.3 Calcul d'un terme : réécriture, simplification, évaluation.

En différenciant la simplification et la réécriture de l'évaluation par les transformations qu'ils réalisent, nous clarifions le travail de l'utilisateur. Pour définir un opérateur, il doit lui associer des propriétés, des règles et une fonction de calcul qui sont des informations clairement différentes.
Nous avons généralisé la notion de calcul équationnel pour factoriser les algorithmes de calcul modulo associativité/commutativité. Ainsi les règles et les fonctions de calculs sont plus faciles à écrire, ce qui libère l'utilisateur.
Il nous faut maintenant définir ce que veut dire calculer un terme dans ce contexte.

Exemple 13 *Le calcul que nous avons tenté dans les systèmes classiques et qui correspond au Problème I (décrit page 16) est aisément mené à bien dans notre contexte.*
Il suffit de déclarer que `mult` *est un opérateur associatif, avec un opposé appelé* `inv` *et un élément neutre* `1`*, et qu'il est associé à une fonction qui calcule le produit de ses deux arguments si ce sont des entiers. Sous ces hypothèses le terme peut être transformé comme suit :*

$$\begin{array}{rll} mult(2, inv(2), x, y, 5, 2) & \longrightarrow_{simp_A} & mult(x, y, 5, 2) \\ & \longrightarrow_{eval_A} & mult(x, y, 10) \end{array}$$

Pour nous, la réécriture, la simplification et l'évaluation doivent être tous les trois utilisés puisque chacun à en charge une partie de l'information sur les opérateurs. Lors du calcul d'un terme, il faut donc les combiner pour obtenir le résultat désiré. Mais il est possible de faire de multiples variations sur le résultat d'un calcul en modifiant la combinaison. Il n'est pas possible de choisir "une bonne combinaison" pour tous les termes et tous les utilisateurs possibles. La combinaison de ces mécanismes de calcul doit donc être contrôlée par l'utilisateur.

Part II

Typage des opérateurs et des termes

Les concepteurs de systèmes de calcul formel ont "donné corps" aux objets mathématique à travers des représentations informatiques.
Une grande partie de la recherche en calcul formel a pour but de définir des représentations informatiques d'objets mathématiques en adéquation avec les calculs à effectuer dessus et augmenter ainsi l'efficacité des calculs et donc la taille des expressions manipulées.
Pour chaque objet mathématique, on peut donc trouver dans la littérature une grande diversité de représentations possibles.

Exemple 1 *Le polynôme*

$$XY + 3X + Y^2 + 4Y$$

peut avoir pour "corps" informatique

```
[[X Y] [[1 [1 1]] [3 [1 0]] [1 [0 2]] [4 [0 1]]]]
```

ou

```
(+ (* X Y) (* 3 X) (^ Y 2) (* 4 Y))
```

alors que la matrice

$$\begin{pmatrix} x & y^2 \\ 5 & z^3 \end{pmatrix}$$

peut avoir elle

```
[x y z] [[[[1 [1 0 0]]] [[1 [0 2 0]]]] [[[5 [0 0 0]]]
                    [[1 [0 0 3]]]]]]
```

ou

```
(matrice-ligne-2-2 x (^ y 2) 5 (^ z 3)).
```

Schématiquement, on peut dire que dans les systèmes de calcul formel, les objets mathématiques sont soit représentés de façon spécifique pour permettre des manipulations optimisées, soit tous représentés de la même manière pour permettre une manipulation homogène.
Quand l'utilisateur désire un traitement particulier pour une expression

mathématique, deux moyens s'offrent donc à lui. Soit il choisit une représentation spécifique et donne ainsi au système toutes les informations concernant cette expression, soit il choisit la représentation générale et le seul moyen qui lui reste pour préciser le traitement qu'il désire est d'utiliser des opérateurs spécifiques.

Afin d'unifier ces deux approches, on est conduit à enrichir la représentation générale d'une information supplémentaire : un type et à créer un lien entre ce type et les représentations spécifiques. Dés lors, l'utilisateur pourra préciser le traitement qu'il désire faire subir à une expression en donnant un type formel. Pour éviter à l'utilisateur le travail fastidieux de déclarer tous les types de toutes les expressions, nous introduisons un mécanisme de typage.

Dans le premier chapitre de cette partie, nous étudions les liens entre les représentations spécifiques et les conversions pour définir la nouvelle classe de type à introduire.
Dans le second chapitre, nous décrivons l'accroissement de la puissance de la réécriture et de la simplification qu'apporte le typage.
Dans le dernier chapitre on voit émerger une définition précise de l'évaluation quand elle est précédée par du typage par similitude avec l'interprétation utilisée en théorie des modèles.

Chapter 4

Représentations spécifiques et types formels

4.1 Représentations spécifiques

Choix IV *Nous partageons en deux classes les représentations possibles des expressions mathématiques dans un système de calcul formel : la représentation sous forme d'arbre et les représentations étudiées en fonction d'algorithmes spécifiques.*

Définition 12 *Nous qualifierons la représentation sous forme d'arbre de* **générale** *alors que les autres représentations seront qualifiées de* **spécifiques**.

Dans la définition 1 page 28), nous avons pris en compte dans $\mathcal{C}$ des constantes non symboliques qu'il nous est maintenant possible de préciser.

Définition 13 *Les constantes non symboliques de $\mathcal{C}$ sont des termes dans une représentation spécifique, nous les appellerons* **constantes (ou objets) spécifiés**.

Notation *Dans la suite, on notera $\mathcal{S}$ l'ensemble des représentations spécifiques et $\mathcal{C}_{\mathcal{S}}$ l'ensemble des constantes spécifiées (c.a.d. dans une représentation spécifique).*

Chaque expression mathématique peut être représentée dans une ou plusieurs représentations spécifiques ou dans une représentation générale.
Par exemple, l'expression $x + y$ peut être manipulée dans un système de calcul formel soit (en ayant en tête l'idée de polynôme) comme un arbre (+ x y), soit comme une liste de deux triplets ((1 1 0)(1 0 1)) dont le premier représente x comme $1 * x^1 * y^0$ et le second y comme $1 * x^0 * y^1$.

Dans les systèmes de calcul formel, il y a au plus une représentation générale alors qu'il peut y avoir une ou plusieurs représentations spécifiques par classe d'objets mathématiques.
Ainsi, en `Mathematica` ([Wol88] pages 496-497) il n'y a aucune représentation spécifique, à l'inverse en `Scratchpad/Axiom` ([BBD$^+$91] pages 43-67) il n'y a pas de représentation générale. Le système `Macsyma` utilise une représentation générale `general form` ([MIT83] chapitre 3 pages 1-2) et des représentations spécifiques `number, array, list` ([MIT83] chapitre 2 pages 1-21), il en est de même pour `Maple` et `Reduce`.

4.2 Représentations spécifiques modulo conversions

En mathématique, on utilise naturellement les inclusions d'ensembles pour avoir plusieurs points de vue sur un même objet. Par exemple, tout entier peut être vu comme un rationnel puisque $Z \subset Q$.

Dans les systèmes de calcul formel tous les objets d'un même ensemble mathématique n'ont pas forcément la même représentation spécifique. Pour prendre en compte les inclusions entre les ensembles mathématiques, ils utilisent donc des conversions. Si on suppose que les éléments de Z sont représentés dans R_Z et les éléments de Q dans R_Q, il faut donc définir une fonction qui permet de passer d'un objet dans R_Z à R_Q pour que le système simule les deux points de vue sur les entiers liés à l'inclusion $Z \subset Q$.

Dans les systèmes classiques, les représentations spécifiques sont prédéfinies et en petit nombre, les conversions ne sont donc pas déclarées, mais utilisées implicitement dans les fonctions de calcul, il n'en est pas de même dans

Scratchpad/Axiom.

Notation *On note $s_1 <_c s_2$, la relation d'ordre partiel sur $\mathcal{S}$ s_1 "est convertible en" s_2.*

A partir de cette relation d'ordre sur $\mathcal{S}$, une relation d'équivalence est définie par :

$$s_1 =_c s_2 \Leftrightarrow s_1 <_c s_2 \text{ et } s_2 <_c s_1.$$

Définition 14 *Un* **type formel** *est le nom d'une classe d'équivalence d'éléments de $\mathcal{S}$ associée à la relation $<_c$. Un type formel est donc un nom T associé à un ensemble d'éléments de $\mathcal{S}$ $s_1...s_k$.*

L'ensemble des types formels est l'ensemble des représentations spécifiques quotienté par la relation "$=_c$", c'est à dire$\mathcal{S}/_{=_c}$.

Notation *On note $\mathcal{F}$ l'ensemble des types formels, $T = \{s_1...s_k\}$ signifie que T est le type formel associé aux types $s_1...s_k \in \mathcal{S}$, et $T \rightarrow t$ signifie que t a pour type formel T, autrement dit que t est un représentant de T.*

> **Remarque 2** *Intuitivement, on peut voir un type formel comme un ensemble de représentations spécifiques (équivalentes) d'un même concept mathématique.*
> *Par exemple, si e_1 et e_2 sont deux représentations des entiers, il paraît naturel de déclarer que $e_1 <_c e_2$ et $e_2 <_c e_1$, donc de déclarer que la classe Entier contient $\{e_1, e_2\}$.*

Définition 15 *Deux types formels T_i et T_j sont ordonnés en $T_i <_c T_j$ si et seulement si $\forall s_i, s_j$ tels que $T_i \rightarrow s_i$ et $T_j \rightarrow s_j$ on a $s_i <_c s_j$.*

L'ordre sur $\mathcal{S}$ prolongé aux types formels grâce à la définition précédente n'est pas (en général) total.

Définition 16 *On appelle* **chaîne de conversions** *un sous-ensemble de types formels totalement ordonné (par $<_c$).*

Choix V *Nous ne considérons que les ensembles de types formels partitionnés en chaînes de conversions.*
Si on considère un type formel comme un ensemble de représentations spécifiques d'un même concept mathématique, ce choix se justifie par le fait qu'il existe des classes d'objets mathématiques fondamentalement incomparables.

Exemple 3 *L'ensemble des types formels les plus usuels*

$\{Entier, Rationnel, Polynôme, Fraction\text{-}rationnelle, Matrice, Vecteur\}$

est partitionné en deux chaînes

$Entier <_c Rationnel <_c Polynôme <_c Fraction\text{-}rationnelle$
$Vecteur <_c Matrice.$

Cette partition n'est valable que si on ne veut pas considérer les scalaires comme des matrice-1×1 *ou des vecteur-*1.

4.3 Types formels et termes

Il nous fallait définir un niveau de type "au dessus" des représentations spécifiques, permettant de typer à la fois des termes généraux et des termes spécifiques, nous avons choisi d'utiliser les types formels.
On peut justifier ce choix en disant que ces types peuvent être associés à des objets spécifiés car ils sont définis à partir d'éléments de $\mathcal{S}$, mais aussi à des objets généraux car ils sont abstraits de toute représentation. D'autre part, ils restent assez proches des types de $\mathcal{S}$ pour qu'on puisse connaître la structure de $\mathcal{F}$ et en particulier la relation $<_c$.

Si on suppose que les types formels sont définis en fonction des représentations spécifiques, on peut exprimer plus facilement règles et propriétés.
Une partie du Problème II est ainsi résolue : on définit le type formel `entier` par `entier = {Integer}`, la contrainte `_entier` signifie alors `_entier` pour les termes généraux et `_Integer` pour les termes spécifiques. Ainsi, les règles que nous avions tant de problème à préciser s'écrivent simplement (dans la syntaxe de Mathematica) :

```
pui[pui[X_,N_entier],M_entier]:= pui[X,N*M]
mult[pui[X_,N_entier],pui[X_,M_entier]]:= pui[X,N+M].
```

Mais le Problème II n'est pas entièrement résolu car l'expression

```
pui[pui[y,2*entier[m]],4]
```

n'est pas réécrite. En effet, l'expression `2*entier[m]` n'a pas pour type `entier`. Or, si on sait que `2` et `entier[m]` ont pour type `entier`, il suffit au système de savoir que quand `*` a pour arguments deux `entiers` il rend un `entier`, pour pouvoir calculer le type `entier` pour `2*entier[m]`.

Pour pouvoir résoudre ce dernier problème, il nous faut donc introduire un mécanisme de typage sur les types de $\mathcal{F}$, et en particulier définir la notion de type d'un opérateur (par exemple le type de `*` pourrait être `entier,entier,entier`). Ainsi le système pourra calculer le type de `2*entier[m]` à partir de celui de `2` et `entier[m]` et du type de `*`.

Chapter 5

$\mathcal{F}$-typage, simplification et réécriture

Dans le chapitre précédent, nous avons vu la nécessité d'introduire le $\mathcal{F}$-typage pour permettre une meilleure spécification des règles. Comme nos types sont simples, ce mécanisme de $\mathcal{F}$-typage est aisé à définir et on peut en voir une description sous forme de calcul logique dans l'annexe B.1 page 113.

Pour calculer le $\mathcal{F}$-type d'un terme, il faut connaître le type des opérateurs. Nous allons donc définir ce qu'est le type d'un opérateur. Il est facile de remarquer que chaque opérateur est associé à plusieurs types, par exemple on utilise en général à la fois le produit entre entiers, entre polynômes ... et on le note toujours $*$. Suivant ces types un opérateur est défini par différentes propriétés, différentes règles.

Notation *On note A un ensemble de types, dans notre cas A représente $\mathcal{S}$ ou $\mathcal{F}$.*

Nous définissons le type d'un opérateur de façon général, c'est à dire sans tenir compte des caractéristiques de l'ensemble $\mathcal{F}$.

Définition 17 *On appelle A-**type fonctionnel** tout type de la forme*

$$\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$$

où $\forall i \in [1..n+1]\ \alpha_i \in A$. *Le type produit* $\alpha_1 \times ... \times \alpha_n$ *est appelé* **type initial**, *alors que* α_{n+1} *est appelé* **type final**.

Notation *Par la suite nous identifierons produits cartésiens de types et séquences de types, et noterons donc* $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$ *comme* $\alpha_1..\alpha_n \rightarrow \alpha_{n+1}$.

Proposition 18 *Nous avons spécifié les différentes formes de A-types fonctionnels en fonction de l'arité ainsi que des équations de l'opérateur auquel ils sont associés.*
Voici ces différentes formes :

1. *arité n :* $\alpha_1...\alpha_n \rightarrow \alpha_{n+1}$
2. *arité n/commutatif :* $\alpha_1...\alpha_1 \rightarrow \alpha_2$
3. *arité variable :* $\alpha_1...\alpha_1 \rightarrow \alpha_2$
4. *associatif*[1] *:* $\alpha_1\alpha_2 \rightarrow \alpha_1$ *ou* α_2
5. *associatif/commutatif :* $\alpha_1\alpha_1 \rightarrow \alpha_1$.

Un opérateur est donc associé à plusieurs $\mathcal{F}$-types fonctionnels des cinq formes décrites précédemment.

Définition 19 *On appelle* **opérateur générique** *un symbole d'opérateur associé à plusieurs* $\mathcal{F}$*-types fonctionnels, et* **opérateur** $\mathcal{F}$**-typé** *un symbole d'opérateur associé à un unique* $\mathcal{F}$*-type fonctionnel.*

> **Remarque 4** *Pour éviter toute ambiguïté, nous considérons que deux* $\mathcal{F}$*-types fonctionnels d'un même opérateur n'ont pas même type initial.*

[1] Un opérateur associatif est défini comme un opérateur binaire d'après le choix II page 35, mais les informations binaires (règles, fonctions) sont utilisées de façon itérative.

Un opérateur générique peut se voir comme un ensemble d'opérateurs $\mathcal{F}$-typés ayant même symbole d'opérateur. Lors du typage, chaque opérateur générique f contenu dans les termes est remplacé par l'opérateur typé adéquat $f, \alpha_1..\alpha_n \rightarrow \alpha_{n+1}$.

5.1 Description du $\mathcal{F}$-type d'un terme

En calcul formel, il est courant d'introduire dans les termes des symboles qui n'ont pas été définis, notamment des symboles de constantes car ils représentent souvent des noms d'objets. On peut difficilement supprimer cette liberté, il faut donc non seulement vérifier le type des termes, mais aussi inférer les types des symboles indéfinis.

En même temps que le type du terme, il faut donc calculer un **environnement** qui contient les types inférés pour ces symboles.
Le type d'un terme n'est alors pas un élément de $\mathcal{F}$, mais une formule plus compliquée qui peut toujours être représentée par une formule logique construite à partir de l'opérateur ":" qui signifie "a pour type" et des connecteurs logiques $\wedge, \vee, \Leftrightarrow$.

Exemple 5 *Supposons* $+ : EE \rightarrow E$ *et* $+ : PP \rightarrow P$*, le terme* $(a + b)$ *est typé* E *si et seulement si* a *est typé* E *et* b *est typé* E *et* $(a + b)$ *est typé* P *si et seulement si* a *est typé* P *et* b *est typé* P*.*
Le type de $(a + b)$ *peut donc être décrit par la formule logique suivante :*

$$[a : E \wedge b : E \Leftrightarrow (a + b) : E] \bigwedge [a : P \wedge b : P \Leftrightarrow (a + b) : P].$$

Pour la clarté de l'exposé, nous allons supposer que les seuls symboles non typés sont les symboles de constantes. Dans ce cas, un environnement est représenté par une formule logique de la forme $\wedge_j (z_j : \alpha_j)$ où z_j est un symbole de constante (les z_j sont deux à deux différents) et α_j un type atomique.

Exemple 6 *Dans l'exemple précédent, les environnements sont*

$$a : E \wedge b : E \text{ et } a : P \wedge b : P.$$

Un terme pouvant avoir le même type dans plusieurs environnements, son type est donc de la forme :

$$\bigwedge_j \left[\bigvee_k E_{jk} \Leftrightarrow x{:}\alpha_j \right] \text{ avec } \alpha_j \in A \text{ et } \{E_{jk}\} \text{ des environnements,}$$

ou

$$faux.$$

On décide que dans un environnement un terme ne peut avoir qu'un seul type, les $\{E_{jk}\}$ sont donc deux à deux différents.
Le $\mathcal{F}$-typage est un mécanisme de vérification de types, mais aussi d'inférence de type et traite les termes "en profondeur d'abord". Nous ne décrivons pas ici le $\mathcal{F}$-typage qui calcule le type d'un terme à partir des types de ses sous-termes directs et de son opérateur, car comme la structure de l'ensemble $\mathcal{F}$ est simple, la méthode de typage l'est aussi. On donne une description de cette méthode par un calcul logique dans l'annexe B.1 page 113.

5.2 Extension de la simplification et de la réécriture

On suppose maintenant que chaque opérateur générique se compose d'opérateurs typés chacun munis de propriétés compatibles entre elles, ou de règles de réécriture.
Pour simplifier (respectivement réécrire) un terme, il faut donc résoudre la surcharge de son opérateur de tête, puis remplacer dans le terme le nom de l'opérateur générique par celui d'un de ses opérateurs typés, il ne reste plus alors qu'à utiliser les propriétés (règles) de ce nouvel opérateur pour le simplifier (réécrire).

Remarque 7 *Pour la clarté des exemples, on décide de donner un nom à chaque couple opérateur $\mathcal{F}$-type fonctionnel. D'après la remarque 4, deux types fonctionnels d'un même opérateur générique ne peuvent avoir même type initial, on n'utilise donc que les initiales des $\mathcal{F}$-types composants le type initial pour construire le nom de l'opérateur typé. Par exemple, $*_{sm}$ désignera l'opérateur $\mathcal{F}$-typé associé à l'opérateur générique $*$ et dont le type est $scalaire, matrice \rightarrow matrice$.*

Exemple 8 *Si n est un scalaire, le terme*

$$trace(m) * n * 2 * m$$

est transformé par typage en

$$\left[\ (((trace(m) *_s n) *_s 2) *_{sm} m) : matrice \quad , \quad ((n : scalaire)(m : matrice))\ \right]$$

où $*_s$ correspond au produit sur les scalaires, $*_m$ au produit sur les matrices et $*_{sm}$ au produit scalaire/matrice. Ce terme peut être transformée par simplification en

$$\left[\ ((2 *_s n *_s trace(m)) *_{sm} m) : matrice \quad , \quad ((n : scalaire)(m : matrice))\ \right].$$

Malheureusement le typage ne permet pas toujours de résoudre totalement l'ambigüité due à la surcharge.

Exemple 9 *Si n est indéfini, la forme typée du terme*

$$2 * trace(m) * n * m$$

est

$$\begin{cases} ou & [\ (((2 *_s trace(m)) *_s n) *_{sm} m) : matrice \quad , \quad ((n : scalaire)(m : matrice))\ ] \\ & [\ (((2 *_s trace(m)) *_{sm} n)) *_m m) : matrice \quad , \quad ((n : matrice)(m : matrice))\ ] \end{cases}$$

et sa forme simplifiée est

$$\begin{cases} ou & [\ ((2 *_s n *_s trace(m)) *_{sm} m) : matrice \quad , \quad ((n : scalaire)(m : matrice))\ ] \\ & [\ ((2 *_s trace(m)) *_{sm} (n *_m m)) : matrice \quad , \quad ((n : matrice)(m : matrice))\ ] \end{cases}$$

Quand le système n'arrive pas à résoudre de façon unique la surcharge de l'opérateur générique d'un terme, ce terme représente une disjonction de termes typés.
En calcul formel on imagine mal de rendre à l'utilisateur comme résultat du calcul d'un terme générique une conjonction de terme typés. On représente donc la conjonction de ces termes comme un terme dont les noeuds sont soit des symboles d'opérateur, soit des listes de symboles d'opérateurs.

Exemple 10 *On remplace donc l'expression typée précédente par le couple :*

$$\left[\begin{array}{l}(((2 *_s trace(m))[*_s, *_{sm}]n)[*_{sm}, *_m]m) : matrice, \\ [((n : scalaire)(m : matrice))((n : matrice)(m : matrice))]\end{array}\right]$$

Les seules simplifications (où réécritures) autorisées sur de telles expressions sont celles qui sont communes à tous leurs termes typés correspondants. Ces transformations communes sont extrêmement difficiles à déterminer automatiquement si beaucoup d'opérateurs typés ayant beaucoup de propriétés (règles) différentes sont en jeu. De toute façon, il y a en général très peu de transformations communes, on peut donc dire que ces termes ne peuvent être ni simplifiés ni réécrits.

Si les règles de réécriture sont déclarées sous l'interprète (comme en Mathematica, Macsyma, Maple), elles sont elles aussi typées. De ce fait, chaque règle générique représente une disjonction de règles typées. Réécrire un terme par une règle signifie donc réécrire une disjonction de termes typés par une disjonctions de règles typées, ceci n'est possible que si une règle typée réécrit tous les termes typés.

Remarque 11 *On peut remarquer que l'introduction du typage amène une contrainte sur la réécriture due à la cohérence :* **un terme ne peut être remplacé par un autre par réécriture que si ces deux termes ont même type**.
Les règles de réécritures ne sont donc valides que si le terme de gauche et le terme de droite ont même type. De même, les règles de réécritures équivalentes aux propriétés associées à un opérateur doivent être valides.

5.3 Calcul d'un terme : $\mathcal{F}$-typage, réécriture, simplification et évaluation

Le $\mathcal{F}$-typage correspond à un traitement préalable du terme, permettant d'en préciser automatiquement la forme ce qui évite à l'utilisateur d'avoir à le faire.

Nous avons ajouté un quatrième mécanisme au calcul formel qui se compose donc maintenant de :

1. $\mathcal{F}$-typage,
2. simplification,
3. évaluation,
4. réécriture.

Le $\mathcal{F}$-typage est utilisé comme pré-traitement des termes, il faut donc l'utiliser avant chacun des trois autres mécanismes. C'est la seule contrainte concernant la combinaison de ces mécanismes de calcul.

En considérant ce modèle, on peut résoudre entièrement le Problème II décrit dans l'introduction de cette thèse (page 19). Nous avons vu dans la conclusion du chapitre précédent (page 59) que les règles pouvaient être décrites simplement, mais qu'il manquait un mécanisme capable de calculer le type de termes de la forme `2+entier[n]`.

Exemple 12 *Dans un système fictif ayant la même syntaxe que Mathematica, agissant suivant notre modèle, sous les hypothèses suivantes :*

$$
\begin{aligned}
&entier = \{Integer\}\\
&*_e : (entier, entier \rightarrow entier, commutatif)\\
&+_e : (entier, entier \rightarrow entier, commutatif)\\
&n :\rightarrowtail entier\\
&m :\rightarrowtail entier
\end{aligned}
$$

les calculs se feraient comme suit :

```
<fictif> pui[pui[X_,N_entier],M_entier]:=pui[X,N*M]
<fictif> pui[pui[X_,N],M]:=pui[X,N *e M]

<fictif> mult[pui[X,N_entier],pui[X,M_entier]] := pui[X,N+M]
<fictif> mult[pui[X,N],pui[X,M]] := pui[X,N +e M]

<fictif> mult[pui[pui[pui[y,2],entier[m]],4],pui[y,entier[n]]]
<fictif> pui[y,(2 *e 4 *e m) +e n]
```

L'introduction du $\mathcal{F}$-typage dans le cycle de calcul d'un terme nous a permis d'augmenter l'expressivité des règles de réécriture et des propriétés et de rapprocher le calcul symbolique réalisé automatiquement du calcul symbolique réalisé à la main.

Chapter 6

$\mathcal{F}$-typage et évaluation

En introduisant le typage comme traitement préalable à la simplification et à la réécriture, on influence aussi l'évaluation.
Le $\mathcal{F}$-typage permet de valider les termes, donc si il y a cohérence entre les $\mathcal{F}$-types fonctionnels d'un opérateur et sa fonction de calcul, seuls les termes valides seront évalués. Les fonctions de calcul n'ont donc plus à provoquer d'erreur de typage, puisqu'un autre mécanisme s'en charge.

6.1 Fonctions de calcul ou procédure

Dans la partie initiale de cette thèse, nous avons décrit l'action des fonctions de calcul comme pouvant être :

1. renvoyer une erreur,
2. reconstruire le terme initial,
3. effectuer des transformations non descriptibles par des règles de réécriture,

on peut maintenant limiter l'action des fonctions de calcul à :

1. reconstruire le terme initial,

2. effectuer des transformations non descriptibles par des règles de réécriture.

Mais dans l'état actuel des choses, il n'est pas possible de déterminer si il y a ou non cohérence entre la fonction de calcul et les $\mathcal{F}$-types fonctionnels d'un opérateur.
Pour pouvoir vérifier cette cohérence, il est nécessaire de connaître le type fonctionnel de la fonction de calcul.

Choix VI *On considère que seuls les termes dont tous les sous-termes appartiennent à $\mathcal{C}_\mathcal{S}$ peuvent être évalués.*
Ce choix se justifie si on assimile l'évaluation à l'interprétation utilisée en calcul des modèles car un terme ne peut être interprété que si tous ses sous-termes le sont déjà. De plus, si on considère que les représentations de $\mathcal{S}$ sont spécifiées par rapport aux fonctions de calcul, il est clair que cela augmente l'efficacité de l'évaluation.

D'après le choix précédent, les types fonctionnels des fonctions de calcul sont des $\mathcal{S}$-types fonctionnels.
Mais on remarque que les fonctions de calcul sont en général génériques et donc définies par plusieurs $\mathcal{S}$-types.

Exemple 13 *Le type de la fonction* `add` *définie comme suit :*

```
(de add (e1 e2)
        (selectq (type-of e1)
                 (integer
                  (selectq (type-of e2)
                           (integer (add-integer e1 e2))
                           (polynomial (add-int-poly e1 e2))
                           (matrix
                            (error "Je ne peux additionner un
                                        entier et une matrice" ()))
                           (t (make-term + e1 e2))))
                 (polynomial ... )
                 (matrix ... )
                 (t (make-term + e1 e2))))
```

est :

$$ou \begin{cases} integer, integer \rightarrow integer \\ integer, polynomial \rightarrow polynomial \\ polynomial, integer \rightarrow polynomial \\ polynomial, polynomial \rightarrow polynomial \\ matrix, matrix \rightarrow matrix \end{cases}$$

La définition d'un opérateur qui se compose de $\mathcal{S}$-types et de $\mathcal{F}$-types est alors correcte si chacun des $\mathcal{S}$-types correspond à un des $\mathcal{F}$-types : ainsi le $\mathcal{F}$-typage ne rejettera pas de termes qui auraient pu être évalués.

Une fois la fonction de calcul d'un opérateur générique décrite par des $\mathcal{S}$-types, il ne reste plus qu'un pas à franchir pour préciser l'évaluation. Il suffit de décomposer la fonction de calcul en plusieurs fonctions de calcul élémentaires correspondants aux $\mathcal{S}$-types fonctionnels. Toute fonction de calcul correspond donc à un ensemble de couples $\mathcal{S}$-type/fonction de calcul élémentaire.

Définition 20 *On appelle* **interprétation** *tout couple formé d'un $\mathcal{S}$-type fonctionnel $s_1..s_n \rightarrow s_{n+1}$ et d'une fonction f de calcul élémentaire d'arité n. La fonction f prend en entrée n constantes de $\mathcal{C}_\mathcal{S}$ de types $s_1..s_n$ et rend une constante de $\mathcal{C}_\mathcal{S}$ de type s_{n+1}.*

Exemple 14 *L'opérateur + qui était décrit par la fonction* `add` *et ses types peut donc être défini par l'ensemble d'interprétations suivant :*

$$\begin{cases} (integer, integer \rightarrow integer \ , \ add\text{-}integer) \\ (integer, polynomial \rightarrow polynomial \ , \ add\text{-}int\text{-}poly) \\ (polynomial, integer \rightarrow polynomial \ , \ add\text{-}poly\text{-}int) \\ (polynomial, polynomial \rightarrow polynomial \ , \ add\text{-}polynomial) \\ (matrix, matrix \rightarrow matrix \ , \ add\text{-}matrix) \end{cases}$$

Un opérateur générique est composé non seulement d'opérateurs $\mathcal{F}$-typés munis de propriétés, mais aussi d'interprétations.

6.2 Interprétation d'un sous-terme généralisé direct

> **Remarque importante 21** *En théorie des modèles, un terme est* **interprété**[1] *dans un domaine. L'interprétation d'un terme dans le domaine D, correspond à l'association d'une fonction de D^n dans D à chaque symbole de constante ($n = 0$) ou d'opérateur n-aire (on peut voir [CR73] pour plus de détails).*
> *Dans notre cadre, les symboles d'opérateurs sont aussi associés à des couples (type fonctionnel, fonction de calcul). De plus, on peut voir la présence de constantes non symboliques dans les termes comme une interprétation initiale de certaines constantes symboliques. On peut donc assimiler évaluation et interprétation. Cependant, il existe une différence entre l'évaluation et l'interprétation : l'évaluation ne se fait pas dans un domaine connu à priori. Il nous faut donc choisir une interprétation parmi plusieurs lors de l'évaluation.*

Chaque opérateur est défini par une liste d'interprétations, l'évaluation d'un Stgd dont les sous-termes sont spécifiés commence donc par le choix d'une interprétation. Nous étendons l'interprétation définie en logique à la recherche d'une interprétation parmi plusieurs.

Une interprétation est choisie par $\mathcal{S}$-typage (voir l'annexe B.1 page 113), mais comme tous les objets spécifiés sont typés, le $\mathcal{S}$-typage se réduit à la vérification de type.
Une fois l'interprétation choisie, il ne reste plus qu'à appliquer la fonction de calcul qu'elle contient.

Le vérificateur de $\mathcal{S}$-types travaille modulo conversions, il n'est donc pas obligatoire de définir toutes les fonctions élémentaires. Mais si on suppose que le vérificateur de type peut choisir une interprétation qui correspond modulo conversions au terme, il faut convertir tous les sous-termes avant d'appliquer la fonction de calcul. En effet, les fonctions élémentaires s'attendent à ce que les arguments qui leur sont passés soient des objets spécifiés décrits par les $\mathcal{S}$-types.

Ce modèle d'évaluateur se compose donc :

[1] A ne pas confondre avec la définition d'interprétation précédente.

1. d'un vérificateur de types ($\mathcal{S}$-typage),

2. d'un mécanisme de conversion,

3. d'un mécanisme d'application de fonctions de calcul élémentaires.

Les termes à $\mathcal{S}$-typer sont spécifiés (voir choix VI page 70), il ne contiennent donc aucun symbole non typé, leurs environnement de typage sont donc vides. Le $\mathcal{S}$-type d'un terme $f(x_1, \cdots, x_n)$ est donc soit $faux$, il n'est alors pas possible de l'interpréter, soit de la forme $f(x_1, \cdots, x_n) : s_1 \Leftrightarrow \emptyset$. Si le type résultat est $f(x_1, \cdots, x_n) : s_1 \Leftrightarrow \emptyset$ il peut y avoir modulo conversion plusieurs interprétations possibles du terme $f(x_1, \cdots, x_n)$. Mais comme toutes ces interprétations donnent le même résultat (même "valeur" et même type), n'importe laquelle de ces interprétations est correcte.

> **Remarque 15** *On peut vouloir appliquer la fonction la "moins chère". Le coût de l'application d'une fonction ne peut être calculé à priori car seule la personne qui a écrit la fonction est à même de le faire. Donc on ne peut choisir la fonction à appliquer que par le coût des conversions.*
> *Si on considère que la distance entre deux types est proportionnelle à la complexité de la fonction de conversion, donc au coût de la conversion d'un objet d'un type à l'autre, le coût en conversion de l'application d'une interprétation de type initial $s'_1..s'_n$ sur un terme de type numérique $s_1..s_n$ est $d_c(s_1..s_n, s'_1..s'_n)$. On choisit la fonction à appliquer en minimisant cette distance.*

Ce modèle d'évaluateur possède deux avantages : la sécurité et l'incrémentalité. Il est plus sûr car on connaît le type des fonctions, on peut donc vérifier que le type final déclaré est bien le type obtenu par calcul (ou tout au moins un type plus petit), mais on peut aussi vérifier la convergence des $\mathcal{S}$-types de chaque opérateur (voir l'annexe B.2 page 117).
De plus, la décomposition de chaque fonction de calcul en fonctions de calcul élémentaires permet un mode de déclaration incrémental très important pour l'utilisateur.

6.3 Simplification, réécriture et évaluation

Nous avons défini le calcul d'un terme comme la composée du $\mathcal{F}$-typage, de la simplification, de l'interprétation et de la réécriture.

Ce faisant, nous avons précisé les informations nécessaires au calcul d'un terme :

1. propriétés,
2. règles,
3. $\mathcal{S}$-types, conversions et $\mathcal{F}$-types,
4. $\mathcal{F}$-types fonctionnels,
5. interprétations.

Ces informations nous semblent former le support du calcul formel et permettent d'exprimer une bonne partie de la connaissance à priori qui est utilisée lors d'un calcul à la main.

De plus, nous avons été amenés à préciser le sens de la simplification, de la réécriture et de l'évaluation en considérant que ces trois mécanismes étaient complémentaires, donc que leurs champs d'action se devaient d'être différents.

> La simplification et la réécriture agissent sur les termes en représentation générale en faisant des transformations descriptibles par des règles de réécriture.

> L'évaluation agit sur une sous-classe de ces termes dont tous les sous-termes directs sont dans des représentations spécifiques en faisant des transformations non descriptibles par des règles de réécriture.

Part III

Ulysse : Un nouvel outil de calcul formel

Notre but initial était de remplacer le simplificateur du système de calcul formel Sisyphe ([GGP90]), en développement dans le projet SAFIR (INRIA et Université de Nice-Sophia Antipolis) par un simplificateur paramétré. Mais notre réflexion nous a conduit à concevoir d'abord une extension de Sisyphe puis un nouveau système Ulysse qui possède sa cohérence propre et n'utilise plus que la bibliothèque de programmes de Sisyphe.

Pour décrire notre prototype, il est utile de décrire brièvement Sisyphe : il ressemble à Macsyma en ce qui concerne sa syntaxe et ses fonctionnalités, mais possède au moins deux caractéristiques essentielles qui en font un système moderne.

Une première caractéristique vient du fait que ce système a été conçu pour effectuer des calculs optimisés sur les polynômes à plusieurs variables (calcul de base standard) et les matrices.
Les représentations de ces objets ont donc été affinées : tout objet de ces types se compose d'une valeur et de l'anneau dans lequel il vit. Les anneaux sont hiérarchisés suivant leurs propriétés algébriques : ainsi dans chaque anneau A, on trouve les fonctions valides sur les objets de A. Comme chaque objet "porte" son anneau, il porte aussi les fonctions de calculs qui lui sont attachées ce qui facilite les calculs.
Par exemple, lorsque le système calcule `pol(x+y)+pol(x-y)`, il construit les deux polynômes comme `(A,x+y)` et `(A,x-y)`, puis cherche la fonction qui calcule la somme dans l'anneau `A` notée `+(A)` et l'applique à `(A,x+y)` et `(A,x-y)` comme ceci :
`applique(+(A),(A,x+y),(A,x-y))`.

Une deuxième caractéristique est que les expressions de Sisyphe ne sont pas simplifiées comme dans Macsyma mais compilées "au vol", le langage de commande de Sisyphe est donc muni d'un compilateur.
Le calcul des expressions est donc dirigé par le compilateur. Si celui-ci connaît les types des arguments, il appelle la bonne fonction de calcul. Le simplificateur n'est appelé que si le compilateur ne trouve pas de fonction (soit parce qu'il ne connaît pas l'opérateur, soit parce que les arguments sont des symboles ou des expressions non calculables), il est donc relativement peu utilisé.
Ce simplificateur ne réalise que des manipulations purement symboliques puisque le typage par les représentations spécifiques et l'évaluation sont faits avant que l'expression ne lui soit soumise, il n'applique donc aucune fonction de calcul. Il se compose uniquement d'un simplificateur spécialisé par

opérateur courant $(+, *, -, /, log, abs...)$, les termes qui ont pour opérateur de tête ces opérateurs sont donc simplifiés alors que les autres ne sont pas modifiés.
La possibilité d'inhiber le compilateur est offerte à l'utilisateur, c'est alors le simplificateur qui prend le relais et il n'y a plus aucun appel de fonction de calcul.
On peut voir que Sisyphe est construit d'après une "philosophie" de la conception des systèmes différente de celle qui a émergé de notre réflexion, nous avons donc défini un autre système Ulysse.

Plus précisément, l'architecture logicielle d'Ulysse prolonge la vision dissociée du calcul formel que nous avons décrite dans les parties précédentes de cette thèse. Cette vision se caractérise par quatre formes de manipulations sur les termes : le typage, la simplification, la réécriture et l'évaluation, qui sont chacune codées dans une fonction spécialisée.

Dans les deux premières parties de cette thèse nous avons aussi spécifié les types d'informations qui leur sont nécessaires : les chaînes de conversions, les propriétés, les règles, les fonctions de calcul élémentaire, les $\mathcal{S}$-types fonctionnels et les $\mathcal{F}$-types fonctionnels.
A notre avis, l'utilisateur doit pouvoir accéder et modifier facilement ces informations pour pouvoir personnaliser le calculateur formel. Ceci nous semble primordial à partir du moment où le concepteur d'un système ne peut connaître à priori toutes les utilisations postérieures de ce système.

Choix VII *Nous choisissons de différentier explicitement les informations : propriétés, règles, interprétations, types ..., des mécanismes de calcul : simplification, réécriture, évaluation, $\mathcal{F}$-typage ...*
De cette manière la gestion de chaque type d'information est concentrée dans un composant logiciel spécialisé, donc efficace au lieu d'être répartie (comme en Maple) dans chaque définition d'opérateur.

Définition 22 *Les informations nécessaires au calcul sont mémorisées dans une* **base** *(ensemble de données évolutives) alors que les composants dirigés par un* **contrôleur** *forment le* **moteur**.

L'introduction de la notion de base n'a de sens que si la base courante utilisée par le moteur peut être modifiée. Un terme est alors calculé dans un

environnement et est donc valide uniquement dans cette base ou dans une base compatible avec celle-ci.
Pour pouvoir définir la compatibilité entre la base courante et une autre base, il faut donc garder un **historique** des différences positives (ajouts) et négatives (suppressions) entre les différentes bases utilisées. Chaque terme calculé est alors associé à une version d'une base dans un historique.

Chapter 7

Description du prototype

Le calculateur d'Ulysse se compose d'un moteur (quatre calculateurs élémentaires et un contrôleur) et d'une base. Dans le prototype, nous avons cherché à être aussi proches que possible de la description formelle. Ce logiciel est constitué de modules Le-Lisp aussi indépendants que possible. L'un de ces modules est dévolu à la gestion de la base, construction et modification, il y en a un ou plusieurs pour chaque calculateur.

7.1 Moteur

Le moteur qui se compose de quatre calculateurs et d'un contrôleur prend en entré un terme et rend un couple formé du terme calculé et de l'adresse de la base dans l'historique.

7.1.1 La structure des termes

Le moteur manipule des termes qui se composent d'un opérateur, d'un type et d'une liste de sous-termes.
Nous avons implémenté les termes comme des objets structurés de type `term` contenant un nom d'opérateur, un type et un vecteur d'arguments.
Les `term`s constants ont pour vecteur d'arguments le vecteur vide et comme opérateur soit un nom d'opérateur, soit un objet spécifié.

Les vecteurs d'arguments sont typés et ces types sont hiérarchisé suivant les propriétés équationnelles et l'arité de l'opérateur.
De cette manière, nous avons pu d'une part structurer les listes d'arguments en fonction de ces types et d'autre part organiser le code. Voici la forme des listes d'arguments suivant les types :

`:arguments:a`,`:arguments:a:c` associatif (`a`) ou associatif commutatif (`a:c`) : une liste de couples $(n.t)$ où n est le nombre d'occurences successives du terme t dans la liste des arguments,

`:arguments:af`,`:arguments:af:c` d'arité fixe (`af`) ou arité fixe commutatif (`af:c`) : un vecteur de termes,

`:arguments:af:n`,`:arguments:af:n:c` arité variable (`n`) et arité variable commutatif (`n:c`) : une liste de termes.

7.1.2 $\mathcal{F}$-typage, simplification, réécriture et évaluation

La simplification, l'évaluation et le typage suivent la même stratégie : ils calculent en profondeur d'abord. Les fonctions de simplification, d'évaluation et de typage calculent donc "à un niveau", c'est à dire en supposant les sous-termes directs du terme déjà simplifiés (respectivement évalués, typés). Nous décrivons dans la section suivante la façon dont ces fonctions sont combinées pour calculer les termes.

Pour éviter la construction de trop d'objets intermédiaires les quatre calculateurs modifient "en place" leur argument.

Le $\mathcal{F}$-typage

Nous avons défini trois fonctions de typage dépendants des équations, qui suivent essentiellement le calcul logique décrit dans l'annexe B.1 page 113. Le typage associatif est codé comme un typage binaire avec parenthésage à gauche car nous supposons que l'ensemble des $\mathcal{F}$-types de chaque opérateur est convergent.

La structure des vecteurs d'arguments et des types fonctionnels dépend des équations mais aussi de l'arité de l'opérateur, les fonctions de typage utilisent

donc des méthodes de mise en correspondance (match) différentes suivant les types de vecteurs d'arguments et de types fonctionnels.

Le type d'un terme est représenté par une une liste de couples `type/contexte` où chaque `type` est un symbole et chaque `contexte` une liste de couples `op/type` où `op` est un symbole (le nom d'un opérateur constant).

Exemple 1 *Si* $+ : EE \to E$ *et* $+ : PP \to P$*, le type de* $(a + b)$ *est :*

```
((E ((a E)(b E))) (P ((a P)(b P)))).
```

Pour des raisons pratiques, nous avons autorisé l'utilisateur à définir le type de certains opérateurs constants à l'extérieur de la base : nous avons défini une variable qui contient un contexte global et complète les informations de typage données dans la base.

La simplification

Les simplificateurs modulo les équations d'associativité et de commutativité sont codés dans deux fonctions spécialisées qui à partir d'un vecteur d'arguments `af` construisent les vecteurs d'arguments `a,a:c,af:c`.

Nous avons choisi 13 propriétés qui sont décrites dans l'annexe A.1 page 105 et correspondent chacune à un simplificateur par contexte équationnel avec lequel elles sont compatibles (voir l'annexe A.2 page 109). Ce qui revient à écrire une méthode par type de vecteur d'arguments `a,a:c,af,af:c,af:n, af:n:c`.

La fonction de simplification générale teste les propriétés de l'opérateur et appelle les fonctions de simplification, construites automatiquement (voir page 40) grâce aux caractéristiques des propriétés, dans l'ordre défini page 111.

Lors du calcul d'un terme modulo commutativité, le simplificateur tri les arguments simplifiés du terme. Nous avons choisi de comparer les termes par leurs types en utilisant l'ordre des conversions.
Comme cet ordre n'est pas total, il nous a fallu le compléter. Pour cela nous avons ordonné les chaînes et les $\mathcal{S}$-types d'un même type formel suivant l'ordre des déclarations.

Un terme peut avoir à la fois un $\mathcal{F}$-type et un $\mathcal{S}$-type, on compare d'abord les $\mathcal{F}$-type puis les $\mathcal{S}$-type et enfin la structure (l'opérateur et les arguments). On choisit de considérer qu'un terme spécifié est "plus petit" qu'un terme non spécifié de même type formel.
Il faut définir une fonction de comparaison interne à chaque $\mathcal{S}$-type. Pour que cet ordre soit indépendant de la représentation, il faut comparer tous les termes spécifiés de même $\mathcal{F}$-type par la même fonction.

La réécriture

La caractéristique principale de notre mécanisme de réécriture est qu'il est réalisé modulo associativité et commutativité. C'est à dire que les algorithmes de filtrage et d'unification à la base de la réécriture prennent en compte les équations d'associativité et de commutativité.

L'autre caractéristique est que les variables de pattern peuvent être typées par des types formels de manière à contraindre l'utilisation des règles. Les règles sont typées lors de leur déclaration, chaque règle déclarée est remplacée par l'ensemble de ses règles typées. Pour accélérer la réécriture, le système de règles est compilé, c'est à dire qu'il est transformé en un arbre de décision, puis en une fonction Le-Lisp qui est elle compilée par Complice (le compilateur de Le-Lisp).

L'évaluation

L'évaluateur se compose de trois méthodes d'évaluation équationnelles (`a:c,a,af`), de méthodes de mise en correspondance (`a:c,a,af,n`) et d'une fonction d'application qui effectue les conversions si nécessaire.

De façon générale, lors de l'évaluation d'un terme il faut $\mathcal{S}$-convertir les objets spécifiés d'une représentation spécifique vers une autre. Pour chaque $\mathcal{S}$-conversion déclarée, il faut donc définir une fonction de conversion qui effectue cette transformation.
Pour éviter d'avoir à définir toutes ces fonctions de conversions nous utilisons une méthode générale de conversion par défaut qui est simple mais peu efficace. Comme chaque représentation spécifique est associée à une fonction de construction (lecteur) et une fonction de destruction (imprimeur), nous calculons les conversions comme une composition de ces deux types de

fonctions. C'est à dire que si s_1 et s_2 sont deux représentations spécifiques associées respectivement à $(lect_1, imp_1)$ et $(lect_2, imp_2)$:

$$convert(x : s_1, s_2) = lect_2(imp_1(x)).$$

Cela revient à utiliser la représentation générale comme une représentation intermédiaire commune à toutes les représentations spécifiques.

7.1.3 Le contrôle

Les quatre fonctions de calcul présentées préalablement effectuent des classes de transformations différentes sur les termes. Suivant le terme, il peut être nécessaire d'utiliser une forme de calcul plutôt qu'une autre ou même d'en utiliser plusieurs, ces fonctions sont donc des calculateurs élémentaires qui doivent être dirigés de manière à effectuer le calcul désiré.

Dans le prototype, nous supposons que l'équation d'associativité A est portée par l'opérateur générique, de manière à pouvoir simplifier par A avant typage et donc utiliser le bon algorithme de typage.
Nous avons choisi privilégier la simplification par rapport à la réécriture, mais aussi à l'évaluation. La simplification et l'évaluation utilisant la même stratégie, sont couplés pour permettre un unique parcours de l'arbre. Ces calculateurs élémentaires sont donc appelés successivement par le contrôleur dans l'ordre suivant :

1. simplification récursive par A,
2. typage récursif,
3. simplification (par C et les propriétés) et évaluation récursif,
4. réécriture.

Nous avons défini des paramètres qui permettent d'autoriser ou d'interdire l'utilisation de chacun de ces calculateurs.

7.2 Historique de bases

Théoriquement chaque historique contient l'état initial de la base courante, puis pour chaque état l'ensemble des modifications positives et négatives. Mais pour des raisons de simplicité, nous avons choisi de garder dans l'historique la totalité des états de la base. L'historique courant est stocké dans une variable globale.

7.2.1 La structuration de la base

Chaque base contient un grand nombre d'informations de différentes sortes : propriétés, types, interprétations ...
Pour faciliter la recherche d'une information dans cet ensemble, il est nécessaire de les structurer.

Définition 23 *Un* **domaine** *est un type associé à un ensemble de fonctions de calcul munies de propriétés.*

Définition 24 *Un* **opérateur** *est un symbole associé à des propriétés ainsi qu'à des interprétations.*

Les informations de la base peuvent être regroupées autour de la notion de domaine ou de celle d'opérateur. Ces deux visions sont orthogonales et correspondent à des formes différentes de raisonnement, il n'est donc pas possible de décider en toute généralité si l'une est meilleure que l'autre.

Choix VIII *Nous avons choisi de structurer la base autour de la notion d'opérateur et non de celle de domaine pour plusieurs raisons.*
La première est que dans notre approche du calcul formel, la notion de définition incrémentale est primordiale. Dans une structuration par domaines il faudrait introduire la notion de domaine incomplet, et de domaine ramasse miettes pour permettre une définition incrémentale des domaines, ce qui nuit à l'esthétique de l'approche.
La seconde raison est d'ordre purement pratique : quand le système doit calculer un terme $f(x_1, \cdots, x_n)$*, la seule information immédiate qu'il possède*

est le nom f de l'opérateur de tête du terme et c'est à partir de cela qu'il recherche les autres informations. En regroupant toutes les informations autour des noms d'opérateurs, la recherche dans la base est donc facilitée.

De toute manière, par l'intermédiaire de l'interface de création de la base, on rend sa structure interne transparente pour l'utilisateur. Le choix peut donc se faire sans égards envers l'utilisateur.

Dans la base il existe des informations spécifiques à chaque opérateur : propriétés, $\mathcal{S}$-types fonctionnels, $\mathcal{F}$-types fonctionnels, fonctions de calcul; mais aussi des informations globales à tous les opérateurs : les paramètres globaux du contrôleur, les chaînes de conversions ainsi que les règles de réécriture. Les paramètres de contrôle ont une structure simple puisque que ce sont des drapeaux, par contre pour décrire les types il faut donner plusieurs informations (types formels, représentations spécifiques, conversions ...). La base contient donc une deuxième forme d'objet structuré qui permet de rendre compte des informations concernant les types : une **chaîne de types** et enfin un **systèmes de règles**.

Les historiques ainsi que les bases et toutes les connaissances qu'elle contiennent sont définis sous forme d'objets structurés Le-Lisp c'est à dire de vecteurs typés. Une base est crée par chargement d'un fichier Le-Lisp, dans lequel les objets sont construits explicitement.

7.2.2 Les objets de la base

Une base contient donc un ensemble de chaînes de types, un ensemble d'opérateurs génériques, un ensemble d'opérateurs typés, un ensemble d'interprétations, un système de règles et une liste de valeurs de paramètres décrivant le fonctionnement du moteur.
Dans les sections suivantes, nous allons décrire plus précisément l'organisation des informations à l'intérieur des objets structurés : opérateur, chaîne de types et système de règles.

Chaîne de types

Les informations concernant les types décrites dans le chapitre 4 sont :

1. les types formels,
2. les conversions entre types formels.

La structure pratique d'une chaîne de types correspond exactement à la notion théorique de chaîne de conversions. En effet, une chaîne de types est une liste de types formels ordonnés par les conversions dans laquelle chaque type formel est un nom associé à un ensemble (qui peut être vide) de représentations. L'ensemble des types et des conversions qui les lient correspond donc dans la base à un ensemble de chaînes de types (Figure 3.1 page 90).

Un $\mathcal{S}$-type est théoriquement structuré en :

1. un lecteur,
2. un imprimeur,
3. un comparateur.

Dans Ulysse, il n'existe aucune façon de définir de nouveaux $\mathcal{S}$-type, les $\mathcal{S}$-types ne sont donc décrits que par des noms et pour chacun d'entre eux, nous supposons ces fonctions connues et disponibles dans une bibliothèque accessible (par exemple Le-Lisp ou Sisyphe).

Opérateur

Les connaissances concernant un opérateur sont : des propriétés, des $\mathcal{S}$-types fonctionnels, des $\mathcal{F}$-types fonctionnels et des fonctions de calcul.

Comme chaque $\mathcal{S}$-type correspond à un $\mathcal{F}$-type, on peut représenter un opérateur sous forme d'arbre (voir figure page 90). Chaque opérateur générique correspond à un ensemble d'opérateurs $\mathcal{F}$-typés et chaque opérateur $\mathcal{F}$-typé à un ensemble d'opérateurs $\mathcal{S}$-typés. Nous avons stocké les trois niveaux d'opérateur séparément pour accélérer les accès, nous avons donc donné un nom à chaque opérateur $\mathcal{F}$-typé ainsi qu'à chaque opérateur $\mathcal{S}$-typé.

Comme les calculs peuvent se faire sans typage, on autorise un opérateur générique à être associé à des propriétés. Un opérateur générique est donc

formé d'un nom de propriétés, et d'opérateurs typés.
Un opérateur $\mathcal{F}$-typé est composé d'un nom, d'un $\mathcal{F}$-type fonctionnel, de propriétés et d'une liste d'opérateurs $\mathcal{S}$-typés, alors qu'un opérateur $\mathcal{S}$-typé est composé d'un nom, d'un $\mathcal{S}$-type, et d'une fonction de calcul.

Nous avons caractérisé (page 62) la forme des types fonctionnels suivant les équations et l'arité de l'opérateur auquel elles sont associées, à partir de là nous avons défini les types fonctionnels comme des objets structurés de types `af,n,a`.

Les propriétés sont mémorisées dans la base sous forme de listes d'associations nom de propriété/paramètres. Pour un opérateur générique (respectivement $\mathcal{F}$-typé), les paramètres sont des opérateurs génériques ($\mathcal{F}$-typés).

Nous avons ajouté certains paramètres qui permettent à l'opérateur de diriger le comportement du moteur. Par exemple, nous avons introduit la paramètre `non-eval-fils` qui indique au système que les sous-termes ne doivent pas être évalués.

Lorsqu'on autorise la définition d'un opérateur d'un point de vue algébrique, il paraît naturel d'autoriser aussi sa définition d'un point de vue syntaxique. Par exemple, si un opérateur est associatif, on veut en général le considérer comme infix. La base contient donc non seulement les définitions algébriques des opérateurs, mais aussi leurs définitions syntaxiques qui sont précisées dans l'annexe C.3 page 124.

Figure 3.1: Chaînes de types.

Figure 3.2: Opérateur +.

Exemple 2 *Pour construire un opérateur, il faut créer :*

1. *un opérateur générique :*
   *(***generique** *nom liste-de-props liste-d'opérateurs-$\mathcal{F}$-typés)*
2. *des opérateurs $\mathcal{F}$-typés :*
   *(***Fop** *nom type liste-de-props liste-d'opérateurs-$\mathcal{S}$-typés)*
3. *des opérateurs $\mathcal{S}$-typés :*
   *(***Sop** *nom type nom-fonction-de-calcul)*

Définition de l'opérateur + niveau par niveau :

```
(generique '+ '((is-a) (is-c) (has-n . zero) (has-s . -))
              '(|+fe| |+fp| |+fm|))

(Fop-a '|+fe| 'fentier ())
(Sop-a '|+fe| 'entier '#:sisyphe:add)
(Fop-a '|+fp| 'fpolynome ())
(Sop-a '|+fp| 'polynome '#:sisyphe:rat+)
(Fop-a '|+fm| 'fmatrice ())
(Sop-a '|+fm| '#:bag:matrix:dense '#:sisyphe:mat+)
```

Les systèmes de règles

Les règles sont représentées par un objet de type `rule` qui contient deux termes (partie gauche, partie droite). Un système de règles se compose d'une liste de règles et d'une fonction qui correspond au système de règles compilé.

7.2.3 La base par défaut

Nous avons défini une base par défaut qui contient les informations minimales nécessaires au système. Les types `fix,float,string,#:r:z,#:r:q` de Le-Lisp sont reconnus par Ulysse à la lecture et déclarés dans la base par les chaînes de types suivantes :

```
(chaine 'chaine-scalaire 1
     '((fentier . (fix #:r:z)) (frationnel . (#:r:q)) (freel . (float))))
(chaine 'chaine-chaine 2 '((fchaine . (string)))).
```

La base par défaut contient aussi les déclarations concernant les types des objets manipulés par la réécriture :

```
(chaine 'chaine-rule 3
         '((frule . (rule)) (fsysteme . (rewriting-systeme))))

(generique 'rules () '(rules1))
(Fop-n 'rules1 'frule 'fsysteme ())
(Sop-n 'rules1 'rule 'rewriting-systeme '#:i:compile-system)

(generique 'rule () '(rule1))
(Fop-af 'rule1 '(any any) 'frule ())
(Sop-af 'rule1 '(any any) 'rule '#:ul:alg:make-rule)
```

De plus, la base par défaut contient la définition des quatre opérateurs suivants :

1. `charge(nom-base)` qui permet de charger une base de nom `nom-base`,
2. `base(nom-base)` qui permet de positionner la base courante à la base de nom `nom-base`,
3. `contexte` qui donne le contexte de type global,
4. `contextes` qui donne les contextes dans lesquels la dernière expression calculée est valide,
5. `declare(param)` qui défini `param` comme un paramètre de contrôle du moteur (typage, non-typage, eval, non-eval, reecriture, non-reecriture),
6. `declare(var,type)` qui défini `var` comme étant de type `type` dans le contexte global.

```
(generique 'charge '((non-eval-fils)) '(charge1))
(Fop-af 'charge1 '(fchaine) 'fentier ())
(Sop-af 'charge1 '(string) 'fix '#:ul:gestion-base:charge-base)

(generique 'base '((non-eval-fils)) '(base1))
(Fop-af 'base1 '(fchaine) 'fchaine ())
```

```
(Sop-af 'base1 '(string) 'string '#:ul:gestion-base:base)

(generique 'declare () '(declare1 declare2))
(Fop-af 'declare1 '(fchaine fchaine) 'fchaine ())
(Sop-af 'declare1 '(string string) 'string
           '#:ul:gestion-base:declare)

(Fop-af 'declare2 '(fchaine) 'fchaine ())
(Sop-af 'declare2 '(string) 'string '#:ul:base:controle)
```

Enfin, elle contient des informations syntaxiques qui permettent la lecture et l'impression correcte des expressions :

```
(sortie '|paren| '((externe "(" . ")") (rbp . -1) (lbp . -1)))
(sortie '|text|
        '((externe . "") (fixity . nary) (rbp . 0) (lbp . 0)))
(sortie '|text-space|
        '((externe . " ") (externe-t . "~")
        (fixity . nary) (rbp . 0) (lbp . 0)))
(sortie '|comma| '((externe . ",")))
(entree '|->| '((fixity (infix . rule)) (lbp . 90) (rbp . 90)))
(sortie '|->| '((fixity . infix) (lbp . 90) (rbp . 90)))
(sortie 'rule '((fixity . infix) (lbp . 90)))
(entree '|#| '((fixity . ((prefix . Var))) (rbp . 200) (lbp . 200)))
(sortie '|#| '((fixity . prefix) (rbp . 200) (lbp . 200)))
(sortie 'Var '((fixity . prefix) (rbp . 200) (lbp . 200)))
```

Pour définir une base pour Ulysse, il suffit de compléter cette base.

7.3 Pour aller plus loin

Dans le prototype que nous venons de présenter, les bases sont créées par simple chargement d'un fichier mais sans aucune vérification de cohérence. Il est possible de vérifier cette cohérence en créant des liens entre les objets, en particulier des liens d'existence. Par exemple, un tel lien pourrait signifier qu'un type fonctionnel ne peut être déclaré pour un opérateur que si tous les types atomiques qu'il contient sont définis ...

De plus, les fonctions de calcul sont supposées chargées et les types spécifiques (structures Le-Lisp) sont supposés prédéfinis. Pour aller plus loin, nous avons défini une méthode simple pour charger les fonctions ainsi que les définitions des types spécifiques en même temps que la base. Pour définir une base de nom `toto`, nous avons décidé qu'il fallait définir trois fichiers :

1. `toto.base` : qui contient les définitions d'objets de la base,
2. `toto.objs` : qui contient les définitions des types spécifiés (structure Le-Lisp, comparateur, lecteur et imprimeur),
3. `toto.bib` : qui contient les définitions de fonctions Le-Lisp liées aux opérateurs de la base.

Le fait que l'historique soit un objet structuré dans lequel les bases sont rangées après modification permet une mise au point facile des informations à définir (fonctions, règles, propriétés). Cela fait d'Ulysse un outil intéressant à utiliser lors de la spécification d'un problème. En effet face à un problème, l'utilisateur ne sait pas toujours quels opérateurs définir, quelles propriétés (ou règles) leur associer, enfin quelles représentations spécifiques et fonctions de calcul correspondent aux données du problème. Ulysse "joue à fond" la carte de la personnalisation et de l'incrémentalité des déclarations, il correspond à la phase de clarification d'un problème.

Mais on peut penser, une fois les trois fichiers `toto.base, toto.objs` et `toto.bib` satisfaisants, les compiler pour accélérer les calculs. Les deux fichiers `toto.objs` et `toto.bib` sont alors compilés par Complice (compilateur de Le-Lisp) en un fichier `toto.lo`. En ce qui concerne le fichier `toto.base` la compilation pourrait (nous n'avons pas mis en pratique !) correspondre à la construction d'une fonction par opérateur comme pour les système de règles, qui pourraient ensuite être compilées.

Chapter 8

Exemples commentés de session

Nous allons maintenant donner un aperçu des calculs effectués par le système en précisant à chaque fois la base définie. Les déclarations usuelles sont disponibles sous forme d'une base pouvant être chargée et éditée très simplement.
Nous présentons tout d'abord les calculs, et ensuite la base qui permet au système de les faire.

Session de calculs :

```
(1) declare("m1","fmatrice");

1                                   fmatrice

(2) declare("m2","fmatrice");

2                                   fmatrice

(3) declare("e","fentier");

3                                   fentier

(4) contexte;
```

```
4          [[e , fentier] , [m2 , fmatrice] , [m1 , fmatrice]]

(5) m1 * 2 * m2 * 3.4 * e;

5                    (2 *  e *  3.4) *   (m1 *  m2)
                        r    r       rm      m

(6) 2 * 3 * m1;

6                              (2 *  3) *   m1
                                  r     rm

(7) 3 * fib(n+m);

7                         3 * fib(m [+  , + ] n))
                                    r    m

(8) contextes;

8                                      []

(9) f(2)+f(1);

9                             f(1) [+  , + ] f(2)
                                    r    m

(10) contextes;

10  [[freel , [[2 , fentier] , [f(2) , freel] ,
              [f(1) , freel] , [1 , fentier]]] ,
    [fmatrice , [[2 , fentier] , [f(2) , fmatrice] ,
                 [f(1) , fmatrice] , [1 , fentier]]]]

(11) dg(x)+fib(x);

11                             dg (x) +  fib (x)
                                 e     r    e

(12) contextes;
```

```
12                        [[freel , [[x , fentier]]]]

(13) m1 + - m1;

13                                  Zerom

(14) m1 + - m1 + e;

14                             e + (-  m1) + m1 \* Erreur de typage *\
                                     m

(15) contextes;

15                                        []

(16)  m1 * m2 * Zero;

16                                   Zerom

(17) declare("v1","fvecteur");

17                                  fvecteur

(18) declare("v2","fvecteur");

18                                  fvecteur

(19) tr (tr (m1));

19                                     m1

(20) v1 x v2 + v2 x v1;

20                                  Zerov

(21) v3 x v4;

21                               v3 x  v4
                                     v
```

```
(22) m1 * inv(m1);

22                                    Id

(23) contextes;

23            [[fvecteur , [[v4 , fvecteur] , [v3 , fvecteur]]]]

(24) contexte;

24  [[R2 , fmatrice] , [r23 , freel] , [Z0 , fmatrice] , [r12 , freel] ,

     [X0 , fmatrice] , [R1 , fmatrice]]

(25) - tilde(R1*r12*X0)*Z0 + tilde(R1*R2*r23*X0)*Z0;

25            (r23 *   tilde1(R1 *  R2 *  X0)) *  Z0
                    rm            m     m       m
       +  (-  (r12 *   tilde1(R1 *  X0)) *  Z0)
        m   m       rm            m       m

(26) rules(tilde(Zerov)->Zerom,
           tilde(#m)*#m->Zerom,
           tilde(tilde(#m1)*#m2)->tilde(#m1)*tilde(#m2) + -
                                    (tilde(#m2)+ tilde(#m1)),
           tilde(rot(#ax,#angle)*#m)->
               rot(#ax,#angle)*tilde(#m*tr(rot(#ax,#angle))),
           tilde(X0)*Z0->-Y0,
           tilde(X0)*Y0->Z0,
           tilde(Y0)*Z0->X0,
           tilde(Y0)*X0->-Z0,
           tilde(Z0)*X0->Y0,
           tilde(Z0)*Y0->-X0,
           tr(Id)->Id,
           tr(rot(#ax,#angle))*rot(#ax,#angle)->Id,
           rot(#ax,#angle)*#ax->#ax,
           rot(#ax,#angle)*tr(rot(#ax,#angle))->Id,
           rot(Z0,#angle)*X0->sin(#angle)*Y0 + cos(#angle)*X0,
           rot(Z0,#angle)*Y0->cos(#angle)*Y0 + sin(#angle)*X0,
```

```
           R1->rot(Z0,Theta1),
           R2->rot(Z0,Theta2),
           cos(Theta1)-> C1,
           cos(Theta2)-> C2,
           sin(Theta1)-> S1,
           sin(Theta2)-> S2);

26                 rewrite-systeme( ... )

(27) - tilde(R1*r12*X0)*Z0 + tilde(R1*R2*r23*X0)*Z0;

27  rot(Z0 , Theta1) * rot(Z0 , Theta2) * Zerov * Z0
    + -(rot(Z0 , Theta1) * Zerov * Z0)
```

Voici les différences entre la base par défaut précédemment décrite et la base qui permet d'effectuer les calculs énoncés dans cette session. Nous avons ajouté à la base courante la chaîne de types définissant les scalaires :

```
(chaine 'chaine-matrice 3 '((fvecteur) (fmatrice))),
```

ainsi que les définition d'opérateurs suivantes :

```
(generique '+ '((is-a) (is-c) (has-n . Zero) (has-s . -))
              '(|+r| |+m| |+v|))
(Fop-a '|+r| 'freel '((has-n . Zeror) (has-s . |-r|)))
(Fop-a '|+v| 'fvecteur '((has-n . Zerov) (has-s . |-v|)))
(Fop-a '|+m| 'fmatrice '((has-n . Zerom) (has-s . |-m|)))

(generique '|-| '((is-inv)) '(|-r| |-m|))
(Fop-af '|-r| '(freel) 'freel '((is-hom . (+r +r))))
(Fop-af '|-v| '(fvecteur) 'fvecteur '((is-hom . (+v +v))))
(Fop-af '|-m| '(fmatrice) 'fmatrice '((is-hom . (+m +m))))

(generique 'Zero () '(Zeror Zerov Zerom))
(Fop-af 'Zeror () 'freel ())
(Fop-af 'Zerov () 'fvecteur ())
(Fop-af 'Zerom () 'fmatrice ())
```

```
(generique '* '((is-a)) '(|*r| |*m| |*rm| |*mr|))
(Fop-a '|*r| 'freel
    '((is-c) (has-n . Unr) (has-abs . Zeror) (has-s . invr)))
(Fop-a '|*m| 'fmatrice
    '((is-hom . ((*rm *rm)))
          (has-ng . Id) (has-nd . Id)
          (has-absg . Zerom) (has-absd . Zerom)
          (has-sg . invm) (has-sd . invm)))
(Fop-af '|*rm| '(freel fmatrice 'fmatrice)
    '((is-pe . *r) (is-externe)
                   (has-ng . Unr) (has-nd . Id)
                   (has-absg . Zeror) (has-absd . Zerom)))
(Fop-af '|*mr| '(fmatrice freel) 'fmatrice
    '((is-pe . *m) (is-perm . *rm) (is-externe)))

(generique 'Un () '(Unr Id Unv))
(Fop-af 'Id () 'fmatrice ())
(Fop-af 'Unr () 'freel ())
(Fop-af 'Unv () 'fvecteur ())

(generique 'inv '((is-inv)) '(invm invr))
(Fop-af 'invm '(fmatrice) 'fmatrice ())
(Fop-af 'invr '(freel) 'freel ())

(generique 'pui () '(puir puim))
(Fop-af 'puir '(freel fentier) 'freel ())
(Fop-af 'puim '(fmatrice fentier) 'fmatrice ())

(generique 'tr () '(tr1))
(Fop-af 'tr1 '(fmatrice) 'fmatrice
    '((is-inv) (is-hom ((*m *m))) (is-h ((+m +m) (-m -m)))))

(generique 'tilde () '(tilde1))
(Fop-af 'tilde1 '(fmatrice) 'fmatrice
    '((is-hom ((+m +m))) (is-h ((*rm *rm)))))

(generique 'cross '((is-a)) '(cross1))
(Fop-af 'cross1 '(fvecteur fvecteur) 'fvecteur
    '(is-anti is-hom) '(-v -v))
```

```
(generique 'fib () '(fibe))
(Fop-af 'fibo '(fentier) 'fentier ())

(generique 'dg () '(dge))
(Fop-af 'dge '(fentier) 'fentier ())
```

Chapter 9

Vers un nouveau modèle de système de calcul formel

Nous considérons qu'Ulysse est un **interprète** constitué d'un lecteur, d'un moteur et d'un imprimeur, tous trois paramétré par des informations stockées dans une base. Ce système est entièrement personnalisable puisqu'il suffit d'ajouter ou de retrancher des informations à la base pour redéfinir son comportement.

Ce système est parfaitement autonome en ce qui concerne la simplification, la réécriture et le typage, cependant il dépend d'une **bibliothèque** en ce qui concerne l'évaluation. La définition de fonctions et de représentations spécifiques ne rentre pas dans le cadre que nous avons défini, il n'y a donc aucune possibilité pour l'instant de définir des représentations ou des fonctions spécifiques. La bibliothèque doit donc contenir les fonctions de gestion des $\mathcal{S}$-type ainsi que les fonctions de calcul spécifiques, ce qui correspond aux fichiers `.objs` et `.bib` que nous avons définis (page 93). Nous utilisons actuellement le système Sisyphe comme bibliothèque pour Ulysse, pour cela nous avons créée une base `sisyphe` qui une fois chargée transforme Ulysse en un système équivalent à Sisyphe. De par la construction de Sisyphe, nous n'avons pas pu isoler des morceaux de Sisyphe pouvant être chargés séparément, la chargement de la base `sisyphe` entraîne donc le chargement de la plupart des modules de Sisyphe.

A notre avis, un système de calcul formel fondé sur un interprète tel que nous

l'avons défini est nécessairement dépendant d'une bibliothèque qui contient les définitions de fonction de calcul et de représentations spécifiques.
Dans les systèmes classiques Macsyma, Maple, Reduce, Mathematica, un compromis a été réalisé pour que l'interprète permette aussi de définir des fonctions de calcul, par contre il ne permet pas la définition de nouveaux types spécifiés.
En Axiom par contre, deux modes de calcul existent effectivement : un mode interprété et un mode compilé. Mais tandis que le mode compilé est performant puisqu'il permet une définition structurée des types spécifiques et des fonctions, le mode interprété est réduit à sa plus simple expression : l'évaluation.

Pour nous, un système de calcul formel évolué doit concrétiser deux modes de calcul ayant des caractéristiques différentes : le mode interprété dont les maîtres mots sont convivialité, interactivité, souplesse ... car il doit s'adapter à la personnalité de l'utilisateur pour lui permettre de mieux exprimer sa façon de mener un calcul et le mode compilé qui gère un savoir encyclopédique et est régi par l'efficacité, l'extensibilité, la réutilisabilité, la modularité ...
Nous avons défini au long de cette thèse le mode interprété. Axiom nous semble être un exemple représentant le mode compilé, le prototype du langage Xfun développé par S.Dalmas et décrit dans sa thèse [Dal91] en est aussi une illustration.

Pour qu'un système différenciant ces deux modes soit convivial, il lui faut un troisième composant qui unifie les deux premier : l'interface. En effet, il doit être possible d'accéder à ces deux mondes en offrant à l'utilisateur les fonctionnalités spécifiques à chacun d'eux. Les interfaces Mathscribe pour Reduce, SUI pour Macsyma ainsi que le prototype Cas/PI développé par N.Kajler pour Maple, Sisyphe et Ulysse sont des interfaces de haut niveau.

Notre modèle de système de calcul formel évolué est donc formé d'un interprète tel que nous l'avons spécifié dans cette thèse, d'un langage associé à son compilateur et d'une interface de haut niveau.

Appendix A

Un choix de propriétés élémentaires

A.1 Choix des propriétés

Nous avons choisi l'associativité (A) et la commutativité (C) comme équations. Comme la commutativité est une équation dérivée de la g-symétrie (G), nous avons introduit cette équation supplémentaire : $f(b,a) = g(f(a,b)) \Leftrightarrow g\text{-}symmetric(f,g)$.
La forme simplifiée modulo g-symétrie du terme $op(a_1,a_2)$ est $op(a_1,a_2)$ si $a_1 < a_2$ et $g(op(a_2,a_1))$ sinon.
La commutativité est la g-symétrie où $g = Identité$, et l'antisymétrie la g-symétrie avec $g = -$ (si $\wedge$ est antisymétrique $U \wedge V \rightarrow -(V \wedge U)$).

Parmi toutes les propriétés possibles, nous avons sélectionné celles qui sont les plus couramment utilisées en algèbre élémentaire, et correspondent aussi aux axiomes équationnels habituellement utilisés en réécriture [JK91].

Modèles de déclarations	Règle
$P_1 : ext\text{-}op(f,g)$	$R_1 : f(s_1, f(s_2,t_2)) \rightarrow f(g(s_1,s_2),t_2)$
$P_2 : n\text{-}homogeneous(f,g,h)$	$R_2 : f(x_1...g(a,x_i)...x_n) \rightarrow$ $h(a, f(x_1...x_i...x_n))$
$P_3 : n \times m\text{-}morphism(f,g,h)$	$R_3 : f(x_1...g(x_i^1...x_i^m)...x_n) \rightarrow$ $h(f(x_1...x_i^1...x_n)...f(x_1...x_i^m...x_n))$

Les paramètres n et m des noms de propriétés P_2, P_3 correspondent aux arités des opérateurs $n = arite(f)$, $m = arite(g)$, et peuvent être égaux à 1 (en associatif, ils ont pour valeur 2). La variable i utilisée dans les règles peut varier de 1 à n.

Déclaration	*Règle*
$P_4 : projector(f)$	$R_4 : f(f(x)) \rightarrow f(x)$
$P_5 : involutive(f)$	$R_5 : f(f(x)) \rightarrow x$
$P_6 : nilpotent(f, n, h)$	$R_6 : \underbrace{f(f(f...f}_{n}(x))) \rightarrow h$
$P_7 : idempotent(f)$	$R_7 : f(x, x) \rightarrow x$
$P_8 : has\text{-}zero\text{-}l(f, zero)$	$R_8 : f(zero, x) \rightarrow x$
$P_9 : has\text{-}zero\text{-}r(f, zero)$	$R_9 : f(x, zero) \rightarrow x$
$P_{10} : has\text{-}ab\text{-}l(f, ab)$	$R_{10} : f(ab, x) \rightarrow ab$
$P_{11} : has\text{-}ab\text{-}r(f, ab)$	$R_{11} : f(x, ab) \rightarrow ab$
$P_{12} : has\text{-}symmetric\text{-}l(f, h)$	$R_{12} : f(h(x), x) \rightarrow zero$
$P_{13} : has\text{-}symmetric\text{-}r(f, h)$	$R_{13} : f(x, h(x)) \rightarrow zero$

Exemple 3 *Voici quelques exemples de simplifications par ces propriétés.*
Simplification de $m_1 \cdot mul(s_2, m_2) \cdot mul(s_4, m_4 + m_5)$*, où* $\cdot$ *est le produit matriciel, mul le produit scalaire/matrice et* $+$ *la somme. Les hypothèses habituelles concernant ces opérateurs sont :*

$$\begin{array}{rcl} H_1 & : & 2\text{-}homogeneous(\cdot, mul, mul), \\ H_2 & : & 2 \times 2\text{-}morphisme(\cdot, +, +), \\ H_3 & : & 2 \times 2\text{-}morphisme(*, +, +), \\ H_4 & : & ext\text{-}op(mul, *), \\ H_5 & : & 2 \times 2\text{-}morphisme(mul, +, +). \end{array}$$

Comme $\cdot$ *est associatif, la forme simplifiée est calculée dans le contexte associatif. Le tableau des différentes transformations effectuées sur le terme* $m_1 \cdot mul(s_2, m_2) \cdot mul(s_4, m_4 + m_5)$*, sous*

les hypothèses H_1, H_2, H_3, H_4 est :

Hypothèse utilisée	*Terme calculé*
	$m_1 \cdot mul(s_2, m_2) \cdot mul(s_4, m_4 + m_5)$
H_1	$mul(s_2, m_1 \cdot m_2 \cdot mul(s_4, m_4 + m_5))$
H_1	$mul(s_2, mul(s_4, m_1 \cdot m_2 \cdot (m_4 + m_5)))$
H_4	$mul(s_2 * s_4, m_1 \cdot m_2 \cdot (m_4 + m_5))$
H_2	$mul(s_2 * s_4, (m_1 \cdot m_2 \cdot m_4) + (m_1 \cdot m_2 \cdot m_5))$
H_5	$mul(s_2 * s_4, m_1 \cdot m_2 \cdot m_4) + mul(s_2 * s_4, m_1 \cdot m_2 \cdot m_5)$

Simplification de $inv(a * b * inv(a) * inv(c))$, où $*$ est le produit courant et inv l'inverse courant.
Les déclarations courantes concernant ces opérateurs sont :

$$
\begin{array}{lcl}
H_1 & : & \textit{has-zero-l}(*, un), \\
H_2 & : & \textit{has-zero-r}(*, un), \\
H_3 & : & \textit{has-ab-l}(*, zero), \\
H_4 & : & \textit{has-ab-r}(*, zero), \\
H_5 & : & \textit{has-symmetric-l}(*, inv), \\
H_6 & : & \textit{has-symmetric-r}(*, inv), \\
H_7 & : & \textit{involutive}(inv), \\
H_8 & : & 1 \times 2\text{-}\textit{morphism}(inv, *, *).
\end{array}
$$

Comme $*$ est AC, la forme simplifiée de tout terme d'opérateur $*$ est simplifié dans le contexte AC. Par contre tout terme d'opérateur inv est simplifié en contexte standard. La simplification du terme $inv(a * b * inv(a) * inv(c))$ commence par la simplification de $a * b * inv(a) * inv(c)$, puis simplification du terme total. Voici le tableau des transformations successives :

Hypothèse utilisées	*Terme calculé*
	$a * b * inv(a) * inv(c)$
H_6	$b * un * inv(c)$
H_1	$b * inv(c)$
	$inv(b * inv(c))$
H_8	$inv(b) * inv(inv(c))$
H_7	$inv(b) * c$

Nous avons redéfini toutes les propriétés algébriques utilisées dans les systèmes classiques (excepté la propriété `non-com` de Reduce qui est vraiment non standard) à partir des quinze propriétés décrites dans les sections précédentes.
Voici le tableau des correspondances entre les propriétés des systèmes classiques et celles que nous avons choisies pour Ulysse :

Système	Propriété	Ensemble de propriétés élémentaires équivalentes
Maple	*associative*	A
	commutative	G
	identity	P_8, P_9
	inverse	P_{12}, P_{13}
	zero	P_{10}, P_{11}
	group	$P_{18}, P_{19}, P_{12}, P_{13}$
	linear	P_3, P_2
Mathematica	*Orderless*	G
	Flat	A, P_4
	OneIdentity	P_5
	Listable	P_3

Système	Propriété	Ensemble de propriétés élémentaires équivalentes
Macsyma	*additive*	P_3
	outative	P_2
	linear	P_3, P_2
	multiplicative	P_3
	lassociative	A
	rassociative	A
	commutative	G
	symmetric	G
	antisymmetric	G
Reduce	*linear*	P_3, P_2
	noncom	????
	symmetric	G
	antisymmetric	G

Nos propriétés sont donc suffisamment paramétrées pour que leur puissance d'expression soit plus importante que celle des propriétés des autres systèmes.

A.2 Compatibilité entre propriétés

Un opérateur est en général muni non pas d'une propriété mais d'une liste de propriétés. Or on ne peut simplifier un terme par les propriétés de son opérateur que si elles sont compatibles entre elles, mais aussi compatibles avec le terme lui même, il faut donc définir ces compatibilités.

Les propriétés d'un opérateur f doivent être compatibles par arité, c'est à dire qu'elles doivent supposer la même arité à f.

Lemme A.25 *Les propriétés peuvent être classées suivant l'arité de l'opérateur qui les porte:*

1. *unaire :* $P_2 ... P_6$,
2. *binaire :* $A, G, P_1 ... P_3, P_7 ... P_{13}$,
3. *sinon* $(n > 2)$ *:* $G, P_2, P_3, P_7 ... P_{13}$.

Dans chacune des listes précédemment définies, certaines propriétés s'excluent mutuellement pour diverses raisons :

1. conflits entre leurs règles équivalentes,
   par exemple, sous les hypothèses $projector(f)$ et $involutive(f)$, simplifier l'expression $f(f(x))$ est ambigüe,
2. incompatibilité algébrique,
   par exemple, les hypothèses $ext\text{-}op(f, g)$ et $commutatif(f)$ sont incompatibles car $f(s_1, f(s_2, s_3))$ et simplifié en $f(g(s_1, s_2), s_3)$ et $f(s_1, f(s_3, s_2))$ en $f(g(s_1, s_3), s_2)$ mais alors que $f(s_1, f(s_2, s_3))$ et $f(s_1, f(s_3, s_2))$ sont égaux modulo commutativité, $f(g(s_1, s_2), s_3)$ et $f(g(s_1, s_3), s_2)$ sont en général différents.

L'analyse des parties gauches des modèles de règles permet de définir ces conflits. Ainsi, la simplification modulo les propriétés P_4, P_5, P_6 amène une ambigüité. Ces trois propriétés sont donc considérées comme mutuellement exclusives.
De même, deux autres propriétés P_2 et P_3 rendent la simplification ambigüe.

En effet le terme $f(x_1...g(x_i^1, x_i^2)...x_n)$, est simplifié en deux termes différents suivant deux hypothèses :

- si n-*homogeneous*(f, g, h), en $h(x_i^1, f(x_1...x_i^2...x_n))$
- si $n \times m$-*morphisme*(f, g, h), en $h(f(x_1...x_i^1...x_n), f(x_1...x_i^2...x_n))$.

Ces deux propriétés sont donc aussi déclarées incompatibles.
Et enfin la propriété P_3 est incompatible avec P_4, P_5 et P_6 si ses deux paramètres sont les mêmes.

Lemme A.26 *Le tableau des incompatibilités entre $A, G, P_1...P_{13}$ est :*

	P_1	P_3	P_4	P_5	P_6
G	X				
P_2		X			
P_4		X			
P_5		X	X		
P_6		X	X	X	

Dans ce tableau, X désigne les couples de propriétés présentant des incompatibilités.

Proposition A.27 *Parmi les propriétés $A, G, P_1, ..P_{13}$ les plus longues listes $L_1...L_9$ valides sont les listes suivantes :*

	A	G	P_1	P_2	P_3	P_4	P_5	P_6	P_7	...	P_{13}
L_1				$\bowtie$		$\bowtie$					
L_2				$\bowtie$			$\bowtie$				
L_3				$\bowtie$				$\bowtie$			
L_4	$\bowtie$	$\bowtie$		$\bowtie$					$\bowtie$	...	$\bowtie$
L_5	$\bowtie$		$\bowtie$	$\bowtie$					$\bowtie$	...	$\bowtie$
L_6	$\bowtie$	$\bowtie$			$\bowtie$				$\bowtie$	...	$\bowtie$
L_7	$\bowtie$		$\bowtie$		$\bowtie$				$\bowtie$	...	$\bowtie$
L_8		$\bowtie$		$\bowtie$					$\bowtie$	...	$\bowtie$
L_9		$\bowtie$			$\bowtie$				$\bowtie$	...	$\bowtie$

Dans ce tableau, les $\bowtie$ relient les couples (P_i, L_j) tels que $P_i \in L_j$.

Etant donné un opérateur f muni d'une liste de propriétés $p_1..p_m$, pour simplifier un terme $f(x_1, \cdots, x_n)$ il faut choisir la plus grande sous-liste l de $p_1..p_m$ de propriétés compatibles entre elles et compatibles avec l'arité effective n de f.

Exemple 4 *Si involutif(f) et idempotent(f), pour simplifier $f(x, x)$ seule la propriété qui suppose f binaire est utilisable, donc le terme est simplifié en x.*

A.3 Ordre sur les propriétés

Bien évidemment, les propriétés A, G sont utilisées en premier donc $\forall i\ P_i > G > A$.
En effet, si un terme est simplifié modulo commutativité, puis associativité, le résultat n'est en général plus simplifié modulo commutativité. Par exemple $b + a + b + (c + d) + d$ est transformé en $a + b + b + d + (c + d)$ puis en $a + b + b + d + c + d$ qui n'est plus simplifié modulo commutativité. Par contre, si l'associativité est utilisée avant commutativité, il est certain que le terme est encore simplifié modulo associativité puisque l'utilisation de la commutativité ne peut faire apparaître de sous-terme direct de même opérateur de tête que le terme.

Une fois le terme simplifié par A, G, il faut le simplifier par les autres modulo les équations A, C. L'hypothèse sur l'ordre que nous avons faite nous donne une seule contrainte : $P_{12}P_{13} < P_8P_9$. En effet, les propriétés $P_{12}P_{13}$ créent des zéro qui sont supprimés par P_8P_9.

Comme il n'y a pas convergence des propriétés, le résultat de la simplification dépend de l'ordre d'utilisation des règles.

Exemple 5 *Soient les hypothèses :*

H_1 : *has-symmetric-l*(f, f^{-1})
H_2 : 2-*homogeneous*(f, g, h)
H_3 : *idempotent*(f)

Le terme $f(f^{-1}(g(a, x)), g(a, x))$ peut être simplifié de deux façons

suivant l'ordre d'application des règles :

hypothèse	*terme calculé*	*hypothèse*	*terme calculé*
	$f(f^{-1}(g(a,x)), g(a,x))$		$f(f^{-1}(g(a,x)), g(a,x))$
H_1	$\mathbf{zero}$	H_2	$\mathbf{h(a, f(f^{-1}(g(a,x)), x))}$
	$f(g(a,x), g(a,x))$		$f(g(a,x), g(a,x))$
H_3	$\mathbf{g(a,x)}$	H_2	$h(a, f(x, g(a,x)))$
		H_2	$h(a, h(a, f(x,x)))$
		H_3	$\mathbf{h(a, h(a,x))}$

Mais on peut remarquer que les règles $P_4..P_{13}$ réduisent la taille du terme qu'elles simplifient, c'est à dire qu'elles sont orientables automatiquement, alors que les règles $P_3..P_6$ n'ont pas cette caractéristique. En effet, les règles P_6, P_7 transforment un Stgd en l'un de ses sous-termes directs et les règles $P_6..P_{13}$ le transforment en une constante.
Nous allons donc privilégier l'utilisation des propriétés $P_5..P_{13}$ sur $P_1..P_4$, en posant donc $P_5..P_{13} < P_1..P_4$.

Remarque 6 *Plusieurs instances de la même propriété peuvent être associées à un même opérateur, à partir du moment où leurs paramètres sont différents. Par exemple, l'opérateur de transposition est à la fois un endomorphisme par rapport à* $+$ *(*1×2*-morphisme*$(t, +, +)$*) et à* $-$ *(*1×1*-morphisme*$(t, -, -)$*). Comme les paramètres sont différents, les deux parties gauches de la règle instanciées sont différentes, il n'y a donc pas conflit lors de l'application. Nous n'avons pas défini d'ordre de priorité entre ces instances.*

La simplification d'un terme dont les sous-termes directs sont simplifiés par P l'ensemble des propriétés compatibles de son opérateur de tête comporte quatre phases :

1. simplification par $A \in P$,
2. simplification par $G \in P$,
3. simplification par les $P \cap \{P_5..P_{13}\}$,
4. simplification par les $P \cap \{P_1..P_4\}$.

Appendix B

A-typage : vérification et inférence de types

Soit A un ensemble de types atomiques ($\mathcal{S}$ ou $\mathcal{F}$) partitionné en chaînes de types totalement ordonnées par $<_c$, nous allons définir le A-typage d'un terme réduit à un unique sous-terme généralisé direct.

Le $\mathcal{F}$-typage est utilisé pour résoudre la surcharge des opérateurs comme prétraitement de la simplification et de la réécriture. Il correspond donc à de la vérification et à de l'inférence de type.
Le $\mathcal{S}$-typage est lui utilisé pour choisir une interprétation d'un opérateur lors de l'évaluation, dans ce cas tous les opérateurs sont typés, le $\mathcal{S}$-typage ne réalise donc que de la vérification de type.

Nous allons décrire le typage sous sa forme générale le A-typage pour ne pas faire une description redondante.

B.1 Calcul du type d'un terme : A-typage

Le typage est un mécanisme récursif, c'est à dire que pour typer un terme il faut tout d'abord typer ses sous-termes directs. Nous décrivons donc le typage à un niveau, en supposant que chaque sous-terme direct du terme à typer est lui même typé.

De même que le A-type d'un terme peut être décrit par une formule logique (voir page 63), le A-typage d'un terme réduit à son unique sous-terme généralisé direct sachant que ses sous-termes directs sont typés peut être transformé en un calcul logique.

Problème B.28 *Etant donné $\alpha_1..\alpha_n \to \alpha_{n+1}$ un type fonctionnel de f, $\{x_i\}$ une famille de n termes vérifiant les hypothèses $\mathcal{H}_\infty$ suivantes :*

- *si $x_i \in \mathcal{N}$, x_i n'est pas typé,*
- *sinon x_i est typé et a pour type*

$$\bigwedge_j \left[\bigvee_k E_{ijk} \Leftrightarrow x_i : \alpha_{ij} \right].$$

quels sont les environnements dans lesquels le terme $f(x_1, \cdots, x_n)$ peut être typé grâce au type fonctionnel $\alpha_1..\alpha_n \to \alpha_{n+1}$?

Ce calcul peut aisément se découper en trois étapes :

1. inférence des types des $x_i \in \mathcal{N}$,
2. typage de la famille $\{x_i\}$,
3. typage de $f(x_1, \cdots, x_n)$.

Pour que le type fonctionnel permette de typer $f(x_1, \cdots, x_n)$ où $x_i : \alpha'_i$ il faut que la contrainte $\mathcal{C}$ soit vérifiée :

$$\mathcal{C} = \alpha'_1..\alpha'_n \leq_c \alpha_1..\alpha_n.$$

On peut donc inférer les types des $x_i \in \mathcal{N}$ comme étant α_i, les types des x_i sont donc :

$$\mathcal{H}_2 = \bigwedge_{x_i \in \mathcal{N}} [x_i : \alpha_i \Leftrightarrow x_i : \alpha_i].$$

On peut remarquer que α_i est le plus grand type possible pour $x_i \in \mathcal{N}$.

Pour typer la famille $\{x_i\}$, il faut chercher les environnements dans lesquels tous les termes x_i sont typés. L'ensemble des hypothèses $\mathcal{H}_1$ peut être décrit par une conjonction de la forme :

$$\mathcal{H}_1 = \bigwedge_{x_i \notin \mathcal{N}} \left[\bigwedge_{j=1}^{a_i} \left[\bigvee_k E_{ijk} \Leftrightarrow x_i : \alpha_{ij} \right] \right].$$

L'ensemble des hypothèses $\mathcal{H} = \mathcal{H}_1 \wedge \mathcal{H}_2$ sur la famille $\{x_i\}$ peut donc s'écrire :

$$\bigwedge_{\substack{b_i \in [1..a_i] \\ 1 \leq i \leq n}} \left[\bigwedge_i \left[\bigvee_k E_{ib_ik} \right] \Leftrightarrow \bigwedge_i x_i : \alpha_{ib_i} \right]$$

Or

$$\bigwedge_{i=1}^{n} \bigvee_{k=1}^{m} E_{ijk} \Leftrightarrow \bigvee_{c_i \in [1..m_i]} \bigwedge_{i=1}^{n} E_{ijc_i},$$

donc il nous faut calculer des conjonctions d'environnements pour calculer

$$\bigwedge_i \left[\bigvee_k E_{ib_ik} \right].$$

Une conjonction d'environnements peut se réduire à $faux$ si ces environnements sont incompatibles, c'est à dire si un symbole non typé est associé à deux types incomparables dans deux d'entre eux.
Une conjonction de la forme $(z_i : \alpha_i) \wedge (z_j : \alpha_j)$ se calcule en :

1. $(z_i : \alpha_i) \wedge (z_j : \alpha_j)$ si $z_i \neq z_j$,
2. si $z_i = z_j$
   (a) $z_i : inf_c(\alpha_i, \alpha_j)$ si α_i et α_j appartiennent à la même chaîne de conversions,
   (b) $faux$ sinon,

donc

$$\bigwedge_{i=1}^{n} \bigvee_{k=1}^{m} E_{ijk} \Leftrightarrow \bigvee_{\substack{1 \leq_c i \leq_c n \\ E_{ijc_i} \neq faux}} E_{ijc_i}$$

et finalement,

$$\mathcal{H} \Rightarrow \mathcal{H}' = \bigwedge_a \left[\bigvee_b E_{ab} \Leftrightarrow \bigwedge_i x_i : \alpha_{ia} \right] \text{ où } E_{ab} \text{ est un environnement.}$$

Chaque crochet de $\mathcal{H}'$ correspond à une possibilité de typage de la famille $\{x_1..x_n\}$. Nous avons donc défini l'ensemble des environnements dans lesquels la famille $\{x_i\}$ est typée.
Il nous faut maintenant déterminer parmi ces environnements ceux dans lesquels le terme lui même est bien typé.

Pour que le terme $f(x_1, \cdots, x_n)$ soit bien typé dans l'environnement E_{ab}, il faut que la contrainte $\mathcal{C}$ soit vérifiée, donc :

$$\alpha_{1a}..\alpha_{na} \leq_c \alpha_1..\alpha_n$$

Le type de la famille $\{x_i\}$ pour lequel $f(x_1, \cdots, x_n)$ est bien typé est donc :

$$\bigwedge_{\substack{a \\ \alpha_{ia} \leq_c \alpha_i}} \left[\bigvee_b E_{ab} \Leftrightarrow \bigwedge_i x_i : \alpha_{ia} \right] \text{ où } E_{ab}$$

On sait que

$$\alpha_{1a}..\alpha_{na} \leq_c \alpha_i..\alpha_n \Leftrightarrow f(x_1, \cdots, x_n) : \alpha_{n+1}$$

donc

$$\left[\bigwedge_{\substack{\alpha_{ia} \leq_c \alpha_i \\ 1 \leq_c i \leq_c n}} \left[\bigvee_b E_{ab} \Leftrightarrow \bigwedge_i x_i : \alpha_{ia} \right] \right] \Leftrightarrow \left[\bigwedge_{\substack{\alpha_{ia} \leq_c \alpha_i \\ 1 \leq_c i \leq_c n}} \left[\bigvee_b E_{ab} \Leftrightarrow f(x_1, \cdots, x_n) : \alpha_{n+1} \right] \right]$$

Une fois ce calcul effectué pour chaque type fonctionnel de f, on obtient le type de $f(x_1, \cdots, x_n)$ sous la forme

$$\bigvee_p \left[\bigwedge_a \left[\bigvee_b E_{abp} \Leftrightarrow x : \alpha_{ap} \right] \right].$$

Dans un environnement donné E_{abp} un terme ne doit posséder qu'un seul type, or il est possible qu'après la calcul précédent cette contrainte ne soit pas respectée.
Or si dans l'environnement C le terme $f(x_1, \cdots, x_n)$ possède plusieurs types, ces types sont nécessairement dans la même chaîne, il nous faut alors parmi tous ceux là prendre le plus petit. Il reste ensuite à regrouper les environnements dans lequel $f(x_1, \cdots, x_n)$ a le même type, et on obtient un type de la forme

$$\bigwedge_k \left[\bigvee_p E_{kp} \Leftrightarrow f(x_1, \cdots, x_n) : \alpha_k \right].$$

B.2 Types fonctionnels : canonicité du A-typage

Pour typer le terme $f(x_1, \cdots, x_n)$ une fois les x_i typés, il faut typer successivement ses sous-termes généralisés directs. On peut se demander si ce calcul est unique quel que soit l'ordre dans lequel sont typés les sous-termes généralisés directs.
Dans le cas non associatif, il n'y a qu'un seul terme à typer. Pour qu'il n'y ait pas d'ambigüité, il suffit donc que deux types d'un même opérateur n'aient pas même type initial.

Par contre si l'opérateur est associatif, le résultat dépend en général de l'ordre de typage des sous-termes généralisés directs.

Exemple 7 *Hypothèses :*

Types	*Constantes typées :*	*Types de* $+$
E(ntier)	$e : E$,	$EP \to P$
P(olynôme)	$p : P$,	$PF \to F$
F(raction)	$f : F$	

Sous les hypothèses précédentes, le terme $e+p+f$ *est typé comme* $(e+p)+f$ *en une* $F(raction)$ *ou comme* $e+(p+f)$ *en un terme de la forme* $e+f'$ *où* $f' : F$ *qui ne peut être typé.*
On voit qu'il manque à $+$ *le type* $EF \to F$ *pour typer ce terme quelque soit le parenthésage.*

Ce problème ressemble fort à celui de la convergence en réécriture. Nous allons le définir et le résoudre en utilisant le formalisme de la réécriture, puis

voir ce que la solution signifie pour le typage.

Il nous faut définir les termes à réécrire et les règles de réécriture à utiliser. Dans un contexte donné, tout terme $f(x_1 : \alpha_1, \cdots, x_n : \alpha_n)$ peut être vu comme un mot $\alpha_1 ... \alpha_n$ sur l'alphabet des types atomiques A, et tout type fonctionnel $\alpha_1\alpha_2 \to \alpha_1$ de f comme une règle de réécriture. Nous allons donc regarder le problème dérivé qu'est la réécriture de mots sur un alphabet de types.

Exemple 8 *Sous les hypothèses de l'exemple précédent, typer le terme $e + p + f$ peut être vu comme réécrire le mot EPF par les règles $EP \to P$ et $PF \to F$.*

Problème B.29 *Est-ce que le système composé des types d'un opérateur vus comme des règles de réécriture est convergent ?*

Choix IX *Soit A l'alphabet des types atomiques du système, et R l'ensemble de toutes les règles dérivées des types formels, nous supposons que le système R termine.*
Ce choix se justifie en disant que les règles dérivées des types diminuent la longueur des mots. C'est en général le cas car leur partie gauche est un type produit et contient donc plusieurs types alors que leur partie droite n'en contient plus qu'un, sauf en ce qui concerne les opérateurs constants (arité 0) ou unaires (arité 1) qui doivent être examinés séparément.

De même qu'un ensemble de règles prises au hasard n'est pas confluent, un tel système a peu de chance d'être confluent car l'utilisateur ne définit que les interprétations qu'il juge nécessaires. Un mot peut donc être réécrit en plusieurs mots différents et irréductibles.

Exemple 9 *$A = \{E, P, F\}$ et $R = \{EP \to P; PF \to F\}$ le mot EPF se réécrit en F ou en EF.*

Pour obtenir la confluence, il faut donc compléter l'ensemble des types fonctionnels.

Exemple 10 *Dans l'exemple précédent, la seule superposition est EPF et se calcule de deux façons,*

$$\begin{array}{ccc} & EPF & \\ \swarrow & & \searrow \\ PF & & EF \\ \searrow & & \\ & F, & \end{array}$$

il faut donc ajouter au système R la règle $EF \to F$ pour qu'il soit confluent.

Par conversion entre types, certaines règles peuvent être impliquées par d'autres règles.

Exemple 11 *Nous ajoutons aux hypothèses de l'exemple page 117 une conversion :*

$$E(ntier) <_c P(olynôme).$$

Le terme $e+p+f$ est calculé comme $(e+p)+f$ en une fraction et comme $e+(p+f)$ en une fraction puisqu'on peut convertir e en P.
Dans ce cas il n'est pas nécessaire de rajouter l'interprétation de type fonctionnel $EF \to F$ car elle est induite par la règle $PF \to F$.

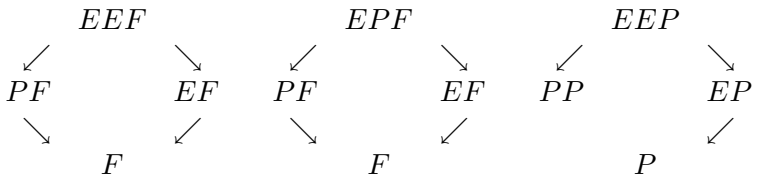
Définition B.30 *On dit que le type $\alpha_1...\alpha_n \to \alpha_{n+1}$ est* **sous-entendu** *par $\alpha'_1...\alpha'_n \to \alpha_{n+1}$ si et seulement si $\alpha_1...\alpha_n <_c \alpha'_1...\alpha'_n$.*

Il ne faut donc pas compléter l'ensemble des types, mais l'ensemble des types et leurs types sous-entendus.

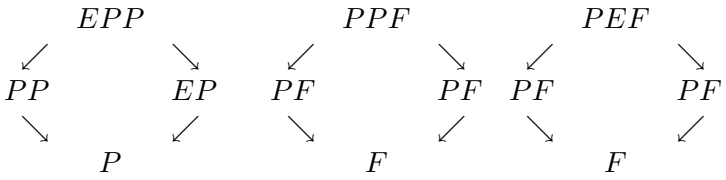
Exemple 12 *Dans l'exemple précédent, l'ensemble de règles à compléter est*

$$R = \{EP \to P; EE \to P; PF \to F; EF \to F\}$$

Les paires critiques possibles sont :



L'ensemble R est complété en $R' = R \cup \{PP \to P, PE \to P\}$ et les paires critiques possibles sont :



Le système de règles R' est alors convergent.

Une fois l'ensemble convergent de types calculé, il suffit de déterminer l'ensemble des types qui ne sont pas déjà sous-entendus pour avoir l'ensemble convergent des types modulo conversions.

On peut toujours supposer que le typage associatif est canonique, puisque s'il ne l'est pas on peut le compléter pour qu'il le soit. On peut donc décider de typer un terme associatif en prenant les sous-termes généralisés directs dans n'importe quel ordre. Pour que l'utilisateur puisse comprendre le typage, on peut donc choisir de typer en prenant les sous-termes directs deux à deux de gauche à droite.

Appendix C

Définition syntaxique des opérateurs

Lorsqu'on travaille dans un système de calcul formel, on s'attend à avoir des entrées et sorties conviviales. La forme préfixe des termes ne correspond pas à cette nécessité, il faut donc que le lecteur et l'imprimeur "connaissent" des informations sur ce qu'ils lisent et impriment.

C.1 Lecture et impression des termes

Lorsqu'on autorise la définition d'un opérateur d'un point de vue algébrique, il paraît naturel d'autoriser aussi sa définition d'un point de vue syntaxique. Par exemple, si un opérateur est associatif, on veut en général le considérer comme infix.

La lecture se fait caractère par caractère, chaque caractère est typé suivant l'alphabet auquel il appartient par les types : numérique, lettre, espace, non alphanumérique ...
L'analyseur lexical regroupe ces caractères suivant leur type pour construire des unités lexicales : identificateur, opérateur (caractère non alphanumérique), nombre ...

Ensuite, l'analyseur syntaxique construit des "phrases" composées d'un opéra-

teur et d'opérandes. Les opérateurs durant l'analyse syntaxique sont des opérateurs lexicaux ou des identificateurs à fonction d'opérateur. L'analyseur syntaxique a besoin d'informations sur ces identificateurs. Notre but n'étant pas de définir complètement un analyseur syntaxique, nous allons montrer son action sur des exemples.

Exemple 13

Analyse lexicale[1]	*Connaissances*	*Analyse syntaxique*[2]
$x;+;y$	$infix(+)$	$(interne(+)\ x\ y)$
$f;(;x;,;y;)$	$matchfix(()$	$(interne(f)\ x\ y)$
	$infix(,)$	
$(;x;+;y;);/;z$	$matchfix(()$	$(interne(/)\ (interne(+)\ a\ b)\ d)$
	$infix(+)$	
	$infix(/)$	
$x;+;y;+;z$	$n\text{-}aire(+)$	$(interne(+)\ x\ y\ z)$
$-;x$	$prefix(-)$	$(interne(-)\ x)$
$x;+;y;*;z$	$infix(+)$	$(interne(+)\ x\ (interne(*)\ y\ z))$
	$infix(*)$	
	$rbp(+) < lbp(*)$	
$x;*;y;+;z$	$infix(+)$	$(interne(+)\ (interne(*)\ x\ y)\ z)$
	$infix(*)$	
	$lbp(+) < rbp(*)$	

Remarque 14 *Il est possible de donner à une même image d'entrée plusieurs images internes si elles sont syntaxiquement différentiables grâce à leurs modes d'insertion.*

De même lors de l'impression, le système est amené à faire la transformation inverse de celle décrite précédemment.

[1] Les unités lexicales sont séparées par des ";".

[2] Les phrases syntaxiques sont notées comme des listes dont le premier élément est l'opérateur et les autres les opérandes.

Exemple 15

$(+\ x\ y)$	$x; sortie(+); y$
$(f\ x\ y)$	$f; sortie((); x; sortie(,); y; sortie())$
$(/\ (+\ x\ y)\ z)$	$sortie((); x; sortie(+); y; sortie()); sortie(/); z$
$(+\ x\ y\ z)$	$x; sortie(+); y; sortie(+); z$
$(-\ x)$	$sortie(-); x$
$(+\ x\ (*\ y\ z))$	$x; sortie(+); y; sortie(*); z$
$(+\ (*\ x\ y)\ z)$	$x; sortie(*); y; sortie(+); z$

Remarque 16 *En changeant la représentation de sortie des opérateurs, on peut "générer" des expressions dans des langages différents. C'est à dire changer la syntaxe et l'image de sortie de l'opérateur.*

C.2 Lecture et impression des constantes spécifiées

Les constantes spécifiés peuvent être construites à deux moments, soit lors de la lecture, soit lors de l'évaluation.

Lors de l'analyse lexicale, le lecteur peut typer certaines unités suivant les caractères qu'ils contiennent, par exemple les entiers, les réels, les chaînes ... Si pour chaque type formel il existe une représentation spécifique standard, et qu'à chaque type d'unité lexicale reconnue correspond un type formel, le système peut construire un objet spécifié dans cette représentation. Ainsi les objets spécifiés de base sont construits automatiquement, mais il reste possible d'en changer la représentation en changeant de type standard.

Les objets spécifiés construits à partir de phrases et non d'unités lexicales ne peut l'être que sur une demande explicite. Par exemple, pour construire un polynôme de valeur $x + y$, il faut écrire $pol(x + y)$ si l'opérateur *pol* possède comme interprétation le constructeur des polynômes.

Une fois une constante spécifiée construite et calculée, le système l'imprime. Il faut donc qu'il transforme l'objet typé en une phrase syntaxique, par exemple le polynôme de valeur $x + y$ doit être transformé en (+ x y) de manière à ce que le système sache l'imprimer.

Pour chaque type de $\mathcal{S}$, il faut donc posséder deux fonctions l'une qui transforme une phrase syntaxique en objet spécifié (lecteur ou constructeur) et l'autre qui fasse l'inverse (imprimeur ou destructeur).

C.3 Conclusion

Pour permettre à l'utilisateur de rendre les lecteur et imprimeur plus conviviaux en les faisant correspondre à ces notations, il faut lui permettre de définir les représentations d'entrée/sortie de ses opérateurs et les mémoriser dans la base. On peut remarquer que certains opérateurs tels que "(", ")" ou ";" (fin d'expression) sont utilisés par le système il faut donc sous peine d'obtenir un dysfonctionnement que ces opérateurs soient définis.

Les informations permettant de définir syntaxiquement les opérateurs sont :

1. image interne,
2. image de sortie,
3. mode d'insertion : infix (/), n-aire ($+, *, \cdot$), prefix (-), postfix (!), matchfix ([,(),
4. puissance de liaison à droite : rbp,
5. puissance de liaison à gauche : lbp,

Appendix D

Athéna

Les ensembles utilisés en mathématique sont très souvent construits à partir d'ensembles de base. On dispose pour cela de méthodes dont les plus simples et les plus usuelles sont celles de la théorie des ensembles classique : produit cartésien, applications d'un ensemble dans un autre, quotient par une relation d'équivalence.

On peut remarquer que de nombreuses structures algébriques se transportent par ces constructeurs (produits de groupes, espaces de fonctions, quotients d'algèbres, etc) : les opérations de la structure complexe sont clairement définies à partir de celles des sous-structures.
De plus les structures de données usuelles pour les éléments d'un produit cartésien, d'une puissance (applications d'un ensemble dans un autre), d'un quotient sont en nombre assez limité et bien connu.
On peut donc construire automatiquement des ensembles complexes avec ces constructeurs de base, c'est à dire définir la représentation spécifique de leurs éléments, et leurs interprétations.

Cette idée a été mise en pratique dans Ulysse par Loïc Pottier en construisant une base spécifique.
Afin de décrire simplement cette application on utilisera le langage de la logique classique, et en particulier la notion de **terme** et de **modèle** qui sont à la base de la sémantique des langages de programmation.
Brièvement, on peut dire qu'un terme est une expression formelle arborescente construite avec des symboles de fonctions et des variables : $(1 + x) * y$

et qu'un modèle est un ensemble sur lequel sont définies des fonctions satisfaisant des axiomes (c'est la notion classique de la théorie des modèles) : Z avec l'addition et la multiplication et les axiomes d'anneau.
Les modèles que nous définissons sont **effectifs** :

- leurs éléments sont associés à des représentations spécifiques,
- leurs fonctions sont des algorithmes manipulant ces représentations spécifiques.

Pour interpréter un terme dans un modèle, il faut

1. associer à chaque symbole fonctionnel une fonction du modèle, et à chaque variable un élément du modèle,
2. rendre l'application de ces fonctions.

Pour évaluer chaque terme dans un système de calcul formel tel que celui que nous avons décrit, il faut l'interpréter dans un modèle.
Lors de la construction d'une algèbre, il faut donc mémoriser une représentation spécifique (nom de représentation, lecteur, imprimeur et comparateur), une interprétation et une liste de propriétés pour chaque opérateur défini dans cette algèbre.
Comme les algèbres doivent être définies dans le système, il faut donc que la base contienne aussi les fonctions permettant de les construire.

D.1 Sémantique

Pour introduire ce mécanisme d'évaluation, une nouvelle structure de $\mathcal{S}$-type a été définie : celle de modèle.
Les types de $\mathcal{S}$ n'étant plus représentés sous forme de symboles Le-Lisp, le mécanisme de $\mathcal{S}$-typage a été modifié, mais nous ne le décrirons pas ici.
D'autre part nous n'avons pas implémenté de lien entre les types de $\mathcal{S}$ et les types formels car pour l'instant nous n'avons pas essayé d'introduire de conversions, le typage formel est donc inhibé.
Les termes sont donc calculés par simplification modulo propriétés et évaluation.

Un modèle sert à typer ses éléments, et possède lui même un type, qui est une **signature**, i.e. un ensemble de symboles formels dénotant ses algorithmes (par exemple $\{+, 0, -\}$ pour une groupe additif). Cette signature est elle même typée par un *schéma de signatures*, qui est un ensemble de types fonctionnels (par exemple $\{\rightarrow X, XX \rightarrow X, X \rightarrow X\}$ pour les opérations d'un groupe) décrivant les ensembles de départ et d'arrivée des fonctions des modèles concernés (ceux dont le type a pour type ce schéma), et d'axiomes qui décrivent les propriétés équationnelles des fonctions de ces modèles (par exemple les axiomes des groupes).

En résumé, un modèle contient les opérations possibles sur une structure de donnée particulière, sa signature permet d'écrire des formules où interviennent ces opérations, et son schéma caractérise les formules correctes (types fonctionnels) et leurs propriétés formelles (axiomes). Dans les sections suivantes, nous détaillons ces notions.

Modèle

Un modèle est donc un ensemble sur lequel des opérations sont définies, utilisant éventuellement d'autres modèles (p.ex. la multiplication externe d'un espace vectoriel utilise le corps des coefficients).

Un modèle est représenté par une structure Le-Lisp qui contient outre les fonctions Le-Lisp implémentant ses opérations, des informations annexes :

- un nom (un symbole Le-Lisp),
- une description de la structure de données Le-Lisp utilisée pour implémenter ses éléments (représentation creuse, dense, pour les polynômes),
- un lecteur, permettant de transformer un terme en un élément du modèle (le terme $5 * X^4 + 9 * X - 7$ sera transformé en ((5 . 4)(9 . 1)(-7 . 0)) si le modèle est une algèbre de polynômes d'implémentation creuse en Le-Lisp),
- un imprimeur, faisant le travail inverse du lecteur,
- des modèles auxiliaires (p.ex. le corps de base d'un espace vectoriel),

- éventuellement un algorithme de décision d'appartenance si le modèle est récursif, ou un algorithme d'énumération s'il est récursivement énumérable,
- ...

Chaque élément d'un modèle porte comme type le nom de ce modèle.

Signature

Un modèle est typé par une **signature** regroupant les symboles utilisés pour nommer ses opérateurs :

$$Z : F\_Ring$$

où $F\_Ring = (+, -, 0, *, 1/, 1)$.

Schéma de signature

Une signature est typée par un **schéma de signatures** qui contient les types fonctionnels de ses opérateurs (affublés d'un nom "générique"), ainsi que les axiomes qu'ils vérifient :

$$F\_Ring : RING$$

où $RING = \{(law, sym, unit, mul, inv, unit\_mul), \ldots \text{axiomes des anneaux} \ldots\}$.

Un schéma de signatures sera de type le symbole **scheme**. Comme pour les éléments d'un modèle, les modèles, signatures, schémas portent leur type.

Les signatures et les schémas ont une structure d'ensemble fini, aussi on utilisera un héritage non automatique par inclusion (p.ex. $GROUP \subset RING$).

Exemple 17 *Voici le modèle des entiers relatifs (Z)* in extenso *:*

```
(1): describe(‘Z);

                        [    ntype = F_Ring   ]
                        [                     ]
```

```
                        [       name = Z      ]
                        [                     ]
                        [read_el = #:ul:read_Z]
                        [                     ]
                        [  prin_el = identity ]
                        [                     ]
                        [   equal = #:ul:Z=   ]
                        [                     ]
1                       [  order = #:ul:Z<?>  ] : T
                        [                     ]
                        [    law = #:ul:Z+    ]
                        [                     ]
                        [        unit = 0     ]
                        [                     ]
                        [    sym = #:ul:Z-    ]
                        [                     ]
                        [    mul = #:ul:Z*    ]
                        [                     ]
                        [     unit_mul = 1    ]
```

Z est lui même un objet typé, constitué d'un ensemble d'équations (<opération> = <algorithme>). Les noms d'algorithmes sont des symboles packagés de Le-Lisp comme par exemple `#:ul:read_Z`.

Son type F_Ring contient sa signature (les symboles nommant ses algorithmes) :

```
(2): describe('F_Ring);

                        [       ntype = RING       ]
                        [                          ]
                        [      name = F_Ring       ]
                        [                          ]
                        [      read_el = read      ]
                        [                          ]
                        [      prin_el = prin      ]
                        [                          ]
                        [        equal = =         ]
                        [                          ]
                        [     order = compare      ]
2                       [                          ] : T
```

```
                        [         law = +          ]
                        [                          ]
                        [         unit = 0         ]
                        [                          ]
                        [         sym = -          ]
                        [                          ]
                        [         mul = *          ]
                        [                          ]
                        [       unit_mul = 1       ]
                        [                          ]
                        [inherit = F_Additive_Group]
```

Le type RING de la signature F_Ring est un schéma de signature*, qui est un ensemble d'opérations génériques, avec leurs types fonctionnels (la variable* `#ALG` *représente le modèle de travail, comme le* `$` *d'Axiom) :*

```
(3): describe(‘RING);
```

```
                [                  ntype = SCHEME              ]
                [                                              ]
                [      parameters = (C_Ring_axioms(#ALG))      ]
                [                                              ]
                [                   name = RING                ]
                [                                              ]
                [    read_el = ([#X] --> (#ALG) In (#ALG))     ]
                [                                              ]
                [    prin_el = ([#ALG] --> (#X) In (#ALG))     ]
                [                                              ]
                [equal = ([#ALG , #ALG] --> Boolean In (#ALG))]
                [                                              ]
3               [  order = ([#ALG , #ALG] --> Z_3 In (#ALG))   ] : T
                [                                              ]
                [ law = ([#ALG , #ALG] --> (#ALG) In (#ALG))   ]
                [                                              ]
                [       unit = ([] --> (#ALG) In (#ALG))       ]
                [                                              ]
                [     sym = ([#ALG] --> (#ALG) In (#ALG))      ]
                [                                              ]
                [ mul = ([#ALG , #ALG] --> (#ALG) In (#ALG))   ]
```

```
[                                                    ]
[     unit_mul = ([] --> (#ALG) In (#ALG))           ]
[                                                    ]
[                inherit = GROUP                     ]
```

Voici un élément du modèle Z :

```
(3): 2343:Z;

3                                    2343 : Z

(4): describe(2343:Z);

                                 [Value = 2343]
4                                [             ] : T
                                 [ Model = Z   ]
```

D.2 Construction automatique de modèles

A partir de modèles (ensembles) de base, on peut en construire des modèles plus compliqués, en utilisant des *constructeurs de modèles*, dont les plus importants sont :

- le produit cartésien,
- l'exponentiation (applications d'un modèle dans un autre),
- le quotient par une relation d'équivalence.

Nous avons choisi ces constructeurs car ils préservent souvent les structures algébriques (produits de groupes, algèbres de fonctions, anneaux quotients, etc), ce qui permet d'engendrer *automatiquement* les algorithmes des modèles construits, en utilisant des structures de données prédéfinies adaptées. Nous allons décrire maintenant les constructeurs disponibles dans Ulysse, avec pour chacun d'eux son fonctionnement, les structures de données possibles, et un exemple.

Produit cartésien

Etant donnés deux modèles A et B, on construit le produit cartésien ensembliste $A \times B$, dont les éléments sont représentés par des `cons` de Le-Lisp : `(a . b)`. Le produit $A.B$ est muni des opérations communes à A et à B, i.e. celles qui ont le même type fonctionnel dans les schémas de A et B. Ces opérations agissent sur chaque composante des couples comme celles de A ou B : $f((a.b)) = (f_A(a).f_B(b))$.

Exemple 18 *Le produit $Z \times Q[X]$ du groupe des entiers et de l'algèbre de polynômes en X à coefficients rationnels donne un groupe additif commutatif $Z \times Q[X]$ où par exemple*

$$(3.X^2 + X - 1) + (-3.1 - X) = (0.X^2).$$

Puissance

B^A désigne l'ensemble des applications de A dans B et les opérations *op* de B sont alors définies dans B^A par :

$$\forall x \in A, \quad op(f_1, \ldots, f_n)(x) = op(f_1(x), \ldots, f_n(x))$$

Les structures de données possibles ici sont assez variées, voici les principales :

dense : Dans le cas où A est fini et indexé par un intervalle d'entiers $[1, \ldots, n]$, on représente une application f de A dans B par un vecteur Le-Lisp de ses valeurs $\#[f(1), \ldots, f(n)]$.

creuse : On se restreint aux applications de A dans B qui ont pour valeur un élément distingué de B (élément neutre, ...) sauf pour un ensemble fini d'éléments de A. L'application est alors représentée par la partie non diagonale de son graphe qui est donc finie, sous forme d'une liste d'association ordonnée de Le-Lisp : $(\ldots, (x.f(x)), \ldots)$.
Notons au passage que les modèles d'Ulysse sont munis d'un ordre total pour des raisons d'efficacité des algorithmes mais qu'il n'est pas nécessaire par exemple que cet ordre soit compatible avec les opérations de l'ensemble.

paresseuse : Dans le cas général A est éventuellement infini. On représente une fonction de A dans B par un algorithme, i.e. une fonction Le-Lisp. On gardera mémoire des valeurs déjà calculées de la fonction, au moyen d'une liste d'association. Ainsi un élément f de B^A sera un couple $(fonction - Le - Lisp.(\ldots,(x.f(x)),\ldots))$.

D'autres représentations plus complexes (semi-dense, récursive, ...) ont été implémentée mais elles ne sont pas décrites ici.
Il est possible que des structures particulières de A et B permettent d'enrichir la structure de B^A. Par exemple si A est un monoïde et B une algèbre, alors B^A est naturellement muni d'une structure d'algèbre différente de celle induite par celle de B.

> **Exemple 19** *Le Z-module libre M engendré par un ensemble fini de générateur $G = \{g_1, \ldots, g_n\}$ sera construit comme Z^G. Si on choisit la représentation dense l'élément $2g_1 + 3g_2 - g_3$ de M sera représenté par le vecteur* `#[2 3 -1]`.
>
> *L'algèbre $Q[[X]]$ des séries formelles en X à coefficients dans Q sera construit comme N^Q. Si on choisit la représentation paresseuse, la série entière de l'exponentielle sera représentée par (*`algo` *((0 . 1) (1 . 1) (2 . 1/2) (5 . 1/120))) (où la fonction Le-Lisp* `algo` *est telle que* `algo(n)=1/n!`*) si on a calculé avant ses coefficients d'indices 0,1,2 et 5.*

Quotient

Etant donnée une relation d'équivalence $\sim$ sur un ensemble E on construit le quotient $E/\sim$. On fait l'hypothèse que les opérations de E sont compatibles avec $\sim$, ce quotient a donc les mêmes opérations que E.

Le relation $\sim$ est donnée sous la forme d'un algorithme f (fonction Le-Lisp) de calcul de *forme canonique*, i.e. vérifiant

$$\forall x, y \in E, \quad f(x) = f(y) \Leftrightarrow x \sim y$$

Une classe d'équivalence de E sera alors représentés par l'élément de E qui est la forme canonique de ses éléments.

Les algorithmes du quotient sont alors simples à générer: on met sous forme canonique les arguments et le résultat de l'algorithme de E correspondant.

Exemple 20 *L'anneau des entiers de Gauss $Q[i]$ est construit comme quotient de l'anneau $Q[X]$ par l'idéal engendré par le polynôme X^2+1, et la fonction de mise en forme canonique f calcule le reste de la division euclidienne de son argument par X^2+1.*

Suites finies

A partir d'un ensemble A on construit le monoïde A^* des suites finies d'éléments de A (la loi de composition est la concaténation)dont les éléments sont représentés par des listes Le-Lisp.

Parties finies

A partir d'un ensemble A on construit l'algèbre $P(A)$ des parties finies de A dont les éléments sont représentés par des listes sans répétitions ordonnées. Remarquons qu'on peut voir aussi $P(A)$ comme l'algèbre des applications presque partout nulles de A dans $Z/2Z$, constructible avec une puissance $(Z/2Z)^A$ en représentation creuse.

Exemple 21 *Voici des exemples de structures classiques construites à partir des constructeurs précédents :*

Polynômes *$Z[X_1,\ldots,X_n]$ est construit comme : $Z^{\left(N^{\{X_1,\ldots,X_n\}}\right)}$. Si on veut que l'ordre utilisé pour ordonner les monômes soit celui du degré il suffit de construire : $Z^{\left(Z\times N^{\{X_1,\ldots,X_n\}}\right)}$.*

Algèbre tensoriel *L'algèbre tensorielle d'un Z-module libre de générateurs G est construite comme : Z^{G^*}.*

Le corps des complexes *On le construit comme un quotient d'anneau de polynômes à une variable i :*

$$\frac{R^{\left(N^{\{i\}}\right)}}{p \mapsto remainder(p, i^2+1)}$$

Construction du produit cartésien $Z \times Z$ dans Ulysse :

```
(1) Z2:=‘Z . ‘Z;

1                                    Z . Z

(2) describe(‘Z2);

                     [       ntype = F_Ring   ]
                     [                        ]
                     [        name = ZxZ      ]
                     [                        ]
                     [read_el = #:ul:cons:read]
                     [                        ]
                     [prin_el = #:ul:cons:prin]
                     [                        ]
                     [      equal = g262      ]
                     [                        ]
2                    [      order = g263      ] : T
                     [                        ]
                     [       law = g259       ]
                     [                        ]
                     [     unit = (0 . 0)     ]
                     [                        ]
                     [       sym = g260       ]
                     [                        ]
                     [      mul = g261        ]
                     [                        ]
                     [   unit_mul = (1 . 1)   ]

(2)  a:=[1,2]:(Z.Z);

2                              [1 , 2] : (Z . Z)

(3) ‘a + ‘a;

3                              [2 , 4] : (Z . Z)
```

Appendix E

Notations, définitions et choix

Notations

$\mathcal{O}$ Ensemble des symboles d'opérateur contenus dans les termes.

$\mathcal{C}$ Ensemble des constantes symboliques ou non symboliques contenus dans les termes.

$\mathcal{R}$ Ensemble des variables de réécriture.

$\mathcal{I}$ Ensemble des variables d'instanciation.

$\mathcal{S}$ Ensemble des représentations spécifiques.

$\mathcal{C}_{\mathcal{S}}$ Ensemble des constantes spécifiées contenues dans les termes.

$<_c$ Relation d'ordre partiel entre types.

$\mathcal{F}$ Ensemble des types formels.

$\rightarrow$ Relation "est le type formel de" entre un type formel et un type spécifique.

A Ensemble de types, représentant dans notre cas $\mathcal{S}$ ou $\mathcal{F}$.

Première partie

Terme, opérateur de tête, sous-terme 28

Une expression encore appelée terme est un arbre dont les noeuds sont étiquetés par des symboles d'opérateur et les feuilles par des constantes symboliques ou non symboliques.
"Le symbole qui étiquette la racine de l'arbre s'appelle la racine du terme (ou opérateur de tête). A un sous-arbre correspond alors un sous-terme."

Variable et règle de réécriture 28

$\mathcal{O}$: Ensemble des symboles d'opéra- teur.

On distingue un symbole unaire de $\mathcal{O}$ noté V, tel que les termes de racine V sont appelés variables de réécriture.
Une règle de réécriture est un couple de termes (t_1, t_2) noté $t_1 \rightarrow t_2$ tels que les variables de t_2 sont toutes des variables de t_1.

Système de réécriture, réécriture 29

Un ensemble de règles de réécriture est appelé système de réécriture, et le *remplacement d'égaux par des égaux* en utilisant des règles de réécriture (de la gauche vers la droite) s'appelle la réécriture.

Sous-terme généralisé 31

Soient $e_1 ... e_n$ les équations modulo lesquelles se fait la réécriture. Soit t un terme appartenant à une classe d'équivalence C modulo $e_1 ... e_n$, tout sous-terme d'un terme appartenant à C est un sous-terme généralisé de t.

Sous-terme direct 34

On appelle sous-terme direct d'un terme un des fils de la racine de l'arbre qui le représente.

Stgd, sous-terme généralisé direct 34

On appelle sous-terme généralisé direct d'un terme t tout sous-terme direct d'un terme équivalent à t.

Choix I 35

Nous proposons de considérer la simplification comme un calcul modulo associativité et commutativité.

Choix II 35

Nous choisissons de ne considérer que les sous-termes généralisés directs binaires des termes associatifs.

Propriété 36

Nous appellerons propriété un couple de termes $(f(x_1, \cdots, x_n), g)$ noté
$f(x_1, \cdots, x_n) \rightarrow g$ dans lequel $f \in \mathcal{I}$ et les termes $x_1 ... x_n, g$ sont construits à partir de variables de $\mathcal{R}$ ou de $\mathcal{I}$, de symboles d'opérateurs et de constantes.

$\mathcal{I}$: Ensemble des variables d'instanciation.
$\mathcal{R}$: Ensemble des variables de réécriture.

Propriété d'un opérateur, paramètre 37

On dit qu'on associe la propriété P à l'opérateur *op* lorsqu'on instancie P en donnant à f la valeur *op*. Une propriété d'un opérateur est parfaitement définie si on connaît le nom de l'opérateur *op*, la propriété P et les valeurs associées aux variables de $\mathcal{I} \setminus \{f\}$ appelées paramètres (puisqu'elles doivent toutes être instanciées).

Choix III 45

Les fonctions de calcul ne doivent pas effectuer de transformations descriptibles par des règles.

Deuxième partie

Choix IV 55

Nous partageons en deux classes les représentations possibles des expressions mathématiques dans un système de calcul formel : la représentation sous forme d'arbre et les représentations étudiées en fonction d'algorithmes spécifiques.

Représentation générale, spécifique 55

Nous qualifierons la représentation sous forme d'arbre de générale alors que les autres représentations seront qualifiées de spécifiques.

Constante ou objet spécifié 55

Les constantes non symboliques de $\mathcal{C}$ sont des termes dans une représentation spécifique, nous les appellerons constantes ou objets spécifiés.

Type formel 57

Un type formel est le nom d'une classe d'équivalence de représentations spécifiques $\mathcal{S}$ associée à la relation $<_c$. Un type formel est donc un nom T associé à des types de $\mathcal{S}$ $s_1 ... s_k$.

$\mathcal{S}$: Ensemble des représentations spécifiques.

Ordre sur les types formels 57

Deux types formels T_i et T_j sont ordonnés en $T_i <_c T_j$ si et seulement si $\forall s_i, s_j$ tels que $T_i \rightarrow s_i$ et $T_j \rightarrow s_j$ $s_i <_c s_j$.

$<_c$: Relation d'ordre partiel sur $\mathcal{S}$.

$\rightarrow$: Relation entre un type formel et un type spécifique.

Chaîne de conversions 57

On appelle chaîne de conversions un sous-ensemble de types formels totalement ordonné (par $<_c$).

Choix V 58

Nous ne considérons que les ensembles de types formels partitionnés en chaînes de conversions.

***A*-type fonctionnel, type initial, type final** 62

On appelle A-type fonctionnel tout type de la forme $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$ où $\forall i \in [1..n+1]$ $\alpha_i \in A$. Le type produit $\alpha_1 \times ... \times \alpha_n$ est appelé type initial, alors que α_{n+1} est appelé type final.

Opérateur générique, $\mathcal{F}$-typé 62

On appelle opérateur générique un symbole d'opérateur associé à plusieurs $\mathcal{F}$-types fonctionnels, et opérateur $\mathcal{F}$-typé un symbole d'opérateur associé à un unique $\mathcal{F}$-type fonctionnel.

Choix VI 70

On considère que seuls les termes dont tous les sous-termes sont spécifiés peuvent être évalués.

Interprétation, fonction de calcul élémentaire 71

On appelle **interprétation** tout couple formé d'un $\mathcal{S}$-type fonctionnel
$s_1..s_n \rightarrow s_{n+1}$ et d'une fonction f de calcul élémentaire d'arité n. La fonction f prend en entrée n constantes de $\mathcal{C}_\mathcal{S}$ de types $s_1..s_n$ et rend une constante de $\mathcal{C}_\mathcal{S}$ de type s_{n+1}.

$\mathcal{C}_\mathcal{S}$: Ensemble spécifiées. des constantes

Troisième partie

Choix VII 78

Nous choisissons de différentier explicitement les informations : propriétés, règles, interprétations, types ..., des mécanismes de calcul : simplification, réécriture, évaluation, $\mathcal{F}$-typage ...

Base, moteur 78

Les informations nécessaires au calcul sont mémorisées dans une base (ensemble de données évolutives) alors que les composants dirigés par un contrôleur forment le moteur.

Domaine 86

Un domaine est un type associé un ensemble de couples (symbole d'opérateur, fonction de calcul).

Opérateur 86

Un opérateur est un symbole associé à des propriétés ainsi qu'à des interprétations.

Choix VIII 86

Nous avons choisi de structurer la base autour de la notion d'opérateur et non de celle de domaine.

Bibliography

[BBD+91] Manuel Bronstein, William H. Burge, Timothy P. Daly, Patrizia Gianni, Johannes Grabmeier, Richard D. Jenks, Scott C. Morrison, Jonathan M. Steinbach, Robert S. Sutor, Barry M. Trager, Stephen M. Watt, and Clifton J. Williamson. *Axiom User Guide*, 1991.

[CAJL] C.Faure, A.Galligo, J.Grimm, and L.Pottier. The extensions of the sisyphe computer algebra system : `ulysse` and `athena`. Soumis à ISSAC'92.

[CGGW88] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, and Stephen M. Watt. *Maple User's Guide*, 1988.

[CR73] C. Chang and R.Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics. Academic Press, Inc., 1973.

[Dal91] S. Dalmas. *Un langage fonctionnel polymorphe. Application aux problèmes logiciels du calcul formel.* PhD thesis, Université de Nice Sophia-Antipolis, 1991.

[Dav91] J.H. Davenport. The axiom system. 1991.

[DJ90] N. Dershowitz and J-P. Jouannaud. *Rewrite systems.*, pages 244–320. Elsevier Science Publishers, 1990.

[GGP90] A. Galligo, J. Grimm, and L. Pottier. The design of sisyphe. In *Proc. of DISCO*. Springer Verlag, 1990.

[Gro89] Computer Algebra Group. *The Scratchpad Computer Algebra System Interactive Environment Users Guide*, 1989.

[Hea85] Antony C. Hearn. *Reduce user's manual (version 3.2)*, 1985.

[JK91] J-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. 1991.

[JL86] J-P. Jouannaud and P. Lescanne. La réécriture. *Technique et Science Informatique*, 5(6):433–452, April 1986.

[MIT83] MIT. *Macsyma. Reference Manual*, 1983.

[PS81] G. Peterson and M. Stickel. Complete sets of reduction for some equational theories. *J. ACM*, 28(2):233–264, 1981.

[Wol88] Stephen Wolfram. *Mathematica. A System for Doing Mathematics by Computer*, 1988.