Université de Montréal

Abitbol : Un langage sur mesure pour la métaprogrammation

par
Vincent Archambault-Bouffard

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

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

04, 2015

Résumé

Ce mémoire a pour thèse que les fonctions devraient être transparentes lors de la phase de métaprogrammation. En effet, la métaprogrammation se veut une possibilité pour le programmeur d'étendre le compilateur [26]. Or, dans un style de programmation fonctionnelle, la logique du programme se retrouve dans les définitions des diverses fonctions le composant. Puisque les fonctions sont généralement opaques, l'impossibilité d'accéder à cette logique limite les applications possibles de la phase de métaprogrammation.

Nous allons illustrer les avantages que procurent les fonctions transparentes pour la métaprogrammation. Nous donnerons notamment l'exemple du calcul symbolique et un exemple de nouvelles optimisations désormais possibles. Nous illustrerons également que la transparence des fonctions permet de faire le pont entre les datatypes du programme et les fonctions.

Nous allons également étudier ce qu'implique la présence de fonctions transparentes au sein d'un langage. Nous nous concentrerons sur les aspects reliés à l'implantation de ces dernières, aux performances et à la facilité d'utilisation. Nous illustrerons nos propos avec le langage Abitbol, un langage créé sur mesure pour la métaprogrammation.

Mots clés: Fonctions transparentes, métaprogrammation, compilation, langages fonctionnels, réification.

Abstract

Our main thesis is that functions should be transparent during the metaprogramming stage. Metaprogramming is intended as a possibility for the programmer to extend the compiler [26]. But in a functional programming style, the program logic is found in the definition of its functions. Since functions are generally opaque, it is impossible for the programmer to access this information and this limits the metaprogramming possibilities.

We will illustrate the benefits of transparent functions for metaprogramming. We will give the example of symbolic computation and also show new forms of optimizations now available at the metaprogramming stage. We will also illustrate that transparency allows us to bridge the gap between the datatypes of a program and its functions.

We will also examine how transparent functions affects other aspects of the language. We will focus on how to implement them, their performance impact and their ease of use.

We illustrate our thesis with Abitbol, a language designed for metaprogramming.

Keywords: Transparent functions, metaprogramming, compilation, functional languages, reification.

Table des matières

Liste des codes sources

Liste des algorithmes

Liste des grammaires

Liste des sigles

ASA Arbre de syntaxe abstraite

BNF Forme de Backus-Naur

À Marie-Danielle, Loan et Éloi.

Remerciements

Je tiens spécialement à remercier mes directeurs de maîtrise Stefan Monnier et Marc Feeley. Merci de m'avoir fait confiance pour le choix du sujet. Merci de m'avoir donné la latitude nécessaire pour faire de ce travail mon travail. Votre encadrement, vos explications et nos discussions hebdomadaires m'ont grandement apporté.

Merci à Loan qui m'a convaincu par son sourire et sa bonne humeur qu'il y a mieux à faire que de travailler sur la maîtrise les soirs et les fins de semaine. Un homme heureux est un homme qui travaille bien.

Merci à Éloi qui m'a tenu réveillé la nuit à marcher de long en large à la recherche d'un rot ou de son sommeil. Beaucoup d'idées concernant le langage Abitbol me sont venues à ces moments où mon esprit était loin de tous les écrans.

Finalement, une fière chandelle à Marie-Danielle qui a accepté mon retour aux études malgré l'incertitude que cela peut entrainer sur plusieurs aspects de notre vie commune. Merci pour ton support indéfectible. Merci d'avoir cru en moi. Merci pour ton amour.

Chapitre 1

Introduction

1.1 Thèse principale

Ce mémoire se concentre principalement sur les langages permettant un style de programmation fonctionnelle. Dans ce paradigme, les fonctions sont les unités élémentaires qui servent à décrire les calculs. Dans la quasi-totalité des langages fonctionnels, les fonctions sont toutefois des objets opaques. Il est impossible de les déconstruire pour observer leur logique et implantation. Ainsi, bien qu'elles soient souvent qualifiées d'objets de première classe, le programmeur est très limité dans ce qu'il peut faire avec les objets de type fonction. À l'opposé, une fonction transparente permettrait au programmeur d'obtenir sa définition et ainsi de l'analyser comme n'importe quel autre objet.

Toujours dans le paradigme de programmation fonctionnelle, nous pensons que l'opacité des fonctions limite la puissance des mécanismes de métaprogrammation qui permettent au programmeur de générer automatiquement du code au début de la compilation. En effet, la métaprogrammation

se veut une possibilité pour le programmeur d'étendre le compilateur [26]. Or, dans un style de programmation fonctionnelle, la logique du programme se retrouve dans les définitions des diverses fonctions le composant. L'impossibilité pour le programmeur d'accéder à cette information limite donc les applications possibles de la phase de métaprogrammation.

La thèse principale de ce mémoire est donc que les fonctions devraient être transparentes lors de la phase de métaprogrammation.

Nous allons illustrer clairement les avantages que procurent les fonctions transparentes lors de la métaprogrammation. Nous allons également étudier ce qu'implique la présence de fonctions transparentes au sein d'un langage. Nous nous concentrerons sur les aspects reliés à l'implantation de ces dernières, aux performances et à la facilité d'utilisation.

Nous illustrerons nos propos avec le langage Abitbol, un langage créé sur mesure pour ce travail.

1.2 Plan du mémoire

Ce mémoire est structuré de la façon suivante :

- Le reste de ce chapitre énonce les motivations générales qui nous ont poussés à entreprendre ce travail. Nous allons également définir ce que nous entendons par fonctions transparentes et métaprogrammation.

- Le second chapitre traite des travaux antérieurs sur les fonctions transparentes et la métaprogrammation. Cela permet de situer le contexte dans lequel ce mémoire s'insère.

- Le troisième chapitre traite du langage Abitbol. La transparence des fonctions bouscule d'autres aspects du langage comme la présence d'ef-

fets de bord ou le mode d'évaluation. Toutes ces considérations sont analysées dans ce chapitre. Nous donnerons également des détails sur l'implantation des fonctions transparentes et le mécanisme d'expansion de macros.

- Le quatrième chapitre offre deux exemples concrets pour illustrer les avantages des fonctions transparentes. Ces exemples sont écrits dans le langage Abitbol vu au chapitre précédent.

- Le cinquième chapitre comporte une discussion sur la présence des fonctions transparentes lors de la phase de métaprogrammation dans le langage Abitbol. Nous discuterons de la simplicité sémantique de cette approche, des performances et des problèmes d'hygiène. Nous verrons également comment obtenir des fonctions transparentes lors de la métaprogrammation avec le langage Racket. Cette approche sera comparée avec celle d'Abitbol.

- Finalement, la conclusion vient résumer l'essentiel de ce mémoire et donne quelques pistes de travaux futurs.

1.3 Motivation

L'auteur a travaillé dans le courtage électronique d'options sur actions avant d'entreprendre cette maîtrise. Durant ces quelques années, il a été témoin qu'un code simple et facile à maintenir n'est pas nécessairement le plus rapide. La vitesse étant cruciale dans les plates-formes de courtage électronique, le programmeur devait donc réécrire ce type de code. Malheureusement l'optimisation aboutit souvent en un code difficile à comprendre et à maintenir. Bien qu'il soit compréhensible qu'un compilateur ne possède pas nativement

toutes les optimisations liées au domaine de la finance, ne serait-il pas possible « d'enseigner » ces optimisations au compilateur ? Ainsi, le compilateur pourrait les appliquer à l'ensemble des programmes de courtage et le code tel que vu par le programmeur serait simple et facile à maintenir. C'est là l'essence de ce mémoire.

En d'autres termes, notre motivation est d'économiser du temps au programmeur, libérer ce dernier de la gestion des détails et encapsuler des connaissances d'experts comme les optimisations dans des interfaces simples et intuitives. La métaprogrammation [26, 7, 27] permet de faire tout cela. Son approche consiste à inclure dans le même code source le programme et les instructions pour modifier et/ou aider à compiler ce dernier. La métaprogrammation laisse donc au programmeur expert la possibilité de modifier le langage sans avoir à modifier le compilateur. Elle est une interface entre le programmeur et le compilateur.

Hélas il manque quelque chose de crucial dans cette interface : l'accès au code source ! Ni Scheme [15, 30], ni LISP [8, 31], ni MetaML [32] et ni Template Haskell [27] ne permettent de réifier le code source écrit par le programmeur mis à part celui explicitement passé en paramètre aux macros.

Évidemment fournir une interface pour simplement donner tout le code source au programmeur n'est pas une solution selon nous, car cela revient à demander au programmeur d'écrire son propre compilateur. Nous voulons conserver l'avantage de la métaprogrammation qui est d'étendre le compilateur et non de remplacer le compilateur. Notre contribution est donc de proposer une méthode simple et efficace de fournir le code source : les fonctions transparentes. Nous pensons que notre méthode peut couvrir un grand ensemble de cas où le programmeur a besoin du code source.

1.4 Fonctions transparentes

Cette section vient donner une définition plus formelle de ce que nous entendons par fonctions transparentes. Elle explique brièvement leurs avantages et pourquoi les fonctions sont généralement opaques.

1.4.1 Définition

Pour ce mémoire, par fonction transparente nous impliquons qu'il est possible pour chaque fonction d'obtenir un triplet qui contient le nom des paramètres, le corps de la fonction et l'environnement de fermeture qui contient les valeurs des variables libres. Dans le reste du mémoire, le terme triplet de définition réfèrera à cette définition. Il faut donc que le langage supporte des datatypes capables de représenter les identificateurs (paramètres), le code source (code de la fonction) et un environnement. Notons qu'il n'est pas nécessaire que les fonctions elles-mêmes soient des objets transparents, mais qu'il existe un mécanisme qui pour chaque objet fonction retourne ce triplet de définition.

1.4.2 Pourquoir vouloir des fonctions transparentes ?

Les fonctions transparentes permettent à l'utilisateur et aux primitives du programme de manipuler symboliquement les fonctions. La section Manipulation symbolique en fournira un exemple. La manipulation symbolique des fonctions est souvent associée aux calculs mathématiques sur ordinateurs, mais cette possibilité n'est pas utile uniquement pour les mathématiciens. Elle l'est également pour les ingénieurs ou pour faciliter les calculs scientifiques en général.

Aussi, les applications des fonctions transparentes ne se limitent pas aux calculs scientifiques. Comme mentionné au début de ce chapitre, la logique

des programmes écrits dans un style fonctionnel est incluse dans les fonctions. De façon à pouvoir accéder à cette logique, il faut donc pouvoir ouvrir les fonctions et regarder à l'intérieur. Par exemple analyser et optimiser un parseur écrit sous la forme de combinaison de fonctions. Cet exemple sera repris au chapitre Applications.

1.4.3 Pourquoi favorise-t-on les fonctions opaques ?

Les fonctions transparentes sont absentes de la plupart des langages fonctionnels contemporains. La raison est simple, cela permet au compilateur de faire librement son travail et de pouvoir optimiser plus agressivement le code source.

En effet, il est fort possible qu'une fois le programme compilé l'environnement ne soit pas présent sous la forme d'un datatype [1]. Il est plutôt implicitement présent via la pile et la structure du code généré. Ainsi il est tout simplement impossible de le fournir à moins de le calculer explicitement ce qui engendre un coût supplémentaire à l'exécution.

Aussi, en présence de fonctions transparentes, il est légitime de la part du programmeur de s'attendre à ce que la définition présente dans le triplet de définition soit la même qu'il ait écrite dans le code source. Nous verrons avec le code source 1 que si ce n'est pas le cas cela devient vite très limitant pour faire une analyse du programme.

Or, si la définition est la même, il faut que pour chaque variable libre on puisse trouver sa valeur dans l'environnement de fermeture. Donc le compilateur doit générer ses valeurs. Le compilateur se trouve donc contraint d'exécuter le code tel qu'il est écrit. La moindre modification qui entraine la modification ou l'élimination d'une variable libre devient incompatible avec la sémantique du langage. Ce problème est similaire à celui des Fexprs en LISP

[29, 28]. Bien que les Fexprs soient des objets différents, elles disposaient du code source de leurs paramètres et de l'environnement. Un peu comme nous venons de le faire, Pitman [24] et Wand [36] concluent après avoir analysé les Fexprs que le compilateur ne peut pas effectuer des optimisations en présence de Fexprs. Cette critique a favorisé l'abandon de ces dernières. Nous pensons que cette expérience avec les Fexprs ainsi que le désir de performance ont contribué à exclure les fonctions transparentes des divers langages modernes.

Néanmoins, comme nous le verrons à la section Performance à l'exécution du chapitre Discussion, la transparence des fonctions lors de la métaprogrammation ne rend pas triviale la sémantique du langage comme c'était le cas pour les Fexprs [36]. Ainsi, nous pensons que c'est une erreur de s'en priver lors de la métaprogrammation.

1.5 Métaprogrammation

Cette section vient donner une définition plus formelle de ce que nous entendons par métaprogrammation. Puisqu'il existe une multitude de mécanismes de métaprogrammation, cette définition sera la plus large possible tout en donnant un sens concret sur lequel nous pourrons nous appuyer lors de notre discussion. Nous détaillerons également brièvement certains avantages que procure la métaprogrammation.

1.5.1 Définition

Pour ce mémoire, la métaprogrammation peut être définie comme tout mécanisme prévu par un langage de programmation qui permet au programmeur de générer automatiquement du code qui sera par la suite intégré au programme à compiler. Nous insistons sur le fait que la métaprogrammation

doit s'effectuer lors de la phase de compilation et que ce mécanisme doit faire partie intégrante du langage. Des générateurs de code ad hoc comme Lex [20] ou Yacc [13] ou autres systèmes externes au compilateur tel que les profileurs n'entrent pas dans notre définition. Cette définition est assez large et couvre par exemple les langages LISP, Scheme, Template Haskell et C++.

Nous entendons par macro une méthode, souvent associée à un identificateur, permettant la génération de code. La macro peut recevoir des arguments. Les macros sont bien souvent les blocs de base du mécanisme de métaprogrammation.

1.5.2 Pourquoi vouloir générer du code ?

Économiser du temps

Certaines écritures de codes peuvent être automatisées facilement. La génération de code permet ainsi de sauver du temps et évite des erreurs humaines. Elle libère donc le programmeur d'un travail fastidieux et peu valorisant sur le plan intellectuel. L'implantation automatique de typeclasses en Haskell est un exemple [27]. Le filtrage par motif en Racket est également un autre exemple [7].

Langages dédiés

LISP et Scheme sont deux exemples de langages où l'utilisation de la métaprogrammation pour générer du code est pratique courante. À un point tel où la génération de code peut aussi servir à inclure des langages dédiés au sein du programme [8]. L'utilisation de langages dédiés ouvre de nouvelles possibilités pour le programmeur [37, 4]. Avec un langage dédié, le programmeur peut exprimer plus rapidement des concepts abstraits et le code existant est plus

facile à lire et maintenir. Les experts non informaticiens peuvent également plus facilement comprendre et vérifier la logique du programme.

Toutefois, créer un langage dédié est une tâche complexe. La métaprogrammation peut faciliter cette tâche. Avec elle, le programmeur dispose de toutes les librairies et fonctionnalités du langage de base. Il n'a pas à réécrire un évaluateur à partir de zéro. Le langage dédié peut également être l'extension d'un autre langage dédié plus élémentaire.

Optimisations

Si, en plus de pouvoir générer du code, il est possible d'analyser le code source alors la métaprogrammation peut servir à écrire toutes sortes d'extensions au compilateur. Les optimisations sont un exemple qui vient rapidement en tête. En effet, tel que décrit Robison [26], les optimisations à la compilation sont similaires à la poésie : bien plus sont écrites qu'elles ne sont utilisées dans les compilateurs commerciaux. La raison principale est de nature économique. La plupart des optimisations ont une clientèle très ciblée et trop petite pour justifier leur développement dans des compilateurs commerciaux.

Ainsi, une solution économiquement plus viable est de déléguer ce travail d'optimisation à la communauté de programmeurs réellement affectée par cette dernière. Il faut donc un mécanisme pour que le programmeur puisse, lors de la métaprogrammation, accéder au code source. C'est tout le problème des systèmes actuels de métaprogrammation. Nous le verrons au chapitre Travaux antérieurs, dans bien des cas il n'est pas possible d'accéder au code source. Les fonctions transparentes offrent une solution intéressante à ce problème.

Chapitre 2

Travaux antérieurs

Dans ce chapitre, nous analyserons deux langages qui offrent des possibilités similaires aux fonctions transparentes et deux langages où la métaprogrammation est bien développée. Nous avons choisi Mathematica et Python pour le premier cas et Racket et Template Haskell pour le deuxième. Nous verrons que la transparence des fonctions et la métaprogrammation sont deux idées qui ont été développées en parallèle et que la nouveauté de ce travail ne réside pas dans l'une ou dans l'autre de ces idées, mais bien dans le croisement des deux. Ce chapitre permettra également de comprendre certains choix de conception du langage Abitbol présenté au prochain chapitre, car ce dernier s'inspire de ceux présentés ici.

2.1 Fonctions transparentes

Nous allons voir que Mathematica et Python offrent des fonctions semi-transparentes. Nous les qualifions de semi-transparentes, car au final soit l'environnement est manquant ou soit la définition obtenue d'une fonction

ne correspond pas exactement à celle du code source. Néanmoins, nous verrons que cela ouvre la porte à de nombreuses possibilités. Nous finirons cette section par un résumé des points qui nous semblent les plus importants.

2.1.1 Mathematica

Le langage de Mathematica est un langage symbolique. Les fonctions ne sont qu'un type particulier d'expressions symboliques. Elles n'ont donc pas un statut particulier. Elles peuvent être décomposées comme tout autre objet, d'où leur transparence. Le calcul symbolique est d'ailleurs un des arguments de vente du programme Mathematica [38].

Le calcul symbolique permet de faire de l'analyse et de la génération de code. Il est par exemple possible de calculer symboliquement la dérivée d'une fonction. Mais il est également possible pour le programmeur d'avoir accès à la définition de ses fonctions. Le code source 2.1 à la page 12 fournit quelques exemples où l'on examine la structure d'une fonction qui décrit un polynôme.

Notons toutefois que dans cet exemple le programme Mathematica réécrit les expressions que le programmeur lui fournit. Cela peut être un peu déstabilisateur. En effet, le code source 2.1 fournit un exemple d'une fonction « g » qui fait appel à la fonction « f », mais ce lien est perdu lorsqu'on souhaite obtenir sa définition. Le programmeur pourrait ainsi conclure de façon erronée que modifier « f » n'affecte pas « g ».

Aussi, Mathematica ne donne pas accès à l'environnement tel que nous le proposons à la section 1.4.1. L'environnement ne peut pas être explicitement manipulé. Il est toutefois présent de façon implicite puisqu'il est possible, pour toutes variables, d'avoir son expression symbolique. Le code source 2.1 fournit un exemple où l'on peut retrouver la définition associée à une fonction passée en paramètre.

Pour résumer, le langage de Mathematica n'offre pas les fonctions transparentes comme nous le définissons, mais une alternative suffisante pour faire du calcul symbolique. Couplé avec les nombreuses primitives du langage, cela est amplement suffisant pour le calcul scientifique.

Code source 2.1 : Manipulation symbolique (Mathematica)

```
Simple fonction représentant un polynôme
f[x_] := x * x + 4 * x + 3; f[0]
3

Fonction mise sous sa forme canonique
Notons que cela ne correspond pas à notre définition
(Power[x, 2] au lieu de Times[x, x] pour le terme
 quadratique de notre définition )
FullForm[f[x]]
Plus[3,Times[4,x],Power[x,2]]

Il est possible d'extraire certaine partie
de la définition
{Part[f[x],0],Part[f[x],1]}
{Plus,3}

Même si la fonction est passée comme paramètre,
on peut l'analyser symboliquement
foo[y_,n_]:=Part[y[x],n];{foo[f,0],foo[f,1]}
{Plus,3}

Il est possible de modifier l'expression.
Ici on remplace l'addition des trois termes
par leur multiplication
ReplacePart[f[x],0->Times]
12 x^3

Mathematica manipule des expressions symboliques.
Toutefois le programme transforme ces expressions
à l'interne. Ici, la fonction g sera simplifiée et
la notion que g est composée de f sera perdue.
g[y_] = f[y] + 1;g[0]
4
FullForm[g[x]]
Plus[4,Times[4,x],Power[x,2]]
```

2.1.2 Python

Python [25] possède de nombreuses possibilités s'apparentant aux fonctions transparentes. Nous verrons qu'il possède les éléments nécessaires à la transparence des fonctions mis à part la réification de l'environnement des variables libres.

Un module pour construire un arbre de syntaxe valide et un autre pour inspecter le code source d'un objet font partie de la libraire standard. Il existe une fonction eval() pour évaluer du code et la fonction exec() pour évaluer du code avec effet de bords. Les fonctions eval() et exec() peuvent capter l'environnement courant ou démarrer avec un autre environnement fourni par le programmeur.

Le code source 2.2 et 2.3 montrent des exemples simples de ce qui peut être fait en Python. Le premier illustre bien qu'il est possible d'analyser le code associé à une classe et une méthode. Toutefois, cette analyse est vite limitée, car on ne dispose pas de l'environnement présent au moment de la définition. En effet, toutes les variables libres deviennent des boites noires.

Le deuxième exemple montre qu'avec des primitives appropriées et des objets transparents on peut effectuer des transformations importantes aux définitions du programme. Dans ce cas-ci nous modifions une méthode à la volée. Cela est possible aussi parce que Python permet que les objets soient modifiés dynamiquement.

En résumé, Python fournit le code source des ses objets et méthodes, mais pas l'environnement de fermeture. Bien que limitant pour l'analyse et l'optimisation de code, cela est moins critique en Python. En effet la culture au sein de la communauté Python est que les modifications liées à la vitesse d'exécution ou l'usage de la mémoire passent par la réécriture du module en C (ex : Numpy [34] et Scipy [14]). L'usage de l'introspection de code à des

fins d'optimisation est donc moins crucial.

Code source 2.2 : Inspection du code source d'une classe (Python)

```python
# Python 3.4
import inspect

closure = 5

class foo(object):
    def add(self, a, b):
        return a + b + closure
    def sub(self, a, b):
        return a - b - closure

# Il est possible d'obtenir le code
# source lié à une classe
fooDef = inspect.getsource(foo)

print(fooDef)

# Résultat :
#class foo(object):
#    def add(self, a, b):
#        return a + b + closure
#    def sub(self, a, b):
#        return a - b - closure

# Notons que rien indique la valeur de
# la variable closure ou comment l'obtenir
```

2.1.3 Résumé

Avec Mathematica et Python, nous avons mis en lumière les points suivants concernant les fonctions transparentes :

- La transparence des fonctions (et des classes) est déjà implantée en partie avec succès dans ces deux langages. Python étant un langage fortement populaire et Mathematica un succès commercial de plus de

Code source 2.3 : Manipulation et modification de méthodes (Python)

```python
# Python 3.4
import inspect
import types

class foo(object):
    def add(self, a, b):
        return a + b
    def sub(self, a, b):
        return a - b

x = foo()

# On peut obtenir les méthodes d'un objet
# et les manipuler
methods =
    inspect.getmembers(x,
                       predicate=inspect.ismethod)
addMethod = methods[0][1]
subMethod = methods[1][1]

print(addMethod(1,1))
#Résultat: 2

print(subMethod(1,1))
#Résultat: 0

# On peut également modifier les méthodes
# d'une instance
setattr(x, 'add', types.MethodType(
                       lambda self, a, b:
                                  2 * (a + b), x))

print(x.add(1,1))
#Résultat: 4
```

25 ans, cela démontre que la transparence des fonctions n'est pas un frein au succès d'un langage, du moins si l'environnement n'est pas explicite. Abitbol démontrera qu'on peut aller plus loin et également fournir l'environnement.

- Mathematica fournit déjà un exemple des avantages des fonctions transparentes : le calcul symbolique.
- L'environnement est absolument nécessaire pour bien comprendre la logique du programme. En effet, la sémantique d'une fonction est également déterminée par ses variables libres.
- Si le langage réécrit la définition des fonctions, cela peut masquer la logique du programme. Il devient donc difficile de se fier aux définitions pour analyser ce dernier. Abitbol gardera donc intacte la définition des fonctions.

2.2 Métaprogrammation

Nous nous tournons maintenant vers la métaprogrammation. Nous allons voir que Racket et Haskell offre deux mécanismes de métaprogrammation bien différents, mais qui toutefois offrent des possibilités similaires. Nous finirons cette section par un résumé des points qui nous semblent les plus importants.

2.2.1 Racket

Les langages LISP et Scheme ont une grande tradition de métaprogrammation via leur système de macros. Dans cette section, nous étudierons les

macros telles que définies par le dialecte Racket. Il s'agit probablement d'un des systèmes de macros les plus avancés dans la communauté Scheme.

Dans Racket, la macro syntax-case est une primitive spéciale qui reçoit comme paramètres l'arbre de syntaxe abstraite (ASA) représentant son appel. Cet ASA est représenté par des objets syntaxiques. Le rôle principal des objets syntaxiques est de préserver la portée lexicale des identificateurs. On parle donc de macros hygiéniques [7, 17, 5]. La macro doit retourner par la suite un autre arbre syntaxique qui lui sera substitué dans le code source. Les macros sont expansées à la compilation ou avant l'évaluation du programme. L'avantage de recevoir les paramètres sous forme d'ASA est qu'il est possible pour la macro de recevoir un ASA qui serait invalide et le transformer en un ASA valide. Cela permet l'introduction de nouvelles syntaxes au sein du langage.

Les macros Racket peuvent aussi communiquer entre elles lors de la compilation [7]. En effet, elles disposent d'un environnement commun lors de la compilation. Cela permet par exemple d'écrire des extensions au compilateur tel que le filtrage par motif. Les macros qui définissent un datatype inscrivent sa définition dans l'environnement et les macros qui effectuent le filtrage par motif utilisent cette définition pour générer l'appel des constructeurs appropriés. Le code source 2.4 à la page 19 offre un exemple qui illustre ce mécanisme de partage d'information. Il illustre également l'utilisation d'un environnement séparé pour les macros.

Les macros en Racket souffrent tout de même de quelques limitations. Les objets syntaxiques permettent de différencier les identificateurs, mais ne contiennent pas leur définition. La macro reçoit donc simplement un ASA sans pouvoir en connaitre la sémantique. De plus, les macros de Racket ne sont pas composables comme des fonctions [16, 2]. Elles ne sont pas non plus

des objets de première classe. Elles ne peuvent pas être passées en paramètre ou mise dans des objets. C'est une différence marquée avec les objets fonctions de Racket qui eux sont de première classe.

En plus de ces limitations, la présence des effets de bords et la mutation des variables impliquent qu'il faut que l'environnement d'exécution soit séparé de l'environnement de métaprogrammation. Par exemple, le programmeur pourrait déclarer un tableau global dans son programme. Il est permis aux macros d'instancier ce tableau dans leur environnement et de le modifier. Mais lors de l'exécution, c'est le tableau d'origine non modifié qui sera utilisé. En fait, il faut en théorie une infinité d'environnements puisque les macros peuvent s'imbriquer les unes dans les autres et chacune doit pouvoir instancier le tableau sans interagir de façon non prévue avec les autres. Cette complexité est nécessaire pour éviter des bogues très difficiles à comprendre.

En résumé, les macros de Racket et plus généralement de LISP et Scheme sont un grand succès pour la métaprogrammation. Elles ont démontré que la métaprogrammation permet d'avoir un langage avec un noyau de primitives très petit et d'étendre celui-ci à l'aide des macros. Aussi très important, elles ont démontré que la métaprogrammation peut respecter la portée lexicale si elle est bien implantée. Dans le chapitre Discussion, nous mettrons à profit les macros de Racket pour implanter les fonctions transparentes au sein de ce langage. Nous comparerons cette approche à celle que nous développerons avec Abitbol.

2.2.2 Haskell

Haskell via l'extension Template Haskell [27] offre également un système de métaprogrammation. Les macros, ou templates dans le jargon de Haskell sont expansées lors de la compilation. Comme pour Scheme, leur expansion

Code source 2.4 : Partage d'information entre macros (Racket)

```
#lang Racket

; Exemple de partage d'information par
; les macros via une variable globale
(define-for-syntax nameList (list))

; Cette macro va mémoriser un nom
; dans la liste nameList
(define-syntax saveName
  (lambda (x)
    (syntax-case x ()
      ((_ name)
         (let* ((nSyntax (syntax name))
                (nDatum (syntax->datum nSyntax))
                (nString (symbol->string nDatum)))
           (begin
             (set! nameList (cons nString nameList))
             #`(begin (display #,nString)
                      (newline)))))))))

; Cette macro va lire les noms dans la liste nameList
(define-syntax readName
  (lambda (x)
    (syntax-case x ()
      ((_)
         #`(begin (display '#,nameList) (newline))))))

(saveName hello)
(saveName world)
(readName)

; Résultat :
; hello
; world
; (world hello)
```

remplace leur appel. Puisque Haskell est un langage pur, Template Haskell se base sur le « quotation monad » pour gérer les effets de bords du système de métaprogrammation (gensym, erreur de compilation, etc.). Intégrer un système de métaprogrammation via les monades dans un langage pur est une contribution importante de Template Haskell.

Lors de l'appel d'un template, contrairement aux macros Racket, Template Haskell ne fournit pas les paramètres sous forme d'ASA. Il évalue les paramètres et fournit leurs valeurs. Cela a pour avantage que, contrairement aux macros Racket, les templates utilisés pour la métaprogrammation sont composables [27].

En fait, ce mode d'évaluation permet l'utilisation de n'importe quelle fonction comme un template. La seule contrainte est qu'elle doive retourner un « quotation monad » qui contient le code source à insérer dans le programme. La distinction entre un template et l'appel d'une fonction se fait donc à l'aide de l'opérateur $ comme en MétaML [32]. Le code source 2.5 à la page 21 fournit un petit exemple de métaprogrammation en Haskell.

Également, Haskell étant un langage pur, il n'est pas nécessaire d'avoir une infinité de niveaux d'environnement pour distinguer la mutation des variables par les différentes macros comme pour Racket. Cela simplifie beaucoup la gestion des macros.

Template Haskell souffre également de limitations techniques. Les templates ne peuvent pas être utilisés dans le même module où ils sont définis tels que le montre le code source 2.6 à la page 22. Il faut donc séparer le code source en deux. Un peu plus embêtant pour le programmeur, les paramètres fournis aux templates doivent parfois aussi être définis dans un module différent de celui où a lieu l'appel. Cela est illustré avec le code source 2.7 à la page 23. Également, si un template introduit des déclarations au niveau

global, ces dernières ne sont pas accessibles aux définitions précédant le template. Le mécanisme d'expansion des templates ne respecte donc pas la portée lexicale qui serait en vigueur si le programmeur avait écrit le code à la main tel que le montre l'exemple 2.8 à la page 24.

Notre langage Abitbol va lever plusieurs de ces restrictions.

Code source 2.5 : Métaprogrammation en Haskell

```haskell
-- Fichier Meta.hs
{-# LANGUAGE TemplateHaskell #-}
module Meta where
import Language.Haskell.TH

-- Foo est un template qui va importer
-- la déclaration de la fonction "bar"
foo :: DecsQ
foo = [d| bar = \x -> "Hello " ++ x |]

-- Fichier Run.hs
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH
import Meta

-- Template dont l'expansion sera la déclaration
-- de la fonction bar
$foo

-- Utilisation de bar comme si elle avait été
-- déclarée
main = print $ bar "World"
-- Résultat :
-- Hello World
```

2.2.3 Résumé

Avec Racket et Template Haskell, nous avons mis en lumière les points suivants concernant la métaprogrammation :

Code source 2.6 : Un template ne peut être défini et utilisé dans un même module (Haskell)

```haskell
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH

foo :: DecsQ
foo = [d| bar = \x -> "Hello" ++ x |]

$(foo)

main = print (bar "World")
{--
    GHC stage restriction:
    `foo' is used in a top-level splice or annotation,
    and must be imported, not defined locally
    In the expression: foo
--}
```

- La génération de code est un aspect bien maitrisé actuellement. Notre contribution est donc bien dans la possibilité d'analyser le code source plutôt que dans la génération de code.

- Le problème d'hygiène est un problème difficile, mais résolu. Scheme fut un grand pionnier dans ce domaine.

- Au-delà de la génération de code, un mécanisme de communication entre macros permet d'augmenter significativement les possibilités de métaprogrammation. Cet aspect est complémentaire aux fonctions transparentes. Il a déjà été traité en détail à l'aide de Racket [7].

- La pureté d'un langage simplifie de beaucoup les choses. Il n'y a pas besoin d'une infinité de niveaux d'environnement méta pour distinguer les effets de bords des différentes macros.

- Il existe au moins deux façons de passer les paramètres à une macro. La

Code source 2.7 : Impossibilité d'utiliser une variable globale comme argument à un template global (Haskell)

```haskell
-- Fichier Meta.hs
{-# LANGUAGE TemplateHaskell #-}
module Meta where
import Language.Haskell.TH

foo :: String -> DecsQ
foo h = [d| bar = \x -> h ++ x |]

-- Fichier Run.hs
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH
import Meta

hello :: String
hello = "Hello"

world :: String
world = "World"

$(foo hello)

main = print (bar world)
-- Résultat :
-- GHC stage restriction:
-- `hello' is used in a top-level splice or annotation,
-- and must be imported, not defined locally
-- In the first argument of `foo', namely `hello'
-- In the expression: foo hello
```

Code source 2.8 : Erreur de portée lexicale (Haskell)

```haskell
-- Fichier Meta.hs
{-# LANGUAGE TemplateHaskell #-}
module Meta where
  import Language.Haskell.TH

  foo :: DecsQ
  foo = [d| bar = \x -> "Hello " ++ x |]

-- Fichier Run.hs
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH
import Meta

world :: String
world = "World"

-- En amont et le code compile
-- $foo

main' = print $ bar world

-- En aval et le code ne compile plus
$foo

main = main'
-- Résultat :
-- Not in scope: `bar'
```

première est de faire comme Racket et de passer l'ASA simplement. La deuxième est de faire comme Template Haskell et de passer la valeur des paramètres. La première façon facilite l'introduction de nouvelle syntaxe et de nouveaux opérateurs. La deuxième permet aux macros d'être composables.

- L'expérience de Template Haskell montre qu'il est préférable qu'un mécanisme de métaprogrammation soit prévu en amont pour bien l'intégrer au langage. Sinon plusieurs contraintes architecturales peuvent limiter son ergonomie.

2.3 Résumé du chapitre

Nous avons vu dans ce chapitre quatre langages qui démontrent qu'à la fois les fonctions transparentes et l'usage de la métaprogrammation sont actuellement des idées déjà implantées. Par exemple, Mathematica et Python permettent d'avoir accès au code source d'une fonction et cela permet notamment le calcul symbolique. Il reste néanmoins que l'environnement de fermeture des fonctions n'est pas fourni explicitement et Abitbol corrigera ce problème. Pour ce qui est des systèmes de macros, Racket et Template Haskell fournissent deux solides bases sur lesquelles nous pouvons baser notre travail. Notons que ces deux systèmes offrent des possibilités différentes telles que par exemple le passage des arguments aux macros par valeur (Template Haskell) ou par ASA (Racket).

Puisqu'à la fois les fonctions transparentes et l'usage de la métaprogrammation sont actuellement des idées déjà implantées, notre travail se démarque en réunissant les deux de façon harmonieuse. Le chapitre 3 décrira en détail comment nous y sommes arrivés.

Chapitre 3

Abitbol

Abitbol est un langage créé pour implanter la thèse principale de ce mémoire soit que les fonctions devraient être transparentes lors de la phase de métaprogrammation. Abitbol va nous permettre d'en comprendre l'impact sur les autres propriétés des langages fonctionnels. Il nous permet également de fournir divers exemples d'utilisation des fonctions transparentes lors de la phase de métaprogrammation.

Mis à part le système de type, Abitbol s'inspire beaucoup de Haskell [23] et possède pratiquement la même syntaxe. Les effets de bords y sont représentés par des monades comme pour Haskell [35]. Par contre Abitbol diffère de Template Haskell. Son mécanisme de macros propose des fonctionnalités différentes.

Nous allons débuter par une présentation des concepts de base du langage Abitbol. Nous allons justifier nos choix de conception du langage en fonction de notre objectif principal, la présence des fonctions transparentes lors de la phase de métaprogrammation. Nous allons ensuite décrire la syntaxe du langage et la structure d'un programme en Abitbol. Nous décrirons par la suite

le mécanisme des fonctions transparentes et celui de la métaprogrammation. Nous finirons par une section sur l'implantation du compilateur Abitbol.

3.1 Concept de base

Abitbol est un langage typé dynamiquement, à portée lexicale, pur et paresseux.

3.1.1 Typage dynamique et objets

Abitbol est typé dynamiquement. Les objets sont de simples conteneurs qui peuvent être différenciés par leur type. Les types sont déclarés à l'aide du mot clef « data » suivi du nom du type et du nom de chaque accesseur. Le nombre d'accesseurs présent dans la déclaration indique le nombre d'objets que contient un objet de ce type. Des types singletons qui ne contiennent aucun objet sont possibles. Abitbol ne possède pas type union comme dans Haskell. Ainsi, au lieu d'avoir un type booléen qui contient soit un objet True ou un objet False, il y a simplement un type singleton True et un type singleton False. Donc une fonction qui retourne un booléen retourne en fait soit un objet de type True ou un objet de type False. Le typage dynamique nous permet plus de granularité dans la définition des types que le typage statique de Haskell. Le code source 3.1 contient des exemples de déclarations.

Notre implantation d'Abitbol fournit plusieurs types soit comme primitives (Tuples, Int, Char, etc.) ou dans la libraire standard (liste, types représentant la syntaxe, etc.). Les objets peuvent être comparés à l'aide de la primitive « == » qui effectue une égalité structurelle simple.

Code source 3.1 : Exemples de déclarations de datatypes (Abitbol)

```
1  -- Exmples de déclaration de types
2
3  -- Singleton
4  data True;
5  data False;
6
7  -- Il n'y a pas de type algébrique
8  -- Voici l'équivalent du datatype Maybe en Haskell
9  data Nothing;
10 data Just fromMaybe;
```

Filtrage par motif

L'ensemble des objets créés par l'utilisateur ou fournis par la librairie standard peut être déconstruit à l'aide de motif. Tout comme Haskell, il est possible d'utiliser ces motifs dans la définition des objets, des fonctions ou avec l'expression case. Du sucre syntaxique pour les motifs des listes et tuples existe également.

3.1.2 Portée lexicale

Abitbol adhère aux règles de la portée lexicale. D'abord parce qu'il s'agit d'un choix naturel en 2015. Ensuite, il faut mentionner qu'avec les fonctions transparentes la portée dynamique est pratiquement incompatible. En effet, l'objectif d'avoir des fonctions transparentes est de pouvoir analyser le code source. Or si les valeurs des variables libres sont déterminées en fonction de la portée dynamique, il n'est pas possible en général d'en savoir la définition avant l'exécution du programme. Puisque notre objectif est de favoriser la métaprogrammation, qui a lieu avant l'exécution du programme, nous avons naturellement choisi la portée statique.

3.1.3 Pureté

Nous avons choisi de faire d'Abitbol un langage pur. Nous entendons par pur que les fonctions sont pures et nous rappelons comme mentionné en introduction qu'Abitbol se concentre uniquement sur le paradigme de programmation fonctionnelle. Il s'agit donc d'un langage fonctionnel pur. La pureté simplifie de beaucoup l'analyse des fonctions transparentes et garantit aussi que la sémantique d'une fonction soit correctement définie par le triplet de définition de la section Fonctions transparentes du chapitre Introduction.

Néanmoins, la présence des fonctions transparentes lors de la phase de métaprogrammation ne nécessite pas que le langage soit pur. Nous pensons que tout langage permettant d'identifier les variables mutables et les constantes peut très bien fonctionner avec les fonctions transparentes. Il existe plusieurs mécanismes pour cela comme l'usage du mot clef « let » pour déclarer des constantes et « var » pour déclarer des variables mutables en Swift. Ou les « reference cell » en F#. Puisque la mutabilité des variables amène quelques problèmes que nous détaillerons sous peu, il est préférable que la déclaration des identificateurs soit par défaut immuable plutôt que mutable. Ainsi, sans annotation explicite du programmeur, il est possible de conclure que la variable représente une constante.

Il existe plusieurs raisons qui avantagent les variables immuables en présence des fonctions transparentes. Prenons l'exemple d'une fonction avec variables libres. Pour en avoir une définition juste, il faut non seulement pouvoir analyser ses variables libres, mais également toutes les expressions pouvant modifier ces variables. Nous allons donner un exemple de ce qui peut mal tourner avec le code source 3.2 et 3.3 à la page 31. Ces exemples illustrent une fonction qui fait appel à une variable globale nommée « tricky ». Lorsque l'objet fonction « foo » est créé cette variable vaut 5. Dans le code source 3.2

cette variable globale est modifiée avant un appel de la fonction pour contenir la valeur 4. La bonne définition, si on s'intéresse à cet appel, est que la fonction contient une variable libre qui vaut 4. Dans le code source 3.3 la variable globale « tricky » est modifiée après l'appel de la fonction. Ici, si on s'intéresse toujours à l'appel de la fonction, la bonne définition est que la fonction contient une variable libre qui vaut 5. Ainsi dans un langage impur les fonctions peuvent changer de définition en cour d'exécution. Mais savoir quand exactement arrive la modification peut nécessiter une analyse complexe du programme et même être impossible à déterminer à la compilation.

Cet exemple a également illustré que certaines parties du code qui ne sont pas présentes dans la définition de la fonction ni via son environnement de fermeture peuvent modifier la définition. Or le triplet de définition ne donne pas accès à ces expressions. Il est donc impossible avec notre approche d'obtenir ce code. Ainsi, au-delà du problème de définition changeante, il est possible que notre mécanisme ne puisse pas fournir le code qui effectue le changement.

Il existe également un autre problème des langages impurs. Si l'objet associé à une variable libre est créé à l'aide d'effet de bords, alors ceux-ci peuvent être différents à la compilation et à l'exécution. Il faut donc que le programmeur soit très attentif aux effets de bords. Le code source 3.4 à la page 32 illustre ce problème.

Au final, la seule façon de garantir que la définition contenue dans le triplet de définition soit juste est d'interdire sa modification. C'est la garantie que procure un langage pur. Mais puisque nous prônons l'usage des fonctions transparentes lors de la phase de métaprogrammation et que c'est le programmeur qui a le contrôle sur cette phase, il est raisonnable de le laisser décider s'il veut faire l'analyse d'une fonction avec variables mutables ou non.

Nous pensons donc que notre approche va bien avec tout langage qui permet de faire la distinction entre une constante et une variable mutable.

Code source 3.2 : Modification de variables libres 1 (Racket)

```
#lang racket
; Pour avoir la bonne définition à l'exécution
; de foo il faut analyser bar et réaliser
; que bar est appelé avant l'usage de foo
(define tricky 5)

(define (foo x) (+ x tricky))

(define (bar)
  (begin (set! tricky 4)
         (print "haha")
         (newline)))

(bar)
(print (foo 5))
;Résultat
;"haha"
;9
```

3.1.4 Évaluation paresseuse

Il est maintenant établi qu'Abitbol sera un langage pur. Haskell, en plus de la pureté, possède aussi un mode d'évaluation paresseuse. Cela n'a toutefois pas été décidé en fonction de critères liés à la métaprogrammation et aux fonctions transparentes [11]. Il importe donc d'analyser lequel des deux mécanismes, strict (call-by-value) ou paresseux (call-by-need), serait le plus approprié pour Abitbol.

Dans un langage pur et pour une même expression, l'évaluation stricte ou l'évaluation paresseuse arrivent au même résultat à l'exception d'un cas de figure : lorsque l'évaluation d'une expression ne termine pas. Le code source 3.5

Code source 3.3 : Modification de variables libres 2 (Racket)

```
#lang racket
; Pour avoir la bonne définition de foo
; à l'exécution, il faut réaliser
; que bar est appelé après
; l'usage de foo
(define tricky 5)

(define (foo x) (+ x tricky))

(define (bar)
  (begin (set! tricky 4)
         (print "haha")
         (newline)))

(print (foo 5))
(bar)

;Résultat
;10"haha"
```

Code source 3.4 : Effets de bords modifiant la définition d'une fonction (Racket)

```
#lang racket
; Bien qu'il soit possible d'analyser foo
; à la compilation, il serait
; une erreur de se fier à sa valeur.


; (current-seconds) retourne le nombre de
; secondes écoulés depuis le 1er janvier 1970
(define s (current-seconds))

(define (foo x)
  (if (== (remainder s x) 0)
      1
      0))
```

montre un exemple. La fonction « foo » reçoit un paramètre mais ne fait rien avec et retourne une constante. On pourrait être tenté de conclure que la fonction retourne toujours. Or si son appel est fait avec une expression qui ne termine pas comme argument, cet appel ne terminera jamais également.

Ainsi, dans le cas d'une évaluation paresseuse il n'est pas nécessaire de se soucier de la définition d'une variable libre ou d'un paramètre si ceux-ci ne contribuent pas au résultat. Ceci est un avantage pour la métaprogrammation, car il n'est pas nécessaire de valider qu'une expression termine ou non avant de l'éliminer ou la déplacer dans le code source. Puisqu'il est en général difficile voir impossible de prouver cela, la métaprogrammation dans une évaluation paresseuse requiert moins d'analyse à ce point de vue.

Toutefois, il faut tout de même mentionner que l'évaluation paresseuse n'a pas que des avantages. Par exemple la complexité mémoire d'un programme est très difficile à évaluer avec l'évaluation paresseuse. Un programmeur qui utilise la métaprogrammation pour améliorer les performances de son programme pourrait préférer une évaluation stricte, car son analyse serait de beaucoup simplifiée.

Le mode d'évaluation n'est pas aussi facile à trancher que pour la pureté. Au final, nous avons choisi de doter Abitbol d'une évaluation paresseuse plutôt par préférence et par familiarité avec Haskell. Aux fins de la thèse principale qui est la présence de fonctions transparentes lors de la phase de métaprogrammation, nous sommes d'avis que les deux types d'évaluation se valent.

Code source 3.5 : Évaluation paresseuse et analyse de code (Haskell)

```haskell
-- Fonction qui ne termine jamais
sansFin :: a -> Int
sansFin = \_ -> 1 + sansFin 0

-- Une analyse naïve de foo conclut qu'un
-- appel à cette fonction termine toujours
-- Mais cela est vrai uniquement avec une
-- évaluation paresseuse
foo :: Int -> String
foo x = "Hello World"

main = putStrLn (foo (sansFin 0))
-- Résultat:
-- "Hello World"
```

3.2 Structure du programme

Dans cette section et la prochaine, nous allons utiliser une syntaxe similaire à la forme de Backus-Naur (BNF) pour décrire la syntaxe du langage Abitbol. Nous allons également employer les notations suivantes :

Grammaire 3.1 Notations utilisées pour les grammaires

$\{patron\}$	$\rightarrow$	Zéro ou plusieurs répétitions
$pat_1 \mid pat_2$	$\rightarrow$	Alternatives
let	$\rightarrow$	Mot clef du langage
explication	$\rightarrow$	Explication résumant la règle

Comme pour Scheme R5RS [15], Abitbol ne possède pas de systèmes de modules. Seule une librairie standard est définie et automatiquement toutes ses déclarations sont à la portée du niveau global du programme. Un programme est constitué de déclarations. Il est possible de déclarer des fonctions, des constantes ou des datatypes. Ces déclarations sont mutuellement récursives. Le programme doit définir une fonction « main » qui sera le point

d'entrée du programme.

Grammaire 3.2 Grammaire d'un programme d'Abitbol

programme	→	*declaration* {*declaration*}

3.3 Syntaxe

La syntaxe d'Abitbol s'inspire beaucoup de celle de Haskell. Elle provient pratiquement du rapport Haskell 2010 [23] sauf pour la déclaration des datatypes car le typage est différent. Afin que le lecteur puisse suivre les exemples, nous allons détailler cette syntaxe. Nous omettrons certains détails qui ne sont pas requis pour comprendre les exemples de façon à alléger la lecture.

3.3.1 Déclarations

Les déclarations permettent d'introduire de nouveaux datatypes et de définir des constantes et des fonctions. La grammaire 3.3 en donne la définition.

Grammaire 3.3 Grammaire des déclarations d'Abitbol

declaration	→	*datatype* | *constante* | *fonction*
datatype	→	**data** *constructeur* {*accesseur*};
constante	→	*motif* = *expression*;
fonction	→	*clause* {*clause*}
clause	→	*variable motif* {*motif*} = *expression*;
motif	→	Motif utilisé dans le filtrage par motif.

Le code source 3.1 à la page 28 montre déjà des exemples de déclarations de datatype. Le code source 3.6 montre des exemples de déclarations de constantes. On reconnait la définition des constantes par le fait qu'il n'y a qu'un seul motif. Ce motif peut toutefois contenir plusieurs variables ce

qui introduit plusieurs constantes. L'expression à droite du signe « = » est évaluée et la correspondance avec le motif sera effectuée de façon paresseuse. Si la correspondance échoue, il s'agit d'une erreur.

On reconnait la définition d'une fonction par la présence d'une variable suivie d'un ou plusieurs motifs. Chaque motif permet d'effectuer du filtrage par motif sur les paramètres. Il est possible de définir une fonction avec plusieurs clauses pour faciliter l'écriture du filtrage par motif. Les clauses sont essayées dans l'ordre de la première à la dernière. Si aucune clause ne correspond aux objets passés en paramètres, il s'agit d'une erreur.

Code source 3.6 : Déclarations de fonctions et constantes (Abitbol)

```
-- Exemples de motifs
a; -- Variable
True; -- Objet True, aucun accesseur
Foo x; -- Objet Foo, 1 accesseur
Bar x (Foo y); -- Objet Bar, 2 accesseurs
(x:xs); -- Liste

-- Déclaration de constantes
x = 2;
y = Foo x;

-- Déclaration de fonctions
foo x = 1;

bar (Just x) = x;
bar Nothing = 0;

baz a b 0 = a + b;
baz a b c = a + b + c;
```

3.3.2 Expressions

Les expressions sont la base de la syntaxe du langage Abitbol. Les expressions, une fois évaluées, retournent des objets. Nous allons décrire la syntaxe

des diverses expressions et donner un exemple de chacune au code source 3.7 à la page 38. La grammaire 3.4 donne une définition formelle des expressions.

Grammaire 3.4 Grammaire des expressions d'Abitbol

let	→	**let**{{*declaration*}} **in** *expression*
if	→	**if** *expression* **then** *expression* **else** *expression*
lambda	→	*motif* {*motif*} → *expression*
case	→	**case** *expression* **of** {{*alternative*}}
alternative	→	*motif* → *expression*;
application	→	*operateur operande* {*operande*}
appInfixe	→	*operande opInfixe operande*

Comme pour Haskell, l'expression let permet d'introduire zéro ou plusieurs déclarations locales. Lorsque plusieurs déclarations sont présentes, celles-ci sont mutuellement récursives.

L'expression if représente un branchement conditionnel comme dans Haskell. La branche else doit obligatoirement être présente.

L'expression lambda permet de créer une fonction anonyme. Nous verrons à la section Fonctions transparentes du présent chapitre comment obtenir le triplet de définition.

L'expression case permet l'application du filtrage par motif à un objet. Les alternatives sont suivies dans l'ordre de la première à la dernière. Dès qu'un objet correspond à un motif, l'expression associée à ce motif est évaluée et cela est la valeur de retour de l'expression case. Si aucun motif ne correspond à l'objet, il s'agit d'une erreur.

Finalement, l'application de fonction suit la même syntaxe que pour Haskell. L'expression de tête est l'opérateur et les expressions suivantes les arguments. Abitbol supporte les applications partielles (currying). Pour éviter une syntaxe ambigüe, il faut mettre des parenthèses autour des expressions let, if, case et lambda si l'on désire les utiliser directement comme argument

ou opérateur. Les applications infixes sont également supportées.

Code source 3.7 : Exemple d'expressions (Abitbol)

```
-- Let
let { y = x + 1;
      x = 0;
      foo 3 = 3 + 2;
      foo x = x + 2 + 1;}
in y + x + foo 8;

-- If
if x > 0
then x
else -x;

-- Lambda
\ x y = x + y;
\ (Just x) (Just y) = Just (x + y);

-- Case
case x of {
  Just x -> x;
  Nothing -> 0};

-- Application
foo 1 2;

bar (let {x = 4;} in x)
    (if True then 0 else 5);

map (\x -> x) [1, 2, 3];

x = 1 + 2 * 3;
```

3.4 Fonctions transparentes

Dans Abitbol, les fonctions définies par l'expression lambda ou par une déclaration de fonction sont des objets de type fonction. Chaque objet fonction

doit pouvoir être décomposé dans le datatype Lambda qui symbolise le triplet de définition. Sa définition formelle est donnée au code source 3.8 à la page 39. La primitive fdefinition permet de prendre n'importe quel objet fonction et le transformer en objet de type Lambda.

Ainsi les objets fonction n'ont pas en soit à être transparents. Cela permet aux implantations de représenter à l'interne les fonctions à leur guise. L'important est de pouvoir faire le pont entre les deux types d'objets. Dans notre implantation les fonctions sont représentées par un triplet également. Il est donc très facile de faire le pont entre les deux datatypes.

De façon à représenter le corps de la fonction, Abitbol expose dans sa librairie standard les datatypes nécessaires pour encoder sa syntaxe. Comme pour Template Haskell, ces datatypes donnent plus d'information que ne le ferait un ASA standard. On y retrouve par exemple un datatype pour l'utilisation des parenthèses. Ceci est pour refléter fidèlement le code tel qu'il est écrit. Le code source 3.8 donne un exemple pour les expressions.

Code source 3.8 : Datatype pour les expressions (Abitbol)

```
1  data Lambda params expr env -- Triplet de définition
2
3  data VarE name; -- Variable
4  data ConE name; -- Constructeur
5  data LitE value; -- Constantes
6  data AppE expr1 expr2; -- Application
7  data InfixE expr1 op expr2; -- Application infixe
8  data NegE expr; -- Négation
9  data ParensE expr; -- Parenthèse
10 data LamE patterns expr; -- Lambda
11 data IfE expr1 expr2 expr3; --If
12 data LetE decls expr; --Let
13 data ListE exprs; --Sucre pour les listes
14 data TupE exprs; --Sucre pour les tuples
15 data CaseE expr matches; --Case
```

3.5 Métaprogrammation

Abitbol a été doté d'un mécanisme de métaprogrammation qui répond bien aux objectifs de ce mémoire. Comme Template Haskell, il faut indiquer par le symbole $ qu'une expression doit être calculée lors de la phase de métaprogrammation. Dans la suite du texte, le terme macro réfère aux expressions qui sont précédées par le symbole $. L'usage des macros n'est permis que là où sont valides les expressions. Il n'est donc pas possible de générer des déclarations au niveau global ou dans une expression let. Template Haskell a déjà démontré cette possibilité et ses avantages. Nous reviendrons sur cet aspect au chapitre Discussion.

Le résultat de l'expansion d'une macro est la valeur produite par l'expression expansée. Le compilateur reçoit donc un objet. Cela est similaire au mécanisme de la primitive « quote » en Scheme [15, 30]. Vue d'une autre façon, après la phase de métaprogrammation le programme est représenté par un ASA dont certains nœuds ne sont pas des expressions mais des objets. Ces derniers proviennent de la métaprogrammation.

Ce mécanisme est simple à implanter et favorise beaucoup la manipulation de fonctions. En effet, en Template Haskell, si l'on veut retourner un objet il faut traduire cet objet en une expression puisque la méta programmation est une transformation source à source. Or les objets fonctions sont difficiles à traduire. Leur définition fait surement référence à plusieurs identificateurs qui, lorsque retranscrits dans une autre section du code, peuvent avoir une définition différente. C'est le problème d'hygiène [17, 5, 3, 6]. Nous reviendrons également sur cet aspect au chapitre Discussion. Le mécanisme d'Abitbol permet de retourner une fonction et de contourner le problème d'hygiène. L'avantage de contourner le problème d'hygiène est que les va-

riables peuvent être représentées simplement par une chaîne de caractères. Il est donc plus facile de manipuler le corps des fonctions lorsqu'on les examine à l'aide du triplet de définition.

Nous allons maintenant préciser la mécanique de la phase d'expansion de macros. Pour simplifier la présentation et sans perte de généralité, nous présenterons seulement l'algorithme d'expansion pour les expressions. Les déclarations globales du programme ont la même sémantique qu'une expression let qui rassemble toutes ces déclarations globales et qui appelle la fonction main. Nous supposerons également qu'il n'y a que des constantes de déclarées. En effet, la déclaration de fonction a la même sémantique qu'une déclaration de constante définie par une expression lambda. Le code source 3.9 fournit des exemples de ces équivalences.

Code source 3.9 : Transformation avant la phase d'expansion de macros

```
1  -- Déclarations initiales
2  foo 0 0 = 0;
3  foo x y = x + y;
4  bar = 4;
5  main = foo bar;
6
7  -- Déclarations pour l'expansion de macros
8  let {
9    foo = \x y ->
10     case (x, y) of {
11       (0, 0) -> 0;
12       (x, y) -> x + y;};
13   bar = 4;
14   main = foo bar;}
15 in main
```

L'algorithme d'expansion de macros figure à la page 43. L'algorithme reçoit un environnement et une expression en entrée et retourne cette expression expansée ou une erreur. Nous symboliserons l'expansion de cette

expression e dans cet environnement env par $[\![e]\!]_{env}$. L'environnement sert à garder en mémoire les valeurs des différentes variables qui sont à portée. En effet, comme pour Template Haskell, lors d'un appel de macro il faut passer à la macro les valeurs des paramètres et non leur ASA. Il faut donc avoir gardé en mémoire ces valeurs.

L'algorithme 3.1 montre que la macro expansion est assez simple pour les variables, l'expression if et l'application de fonction. Il s'agit simplement de faire l'expansion des sous-expressions s'il y a lieu.

L'expansion d'une expression lambda est également assez simple. Il faut faire l'expansion du corps de la fonction avec l'environnement courant dans lequel nous avons ajouté les paramètres de la fonction. Puisque la valeur des paramètres n'est pas connue à la compilation, nous leur associons une erreur. Ainsi, si lors d'une macro le programmeur fait appel à la valeur d'un des paramètres l'algorithme d'expansion de macro s'arrêtera et indiquera à l'utilisateur que cette valeur n'est pas disponible à la compilation.

Pour une expression case, le sujet du case est expansé dans l'environnement courant. Pour chacune des alternatives, il faut tenir compte des variables qui seront introduites par le motif de l'alternative. Il faut donc ajouter ces variables à l'environnement avant d'expanser l'expression associée au motif. Si le sujet du case est connu à la compilation, alors il est possible d'effectuer le filtrage par motif et d'associer une valeur à chaque variable du motif. Si le sujet du case n'est pas connu, alors on associe une erreur aux variables du motif. On associe également une erreur si le sujet du case est connu mais que le filtrage échoue. Il faut bien sûr comprendre que cela s'effectue de façon paresseuse. Donc l'évaluation du sujet du case afin d'obtenir le filtrage par motif et celui-ci seront effectués seulement si la valeur de la variable est réellement demandée par une macro.

Algorithme 3.1 Algorithme d'Abitbol pour la phase d'expansion de macros

$$
\begin{aligned}
[\![x]\!]_{env} &\rightsquigarrow x \\
[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]_{env} &\rightsquigarrow \textbf{if } [\![e_1]\!]_{env} \textbf{ then } [\![e_2]\!]_{env} \textbf{ else } [\![e_3]\!]_{env} \\
[\![op\ arg_1 \ldots arg_n]\!]_{env} &\rightsquigarrow [\![op]\!]_{env}\ [\![arg_1]\!]_{env} \ldots [\![arg_n]\!]_{env} \\
[\![\backslash motif_1 \ldots motif_n \rightarrow e]\!]_{env} &\rightsquigarrow \backslash motif_1 \ldots motif_n \rightarrow [\![e]\!]_{env_{app}} \\
\left[\!\!\left[\begin{array}{l}\textbf{case } e \textbf{ of}\{ \\ \quad motif_1 \rightarrow e_1; \\ \quad \ldots \\ \quad motif_n \rightarrow e_n;\}\end{array}\right]\!\!\right]_{env} &\rightsquigarrow \begin{array}{l}\textbf{case } [\![e]\!]_{env} \textbf{ of}\{ \\ \quad motif_1 \rightarrow [\![e_1]\!]_{env_{case_1}}; \\ \quad \ldots \\ \quad motif_n \rightarrow [\![e_n]\!]_{env_{case_n}};\}\end{array} \\
\left[\!\!\left[\begin{array}{l}\textbf{let}\{c_1 = e_1\ ; \\ \quad \ldots \\ \quad c_n = e_n\ ;\} \\ \quad \textbf{in } e\end{array}\right]\!\!\right]_{env} &\rightsquigarrow \begin{array}{l}\textbf{let}\{c_1 = [\![e_1]\!]_{env_{let}}\ ; \\ \quad \ldots \\ \quad c_n = [\![e_n]\!]_{env_{let}}\ ;\} \\ \quad \textbf{in } [\![e]\!]_{env_{let}}\end{array} \\
[\![\$e]\!]_{env} &\rightsquigarrow box\ (eval_{env}\ [\![e]\!]_{env})
\end{aligned}
$$

où

env_{app} est l'environnement augmenté des variables présentes dans les motifs des paramètres. Ces variables sont toujours associées à une erreur car leur valeur n'est pas connue à la compilation.

env_{case_i} est l'environnement augmenté des variables présentes dans le i$^{\text{ème}}$ motif. Ces variables sont associées à une erreur si leur valeur n'est pas connue à la compilation ou celle-ci est calculée façon paresseuse.

env_{let} est l'environnement augmenté des variables présentes dans les déclarations. Ces variables sont associées à une erreur si leur valeur n'est pas connue à la compilation ou celle-ci est calculée façon paresseuse.

box signifie que l'objet obtenu remplace l'appel de la macro dans l'ASA.

$eval_{env}$ signifie évaluer l'expression dans l'environnement *env*

Pour l'expression let, toutes les nouvelles variables provenant des déclarations sont ajoutées à l'environnement. Leur valeur correspond à l'évaluation paresseuse dans cet environnement augmenté de leur expression préalablement expansée également dans cet environnement. Il s'agit ici de la même logique que pour l'évaluation classique d'une expression let dans un langage paresseux. Notons qu'il est possible de se marcher sur les pieds et d'écrire une boucle infinie avec deux définitions qui s'appellent mutuellement lors de l'expansion de macros.

Finalement, lors d'un appel de macro il faut d'abord expanser l'expression et ensuite l'évaluer. Le résultat de l'évaluation est inséré dans l'ASA du programme.

Le code source 3.10 montre des exemples de métaprogrammation. Dans ces exemples, celui de la fonction Fibonacci illustre un aspect subtil de l'algorithme. Cette fonction fait un appel de macro avec elle-même. Cela est possible car le code exécuté pendant l'évaluation de ces appels de macro ne comprend pas d'autres appels de macros qui forment une boucle. Si l'on avait exigé que l'expansion d'une fonction soit faite au complet avant de pouvoir l'utiliser dans la phase de métaprogrammation cette définition n'aurait pas fonctionné. À remarquer que Scheme permet également des appels de macros récursifs.

3.5.1 Paresse et évaluation lors de l'expansion de macros

Avant de passer au chapitre suivant, nous voulons revenir sur un aspect fondamental de l'algorithme 3.1. Puisqu'il faut passer les valeurs des paramètres aux macros, il faut que pour chaque variable dans l'environnement on puisse

Code source 3.10 : Exemples d'expansion de macros (Abitbol)

```
-- Erreur car la valeur de la variable x
-- n'est pas connue à la compilation
bar x = 3 + $(fib x);

-- OK
baz = case (Just 4) of {
            Just x -> $x;
            Nothing -> 0;
          };

-- Fibonacci
fib 0 = 1;
fib 1 = 1;
fib 2 = $(fib 1 + fib 0);
fib 3 = $(fib 2 + fib 1);
fib n = fib (n - 1) + fib (n - 2);

-- Let
let {
  x = 4;
  foo y = y + $(x + 1);
  bar = $foo;
  }
in bar x;

-- Let un peu tordu
let {
  bar 0 = 1;
  bar n = $(foo 1);

  foo 1 = 0;
  foo n = $(bar 0);
  }
in bar 5;
```

potentiellement obtenir sa valeur ou une erreur si celle-ci n'est pas connue à la compilation. Ce sont les expressions let et case qui introduisent de nouvelles variables. Ainsi pour chaque nouvelle variable dans ces expressions l'algorithme calcule de façon paresseuse sa valeur. En calculant la valeur de chaque variable de façon paresseuse, cela revient en fait à calculer seulement les valeurs réellement demandées par les macros.

Si Abitbol avait un mode d'évaluation stricte, il serait possiblement coûteux d'utiliser cette stratégie. En effet cela reviendrait à calculer la valeur de toutes les variables présentes dans les expressions let et case. Mais il est fort probable que seulement une faible proportion de ces valeurs seront nécessaires pour la métaprogrammation. C'est donc un autre avantage de la paresse en lien avec les fonctions transparentes. Elles permettent de calculer à la demande les valeurs nécessaires lors de la métaprogrammation et seulement celles-ci.

Nous verrons au chapitre Discussion une méthode alternative à l'aide des macros de Racket pour obtenir des fonctions transparentes à la métaprogrammation dans un langage strict.

3.6 Implantation

Nous allons fournir dans cette section quelques détails sur l'implantation de notre évaluateur pour le langage Abitbol. Nous allons également décrire l'architecture globale ainsi que quelques problèmes rencontrés.

L'évaluateur a été écrit à l'aide du langage Haskell. Le choix du langage est principalement dû à ses propriétés communes avec Abitbol, notamment la pureté, la paresse et le passage par valeur des arguments aux macros (Template Haskell). L'analyse lexicale et syntaxique a été écrite à l'aide de la

librairie Parsec [18, 19]. Le Haskell Report 2010 [23] fournit au chapitre 10 des pseudo algorithmes pour les analyses lexicales, syntaxiques et la résolution de la priorité des opérateurs du langage Haskell. Nous nous sommes inspirés de ceux-ci pour notre évaluateur.

3.6.1 Ordre d'exécution

L'ordre d'exécution de l'évaluateur est le suivant :

Analyse lexicale Lecture du fichier et création des jetons lexicaux. Abitbol ayant une syntaxe quasi identique à celle de Haskell, l'algorithme utilisé peut être trouvé dans le Haskell Report 2010.

Analyse syntaxique Formation de l'ASA. À ce stade la priorité des opérations n'a pas encore été résolue. Les expressions infixes sont donc contenues dans une branche de l'ASA spécialement marquée comme étant non résolue. L'algorithme pour la formation de l'ASA est également similaire à celui du Haskell Report 2010.

Priorité des opérateurs Cette passe spécialisée dans la priorité des opérateurs permet de convertir les branches de l'ASA marquées comme non résolues en branche avec opérateurs binaires dont la structure respecte la priorité des opérations. L'algorithme s'inspire également du Haskell Report 2010.

Expansion de macro Expansion des macros comme décrite à la section précédente. Bien que Haskell soit un langage paresseux, notre évaluateur s'assure en utilisant les monades que l'expansion des macros soit terminée avant de passer à la phase suivante.

Évaluation Évaluation paresseuse du code expansé. L'évaluation se fait sans optimisation et les fonctions sont opaques. La sémantique opérationnelle est similaire à celle de Haskell une fois les macros expansées.

Au total l'évaluateur dans son ensemble comprend environ 5000 lignes de code Haskell. Le résultat de l'évaluation est soit une valeur ou une erreur. L'évaluateur émet des messages d'erreurs compréhensibles pour les phases d'analyse lexicale et syntaxique. Lors d'une erreur d'évaluation lors de l'expansion de macros ou de l'évaluation, le message d'erreur retourne le numéro de ligne de code en question et le type ou la valeur attendue. Toutefois, l'absence de trace rend difficile la compréhension des erreurs dans les fonctions récursives.

3.6.2 Tests unitaires

Nous avons doté notre évaluateur de plus de 20 fichiers de tests unitaires couvrant la plupart des aspects du langage. Chaque fichier peut contenir entre 10 et 20 tests unitaires. Chaque test unitaire comprend un bout de code Abitbol qui doit être envoyé à l'évaluateur et le résultat attendu. Une erreur peut également être un résultat attendu. Grâce à ces tests unitaires, nous sommes sûrs que notre évaluateur respecte bien la sémantique du langage décrit dans ce chapitre.

3.6.3 Difficultés rencontrées

3.6.3.1 Paresse extrême

Puisque le résultat d'évaluation est soit une valeur ou une erreur, nous avons créé un type Thunk qui regroupe ces deux possibilités. Il s'agit du type de

retour de l'évaluateur. L'implantation de la paresse de l'évaluateur Abitbol provient du fait que Haskell est lui même paresseux. Ainsi, pour implanter la paresse, les fonctions reçoivent simplement leurs arguments sous forme de Thunk. C'est-à-dire qu'on évalue les arguments avec l'évaluateur dont la valeur de retour est un Thunk. Mais cette évaluation est paresseuse en Haskell.

Par contre dans une première version de l'évaluateur, à chaque fois qu'un objet reçu en paramètre était passé à une autre fonction un autre Thunk était créé autour du premier. Ainsi, lors de fonctions récursives il pouvait y avoir plusieurs dizaines de Thunk imbriqués comme des poupées russes. Du point de vue sémantique cela ne pose pas de problème, car un Thunk de Thunk est équivalent au Thunk intérieur. Toutefois cela nuit aux performances. Cela nuit aussi beaucoup au débogage lorsqu'on veut inspecter la valeur d'un objet.

Nous avons corrigé cette faille en regardant pour chaque paramètre s'il s'agissait déjà d'un Thunk pour éviter d'en créer un deuxième.

3.6.3.2 Monades transformers

Une autre difficulté rencontrée fut la complexité des monades transformers [21]. Ces derniers sont utilisés pour mélanger plusieurs monades ensemble. En effet, lorsqu'on écrit dans un style monadique pur un évaluateur jouet un seul monade est souvent nécessaire. Par exemple le monade Either. Le résultat est soit une erreur ou une valeur. Par contre, dans un évaluateur plus complexe il faut également passer certains états pour faciliter le débogage et il faut également interagir avec le monde extérieur. Il faut donc utiliser les monades transformers pour imbriquer au moins ces trois monades (Either, State, IO) entre eux.

Déjà la complexité intellectuelle n'est pas évidente, mais notre principale difficulté avec les monades transformers est qu'il est difficile de modifier le code. La moindre modification implique qu'il faut adapter chaque fonction, car le type du monade transformer a changé. Au final, cela peut décourager de faire de simples modifications non essentielles comme l'insertion de logs. Nous n'avons pas trouvé de solution satisfaite à ce problème en Haskell.

Chapitre 4

Applications

Dans ce chapitre, nous allons démontrer deux applications des fonctions transparentes en combinaison avec la métaprogrammation. Cela répondra à notre objectif d'illustrer clairement les avantages que procurent les fonctions transparentes, notamment pour la métaprogrammation.

Nous avons vu au chapitre Travaux antérieurs que la manipulation symbolique des fonctions permet au programme Mathematica de faire aisément des calculs algébriques symboliquement. Dans le premier exemple, nous allons reprendre un exemple similaire de manipulation symbolique de polynômes et intégrer la composante de métaprogrammation. Pour notre deuxième application, nous allons créer une macro qui va optimiser un parseur écrit sous la forme d'une combinaison de parseurs plus primitifs. Ce genre de libraires, comme Parsec [18, 19], est fréquemment rencontré dans les langages fonctionnels.

Le premier exemple est en lien avec les motivations de l'auteur et des difficultés réellement rencontrées dans le domaine de la finance par ce dernier. Il ne s'agissait pas de manipuler symboliquement des polynômes mais

l'équation de Black-Scholes-Merton [10] ou celle du modèle SABR [9]. Ces équations sont bien sûr beaucoup plus complexes que de simples polynômes. Nous avons simplifié l'aspect mathématique car cela ne change pas la nature de notre démonstration. Le deuxième exemple vient du domaine des langages fonctionnels où beaucoup d'énergie est consacrée à écrire des libraires de parseurs efficaces [12, 18]. Nous pensons que notre approche offre une nouvelle façon de résoudre ce problème.

De plus, les deux applications, écrites dans le langage Abitbol, permettront de mettre en lumière les points suivants :

1. Relation entre les fonctions et les datatypes. Grâce à la transparence des fonctions, nous allons pouvoir convertir des fonctions en datatypes et vice-versa. Cela répond au besoin énoncé à l'introduction de pouvoir saisir la logique du programme contenu dans les fonctions et de manipuler cette logique explicitement à l'aide de datatypes.

2. Optimisations qui se limitent au besoin du code source. Ces optimisations, puisque faites sur mesure pour le code, sont faciles à écrire. Elles ne seraient pas assez robustes pour faire partie d'un compilateur qui doit faire face à tout type de code. Ainsi, le programmeur évite le long développement d'optimisations robustes et conservatrices qui peuvent être intégrées dans un compilateur. Il peut se permettre des optimisations plus agressives, car il connait le contexte d'application.

4.1 Manipulation symbolique

Dans cette section, nous allons utiliser les fonctions ouvertes pour manipuler symboliquement des polynômes. Comme mentionné à l'introduction, de façon

générale la manipulation symbolique permet au programmeur d'exprimer une fonction sous une forme simple et peut-être non optimale pour la transformer ensuite via métaprogrammation. Il économise donc du temps et s'évite des erreurs humaines.

Commençons par préciser notre exemple. Imaginons un programmeur dont le client lui fournit une série de fonctions objectifs de type polynomiale. Pour les besoins du programme, le programmeur doit combiner ces fonctions et calculer la dérivée du résultat. Le code source 4.1 illustre le problème.

Code source 4.1 : Mise en contexte pour la manipulation symbolique (Abitbol)

```
1 -- Les fonctions fournies par le client
2 poly1 x = x * x + 3 * x - 4;
3 poly2 x = x ^ 4 + x + 3;
4
5 -- Ce dont le programmeur a besoin
6 poly3 = -- dérivée (poly1 * poly2);
```

L'objectif du programmeur est donc d'obtenir la définition du troisième polynôme à la compilation. Une première solution est de sortir papier crayon, faire un peu d'algèbre et retranscrire le résultat. Il n'est jamais mauvais de faire un peu de calculs mathématiques. Mais cette solution est sujette aux erreurs humaines et le lien entre les deux premiers polynômes et le troisième est rompu. Si la définition des fonctions objectifs devait changer, penserait-on à mettre à jour la définition du troisième ?

Une deuxième possibilité est d'utiliser un outil extérieur, comme Mathematica ou Sympy [33] pour calculer la dérivée. Cette solution a pour avantage d'éviter les erreurs humaines, du moins de calcul et non de retranscription. Mais le lien est toujours rompu entre nos trois fonctions.

Une autre possibilité est d'écrire les fonctions sous forme de datatypes

directement. En effet, un polynôme peut être représenté soit sous la forme d'une fonction du style $f(x) = 3x^2 + 2x + 1$ ou par exemple par une liste de coefficients en ordre croissant du degré du terme soit $[1, 2, 3]$. L'addition, la multiplication et la dérivation se définissent très facilement lorsque le polynôme est sous la forme d'un datatype tel que le montre le code source 4.4 à la page 57. Mais cette solution impose au programmeur de représenter les polynômes sous la forme d'un datatype précis. Ceci n'est pas toujours possible. Peut-être que la manipulation symbolique n'est qu'un aspect mineur du programme et il faut que le polynôme soit une fonction pour être compatible avec le reste de celui-ci. De plus, permettons-nous de préciser qu'il est souvent plus lisible de définir un polynôme qui a des caractéristiques spéciales à l'aide d'une fonction, car l'écriture de cette dernière est très flexible. Le code source 4.2 fournit un bon exemple.

Code source 4.2 : Pour certains polynômes il est plus facile de les définir via une fonction qu'un datatype (Abitbol)

```
1 -- Un polynôme avec racines : -5, 5, 7
2 p x = (x + 5) * (x - 5) * (x - 7)
3
4 -- Quel est l'équivalent sous une liste de coefficients ?
5 p' = [175,-25,-7,1]
6
7 -- Pas très lisible sous cette forme
8 p'' = multP (multP [5, 1] [-5, 1]) [-7, 1]
```

Ainsi, ce qu'on voudrait pouvoir faire de façon simple et efficace est de définir les fonctions objectifs sous la forme de fonction. Pour calculer le troisième polynôme lors de la métaprogrammation, nous voudrions transformer les fonctions objectifs dans un datatypes facile à manipuler, combiner les fonctions objectifs et calculer la dérivée avec ces datatypes et ensuite convertir le résultat sous la forme d'une fonction.

Grâce aux fonctions transparentes, il est possible de faire cela simplement. Le code source 4.3 montre la solution finale. Les fonctions objectifs restent de simples fonctions faciles à lire et à écrire. La métaprogrammation retourne une fonction qui sera automatiquement mise à jour si la définition des fonctions objectifs change. Le code source 4.3 est le code que l'utilisateur doit écrire dans son programme. Les macros *asDataType* et *makePoly* peuvent être définies dans une librairie et réutilisées.

Code source 4.3 : Solution avec fonctions transparentes (Abitbol)

```
1 -- Les fonctions fournies par le client
2 poly1 x = x * x + 3 * x - 4;
3 poly2 x = x ^ 4 + x + 3;
4
5 -- Ce dont le programmeur a besoin
6 p1 = asDataType poly1;
7 p2 = asDataType poly2;
8 poly3 = $(makePoly (derivative (multP p1 p2)));
```

Il nous reste à préciser certains détails. Notamment la question de comment passer d'une fonction à un datatype qui représente un polynôme ? Il n'est certainement pas possible de détecter toutes les formes de fonctions qui pourraient représenter un polynôme. Mais l'usage des fonctions transparentes lors de la métaprogrammation vient ici résoudre ce problème. Puisque c'est le programmeur qui écrit la routine de transformation, il peut l'adapter à ses besoins. Il n'a donc pas à écrire une fonction aussi étoffée que si elle devait être intégrée à un compilateur ou à Mathematica. Le code source 4.5 à la page 58 montre la fonction coeff qui en quelques lignes seulement gère les cas les plus communs. Comme le code est modifiable par le programmeur, ce dernier peut facilement l'adapter à de nouveaux scénarios. Notons que ce code contient bien des défauts. Par exemple, il ne vérifie pas que la fonction est une fonction d'une variable. C'est un exemple de ce qui peut être ac-

ceptable en métaprogrammation, mais qui serait absurde d'inclure dans un compilateur commercial.

Il faut également pouvoir passer du datatype à une fonction. Cela peut se faire très simplement avec une petite fonction qui sert d'évaluateur pour le datatype. Le code source 4.6 montre un exemple. Mais il est aussi possible avec Abitbol de créer le corps de la fonction à partir du datatype représentant ses coefficients puis faire appel à la primitive « function ». Cette primitive prend une liste de paramètres, une définition de fonction et un environnement pour créer une fonction. Il s'agit de l'inverse de « fdefinition ». Nous avons illustré cette possibilité au code source 4.7. Pour cet exemple nous avons créé l'environnement en retirant de l'environnement standard toutes les définitions sauf celles de $+$, $*$ et $\hat{}$. Bien que plus complexe, cette fonction à l'avantage de retourner une fonction qui possède une définition similaire à celle qu'un programmeur aurait écrite.

4.2 Optimisation de parseurs

Notre deuxième démonstration portera sur l'optimisation de parseurs. Les langages fonctionnels utilisent souvent des librairies de parseurs écrits sous forme de fonctions pouvant se combiner [18, 19, 12]. Avant d'aller plus loin, nous allons brièvement décrire les constructeurs de parseurs dont nous aurons besoin pour cette application.

string *str* Accepte une chaîne de caractères et retourne un parseur capable de reconnaitre cette celle-ci.

seq *list* Parseur de séquencement. Accepte une liste de parseurs et retourne un parseur qui applique en séquence ceux-ci.

Code source 4.4 : Fonctions faciles à définir sur un datatype

```
-- Calculer la dérivée d'un polynôme
derivative [_] = [0];
derivative (_:xs) = derivative' xs 1
  where {
    derivative' [] _ = [];
    derivative' (x:xs) n = x * n : derivative' xs (n + 1);
    };

-- Addition
addP [x] (y:ys) = y + x : ys;
addP (x:xs) [y] = y + x : xs;
addP (x:xs) (y:ys) =
  let {ps = addP xs ys }
  in y + x : ps;

-- Multiplication
multP xs ys = multShift xs ys 0
  where {
    multShift [] ys _ = [0];
    multShift (x:xs) ys shift =
      let {a = replicate shift 0 ++ map (* x) ys;
           b = multShift xs ys (shift + 1);}
      in addP a b
  };
```

Code source 4.5 : Passage d'une fonction à un datatype polynôme (Abitbol)

```
-- Passage d'une fonction à un datatype
asDataType p = coeff (expr (fdefinition p))

-- Un simple entier
coeff (LitE (IntegerL n)) = Just [n];
-- Une variable
coeff (VarE _) = Just [0, 1];
-- Expression entre parenthèse
coeff (ParensE x) = coeff x;
-- Expression infixe +
coeff (InfixE x
               (VarE (LexicalName "+" []))
               y) |
      Just a <- coeff x,
      Just b <- coeff y = Just $ addP a b;
-- Négation
coeff (NegE y) |
      Just b <- coeff y = Just $ map (* (-1)) b;
-- Expression infixe -
coeff (InfixE x
               (VarE (LexicalName "-" []))
               y) |
      Just a <- coeff x,
      Just b <- coeff y = Just $ subP a b;
-- Expression infixe *
coeff (InfixE x
               (VarE (LexicalName "*" []))
               y) |
      Just a <- coeff x,
      Just b <- coeff y = Just $ multP a b;
-- Expression infixe ^
coeff (InfixE x
               (VarE (LexicalName "^" []))
               (LitE (IntegerL n))) |
      Just a <- coeff x,
      n >= 0 = Just $ expP a n;
-- Echec de la convertion
coeff _ = Nothing;
```

Code source 4.6 : Conversion d'un datatype en une fonction via un évaluateur (Abitbol)

```
-- Datatype vers une fonction
makePoly x = \y -> evalPoly y x 0

evalPoly _ [] _ = 0;
evalPoly v (x:xs) n = x * v ^ n + evalPoly v xs (n + 1);
```

lookAhead *test ok ko* Accepte trois arguments. Retourne un parseur qui essaie le premier argument et si celui-ci échoue reprend du début avec le troisième argument. Sinon il continue avec le deuxième en lui fournissant le résultat du premier. Cela permet d'éliminer les fuites mémoires inutiles, car on doit garder en mémoire la chaîne de caractères seulement le temps du premier parseur qui est en général assez court.

withString *parseur str* Parseur utile pour encoder le deuxième argument du parseur lookAhead. Il prend un parseur en premier argument et une chaîne de caractères et retourne un parseur qui fait la concaténation des deux si le premier argument réussit.

parseur1 </> parseur2 Parseur alternatif. Il accepte deux arguments et retourne un parseur qui essaie le premier argument et, en cas d'échec seulement, le second.

Nous sommes maintenant en mesure de fournir un exemple pour exprimer nos propos. Si on veut pouvoir analyser une chaîne de caractères pour savoir si elle contient la chaîne « Bon matin » ou « Bonjour », on pourrait être tenté d'écrire un parseur qui d'abord essaie la première alternative « Bon matin » et en cas d'échec la deuxième alternative « Bonjour ». Le code source 4.8 fournit un exemple.

Code source 4.7 : Conversion d'un datatype en une fonction via la primitive function (Abitbol)

```
-- Datatype vers une fonction
makePoly2 l = let {body = makeBody l 0;
                   params = [VarP "x"];
                   env = envKeep ["*", "+", "^"] stdEnv}
              -- La primitive function retourne
              -- un objet de type fonction
              in function params body env
  where {
    -- Cas terminaux
    makeBody [x] 0 = LitE (IntegerL x);
    makeBody [x] 1 = VarE "x";
    makeBody [x] n = InfixE (LitE (IntegerL x))
                            (VarE *)
                            (InfixE (VarE "x")
                                    (VarE "^")
                                    (LitE (IntegerL n)));
    -- 1 + ...
    makeBody (x:xs) 0 = InfixE (LitE (IntegerL x))
                               (VarE "+")
                               (makeBody xs 1);
    -- 2 * x + ...
    makeBody (x:xs) 1 =
      let {xs' = makeBody xs 2;
           x' = InfixE (LitE (IntegerL x))
                       (VarE *)
                       (VarE "x");}
      in InfixE x'
                (VarE "+")
                xs';
    -- 4 * (x ^ 2) + ...
    makeBody (x:xs) n =
      let {xs' = makeBody xs (n + 1);
           x' = InfixE (LitE (IntegerL x))
                       (VarE *)
                       (InfixE (VarE "x")
                               (VarE "^")
                               (LitE (IntegerL n)));}
      in InfixE x'
                (VarE "+")
                xs';};
```

Code source 4.8 : Un simple parseur alternatif causant bien des maux de tête (Abitbol)

```
-- "Bon matin" ou "Bonjour"
p = string "Bon matin" <|> string "Bonjour";
```

Or un parseur naïvement écrit comme celui-ci n'est pas très performant. D'abord, puisque les fonctions sont généralement opaques, le parseur alternatif n'a aucune façon de détecter que chaque phrase recherchée commence par « Bon ». Ensuite, ce parseur souffre d'un problème de fuite mémoire. En effet, il doit conserver en mémoire la chaîne de caractères jusqu'à ce que le premier parseur finisse son travail car il n'a aucune façon pour le parseur alternatif de savoir si la deuxième option est toujours viable au fur et à mesure que la première progresse. Ainsi, le programmeur doit réécrire à la main cette définition pour obtenir de meilleure performance. C'est dommage car cette définition est très lisible et facile à maintenir.

Ici encore la métaprogrammation et les fonctions transparentes peuvent venir aider le programmeur. Il sera possible d'écrire ce parseur de façon lisible et peu efficace et de le transformer en une combinaison efficace. En effet, il est possible de transformer le parseur alternatif <|> en une combinaison des parseurs lookAhead et seq. Le code source 4.9 montre qu'en une ligne le programmeur peut obtenir une version plus efficace. S'il faut modifier le parseur, il suffit de changer la version simple pour que ces changements soient automatiquement inclus dans la version optimisée.

Code source 4.9 : Optimisation de parseur (Abitbol)

```
p = string "Bon matin" <|> string "Bonjour";
pOpt = $(makeParser $ optimize $ asData p);
```

La stratégie utilisée pour obtenir le parseur optimisé est essentiellement la même que celle utilisée à l'application précédente. Transformer le parseur en un datatype facile à analyser et optimiser. Ensuite, convertir le nouveau parseur optimisé sous la forme d'une fonction. Le code source 4.9 est encore une fois le code que l'utilisateur doit écrire dans son programme. Les macros $makeParser$, $optimise$ et $asData$ peuvent être définies dans une librairie et réutilisées plusieurs fois.

Le code source 4.10 montre le code qui permet de passer d'une fonction parseur à un datatype le représentant. La fonction est très simple, elle prend le corps de la fonction à reconnaitre et le compare aux corps des parseurs connus. Le code source 4.11 fournit le code source pour optimiser les parseurs alternatifs et le code source suivant le code pour transformer le datatype en fonction.

Code source 4.10 : Passage d'un parseur à un datatype (Abitbol)

```
-- Datatype pour représenter un parseur
data StringP value;
data SeqP value;
data LookAheadP pTest pOk pKo;
data WithStringP p1 value;
data AltP p1 p2;

-- Passage d'une fonction à un datatype
-- Illustration pour StringP et AltP
asData x =
  let{ Lambda _ def env = fdefinition x;
       Lambda [VarP altP1, VarP altP2] altDef _ =
            fdefinition (<|>);
       Lambda [VarP stringParam] stringDef _ =
            fdefinition string;
       }
  in if def == expr stringDef
     then (StringP (envLookup stringParam env))
     else
       if def == expr altDef
       then AltP (asData (envLookup altP1 env))
                 (asData (envLookup altP2 env))
       else x
  ;
```

Code source 4.11 : Optimisation d'un datatype parseur (Abitbol)

```
-- Deux alternatives égales
optimize (AltP x y) | x == y = optimize x;

-- Deux parseurs de chaîne de caractères
optimize (AltP (StringP x) (StringP y)) =
  case analyze x y of {
    ([], x, y) -> makeLookAhead x y;
    -- La première alternative prévaut
    (c, [], y) -> StringP c;
    (c, x, y) -> SeqP [StringP c, makeLookAhead x y];
    }

  where {analyze (x:xs) (y:ys) | x == y =
            let {(c, p1, p2) = analyze xs ys}
            in (x : c, p1, p2);
         analyze xs ys = ([], xs, ys);

         makeLookAhead (x:xs) ys =
          LookAheadP (StringP [x])
                     (WithStringP (StringP xs) [x])
                     (StringP ys);

        };

-- Aucune optimisation possible
optimize x = x;
```

Code source 4.12 : Datatype parseur vers fonction parseur (Abitbol)

```
-- Construire un parseur depuis son datatype
makeParser (StringP value) = string value;
makeParser (WithStringP p1 value) =
  withString (makeParser p1) value;
makeParser (SeqP value) =
  seq (map makeParser value);
makeParser (LookAheadP pTest pOk pKo) =
  lookAhead (makeParser pTest)
            (makeParser pOk) (
            makeParser pKo);
makeParser (AltP p1 p2) =
  alt (makeParser p1) (makeParser p2);
```

Chapitre 5

Discussion

Dans ce chapitre, nous allons analyser davantage ce qu'implique la transparence des fonctions lors de la phase de métaprogrammation. Nous allons entreprendre notre discussion en illustrant la simplicité sémantique de cette approche. Nous allons également aborder la notion de performance et le problème d'hygiène. Abitbol contourne habilement ce problème, mais il est intéressant de voir en quoi les fonctions transparentes affectent les solutions connues au problème d'hygiène.

5.1 Simplicité sémantique

Notre solution pour une meilleure expérience de métaprogrammation s'illustre par sa simplicité sémantique. Maintenant tout objet créé par le programmeur peut être déconstruit, y compris les fonctions. Les fonctions sont donc un objet comme les autres, leur traitement est uniforme avec celui des autres objets.

Aussi, cela permet d'exposer une grande partie du code du programme

sans que le programmeur ait à apprendre de nouvelles notions sémantiques. En effet, la transparence des fonctions expose les paramètres, le corps des fonctions et l'environnement de fermeture. Toutes ces notions sont déjà acquises de la part du programmeur. Même si non explicite avec les fonctions opaques, l'environnement de fermeture est lié à la portée lexicale qui elle devrait être bien comprise par le programmeur. Ainsi, contrairement à Scheme, avec Abitbol il n'y a pas deux notions distinctes que sont les macros et les fonctions. Et contrairement à Template Haskell, il est possible d'analyser le code source du programme.

Un autre aspect intéressant de notre proposition est qu'elle ne nécessite aucune modification à la façon de déclarer les fonctions. Elle pourrait très bien être intégrée à un langage comme Haskell et le code existant pourrait être analysé. Puisqu'aucune action n'est requise de la part du programmeur pour obtenir les fonctions ouvertes, cela n'alourdit pas le travail de ce dernier.

5.2 Performance à l'exécution

Nous ne proposons pas d'inclure la transparence des fonctions à l'exécution. Toutefois, une expérience similaire a déjà été réalisée dans la communauté LISP avec les Fexprs. Dans les langages LISP, les Fexprs étaient un type de fonction disponible à l'exécution qui recevaient ses arguments non évalués. Par exemple la fonction recevait ses arguments sous la forme d'un ASA comme pour les macros LISP, mais avec en plus l'environnement courant nécessaire à leur évaluation. Ainsi, une Fexpr pouvait évaluer ses arguments à l'aide de l'environnement et agir comme une fonction normale. Mais elle pouvait aussi simplement manipuler l'ASA de ses arguments comme une macro.

Ainsi, tel qu'expliqué au chapitre Introduction, puisque la Fexpr peut

manipuler l'ASA le compilateur ne peut pas modifier l'expression d'un paramètre pour l'optimiser car cela pourrait affecter la valeur de retour de la Fexpr. Pitman [24] a démontré en 1980 qu'il est impossible en général pour une analyse statique à la compilation de déterminer si une fonction est une Fexpr ou non. Donc au final le compilateur ne peut pas optimiser les appels de fonctions.

Les fonctions transparentes amènent des difficultés similaires. Tout comme pour les Fexpr, la possibilité d'obtenir le code source et son environnement limite les optimisations que peut faire le compilateur. Mais la grande différence avec les Fexpr est que notre thèse principale s'intéresse aux fonctions transparentes lors de la métaprogrammation. Ainsi, leur absence lors de l'exécution permet d'éviter le problème des Fexpr. Le code, une fois compilé, s'exécutera de la même façon que s'il avait été écrit avec des fonctions opaques. L'impact en performance est uniquement présent à la compilation.

5.3 Performance à la compilation

Nous allons maintenant analyser le cout de performance de la thèse principale, soit rendre les fonctions transparentes lors de la phase de métaprogrammation. Nous allons comparer notre approche avec la performance des macros LISP et celles de Template Haskell qui sont déjà bien établies.

Sans la transparence des fonctions, notre mécanisme de macros décrit à l'algorithme 3.1 est très similaire à un algorithme d'expansion de macros pour Template Haskell. Il n'y a pas de coût supplémentaire par rapport à ce dernier. Nous allons donc nous concentrer sur la transparence des fonctions. La transparence des fonctions se traduit techniquement par le triplet de définition défini à la section 1.4.1. Les deux premiers éléments du triplet

de définition ne représentent pas de difficulté en soi. Les paramètres et le corps de la fonction sont des structures de données statiques déjà connues à la compilation. Leur impact sur la performance est donc négligeable. Par contre, le troisième élément du triplet, l'environnement de fermeture, est plus délicat. Nous allons analyser son impact sur l'utilisation mémoire et le temps de compilation.

5.3.1 Utilisation mémoire

En l'absence d'optimisation faite par le compilateur lors de la phase de métaprogrammation, les fonctions transparentes ne sont pas un cout mémoire supplémentaire. Les macros LISP et celles de Template Haskell doivent bien avoir le corps de la fonction et la valeur de ses variables libres en mémoire également. Peut-être qu'elles n'y seraient pas représentées explicitement sous la forme d'un environnement de fermeture, mais ces données doivent être en mémoire pour le bon fonctionnement de la phase d'expansion de macros.

La seule contrainte supplémentaire est qu'il n'est plus permis au compilateur d'optimiser les fermetures. Ainsi, si certains objets volumineux sont captés par une fermeture sans que cela soit nécessaire, il y aura une fuite mémoire. Ce point négatif de notre approche est nécessaire seulement lors de la compilation. Lors de l'exécution le compilateur pourra faire l'optimisation s'il en est capable, car les fonctions restent opaques.

Nous pensons donc que la transparence des fonctions à la métaprogrammation n'est pas une contrainte forte sur l'utilisation de la mémoire et qu'en cas de difficulté le programmeur dispose déjà des outils et techniques pour gérer ces problèmes liés aux variables libres.

5.3.2 Temps de compilation

La présence du triplet de définition lors de la phase de métaprogrammation ne devrait pas beaucoup ralentir cette dernière dans le cas où l'évaluation des macros est faite par un simple évaluateur. La performance devrait être similaire à celles de macros LISP et de Template Haskell, car ces derniers aussi évalueraient simplement le corps des fonctions et leurs variables libres comme pour l'algorithme 3.1.

Par contre LISP et Template Haskell pourraient faire des optimisations durant l'évaluation des macros en optimisant les fonctions, car ils ne sont pas tenus d'en fournir la définition à l'utilisateur. Mais en présence d'un langage paresseux, il est possible de faire cela également avec les fonctions transparentes. Il suffit pour le compilateur de générer pour chaque fonction un quintuple plutôt qu'un triplet. Les trois premiers éléments sont les mêmes que nous connaissons, mais calculés de façon paresseuse. Leur empreinte mémoire est donc toujours présente. Le quatrième contient un corps de fonction optimisé et le cinquième l'environnement de fermeture de ce dernier. Lors de l'évaluation des macros, seulement le quatrième et le cinquième élément sont utilisés à moins que le programmeur demande la définition. Auquel cas le compilateur peut retourner le deuxième et le troisième élément. Il y a donc une utilisation mémoire légèrement augmentée, car les fonctions sont possiblement définies deux fois en mémoire. Mais cela permet par contre de faire les mêmes optimisations qu'il serait possible de faire pour les macros LISP ou Template Haskell.

Au final, les macros en Abitbol peuvent offrir des performances similaires aux macros LISP ou Template Haskell. L'utilisation mémoire peut s'en trouver légèrement augmenté par contre, mais encore une fois ce coût n'est présent que lors de la phase d'expansion des macros.

5.4 Maintenance

Il existe tout de même un coût supplémentaire à payer pour obtenir des fonctions transparentes à la métaprogrammation. À moins d'autoriser les fonctions transparentes à l'exécution du programme, leur présence lors de la métaprogrammation nécessite d'avoir un interpréteur ou compilateur différent de celui utilisé pour l'exécution. C'est un point faible de cette approche qui n'affecte pas directement les performances, mais la facilité de maintenir et de faire évoluer le compilateur du langage. Rien n'est gratuit, la facilité d'inspection du code source pour le programmeur vient au prix de la complexification du compilateur.

5.5 Hygiène

Comme nous l'avons vu à la section Métaprogrammation du chapitre Abitbol, notre implémentation contourne astucieusement le problème d'hygiène. Cela se fait notamment au prix qu'il n'est pas possible pour le programmeur d'expanser des déclarations.

Si nous voulions avoir une telle possibilité, deux solutions s'offrent à nous à première vue. La première est de faire comme LISP et permettre une expansion non hygiénique. Cette solution est facile à implanter et ne nécessite aucun changement au langage. Par contre elle ignore l'ensemble des progrès fait par la communauté Scheme au niveau de l'hygiène des macros.

La deuxième solution est de faire comme Scheme et inclure un mécanisme d'hygiène. Le plus gros du travail concerne les identificateurs. Plutôt que d'être un simple datatype qui indique le nom de la variable sous forme de chaîne de caractères, il faut un mécanisme qui capture également la por-

tée lexicale de cet identificateur. Plusieurs mécanismes [7, 17, 3, 6] existent, mais pour chacun les datatypes (ex. : objets syntaxiques) qui représentent les identificateurs sont complexes. Ainsi, modifier le datatype qui encode les identificateurs et exposer le programmeur explicitement au mécanisme d'hygiène est une possibilité. Après tout si le programmeur veut avoir plus de contrôle sur la compilation, il doit accepter la complexité qui vient avec ce contrôle. De plus, avec l'ajout de quasi-quotation au langage le programmeur doit rarement manipuler directement les datatypes des identificateurs [22].

Mais avec les fonctions transparentes, il existe une alternative qui se situe entre ces deux scénarios. Cela démontre encore la simplicité sémantique et le peu de changement que doit amener la transparence des fonctions. D'un côté il suffit de garder opaque le mécanisme d'hygiène comme dans Template Haskell ou Racket. De l'autre, la seule façon pour le programmeur d'avoir accès au code source est de passer par les fonctions. Mais il obtient aussi l'environnement de fermeture. Or cet environnement est également une façon de capturer la portée lexicale d'un identificateur. Ainsi, plutôt que d'avoir des datatypes complexes pour les identificateurs, il est possible de les garder simples puisque la fonction de l'environnement de fermeture est justement le respect de la portée lexicale. Cela démontre aussi l'importance d'avoir l'environnement de fermeture dans le triplet.

Le code source 5.1 donne un exemple pour faciliter la compréhension. Pour cet exemple nous supposons qu'Abitbol peut expanser des définitions et possède la quasi-quotation comme Template Haskell. La variable foo est définie de façon globale. L'expansion de la macro bar introduit la fonction baz qui elle contient une référence à un autre identificateur foo définie dans la macro. Lorsque le programmeur demande la définition de baz, il reçoit bien avec celle-ci un simple identificateur foo. Sauf que la valeur associée dans

l'environnement de fermeture est celle de la variable locale et non celle de la variable globale. Ainsi l'hygiène des macros est respectée.

Code source 5.1 : L'environnement sert de contexte lexical pour l'hygiène des macros (Abitbol)

```
-- Variable globale foo
foo = 1;

-- Fonction qui va retourner la déclaration
-- de la fonction baz
-- [quote|...|] est un mécanisme de quasi quotation
bar = let {foo = 0}
      in [quote| baz = \x -> foo + x|];

-- Appel de macro pour obtenir la fonction baz
-- Le mécanisme d'hygiène doit lier la variable
-- foo à sa définition locale dans bar (foo = 0)
-- Ce mécanisme est opaque pour le programmeur
$bar;

-- Valeur de foo dans l'environnement de fermeture
-- de baz. L'environnement sert de contexte lexical
-- monFoo = 0
monFoo = envlookup "foo" (env (fdefinition baz));
```

5.6 Implémentation à la Racket

Au chapitre 3, nous avons détaillé notre implantation des fonctions transparentes et l'algorithme d'expansion de macros pour le langage Abitbol. Tel que mentionné à la section 3.5, notre algorithme repose sur la paresse du langage. Nous allons maintenant détailler une autre implantation possible faite à l'aide du système de métaprogrammation de Racket. Cela nous permet d'illustrer une implantation dans un langage strict.

L'implantation des fonctions transparentes lors de la phase de méta-

programmation peut se faire en Racket à l'aide de macros. En effet, le système de macros est assez puissant pour permettre d'ajouter cette fonctionnalité au langage. Toutefois, le fait d'utiliser des macros implique certains désavantages que nous n'avons pas avec notre approche. Une comparaison détaillée des deux implantations sera faite à la fin de cette section.

Il est possible d'implanter les fonctions transparentes lors de la phase de métaprogrammation en Racket notamment parce que Racket permet le partage d'information entre les macros, que celles-ci prennent leur argument sous forme d'ASA et qu'elles sont hygiéniques. La première étape est de définir une nouvelle forme de définition de fonction qui gardera en mémoire la définition et la rendra accessible. La macro define-transparent effectue ce travail. Elle accepte une définition standard, l'enregistre dans la variable globale fdefinitions et retourne une expression représentant la déclaration de fonction telle que l'aurait écrit le programmeur. La deuxième étape est de définir une nouvelle forme pour obtenir le triplet de définition. La macro fdefinition accepte un nom d'identificateur et retourne la définition associée dans la variable globale fdefinitions si possible. Cette définition comprend les noms des paramètres et le corps de la fonction. L'environnement de fermeture est absent, mais il est représenté dans cette implantation par la variable globale fdefinitions qui est toujours accessible. Le code source 5.2 à la page 74 donne les définitions de ces deux macros.

Puisque Racket utilise des objets syntaxiques pour représenter les identificateurs, chaque identificateur inséré dans la variable fdefinitions restera unique. Ainsi, il n'est pas possible de cacher une définition en introduisant une autre variable du même nom. Les deux identificateurs, bien que portant le même nom dans le code source, seront distincts. La variable globale fdefinitions peut donc réellement servir d'environnement de fermeture, car les

Code source 5.2 : Implantation des fonctions transparentes en Racket

```
#lang racket

; Liste qui va contenir les définitions des fonctions
; et le nom des paramètres
(define-for-syntax fdefinitions (list))

; La macro define-transparent crée une fonction
; et mémorise sa définition
; S'inspire d'une définition similaire fournie
; par Antoine B sur la mailing list de Racket
; (users@racket-lang.org)
(define-syntax (define-transparent stx)
  (syntax-case stx ()
    [(_ (name . params) value)
     (begin (set! fdefinitions
                  (cons (cons #'name (cons #'params #'value))
                        fdefinitions))
            #`(begin (define #,(cons #'name #'params) value)))]))

; Exemples d'utilisation
(define-transparent (foo x) (+ x 3))
(define-transparent (bar y) 4)

; La macro fdefinition retourne toutes les définitions connues
; ou la définition pour un identificateur particulier
(define-syntax fdefinition
  (lambda (x)
    (syntax-case x ()
      ((_)
         #`'#,fdefinitions)
      ((_ name)
       (let ((x (assoc #'name fdefinitions free-identifier=?)))
         #`(if '#,x (cdr '#,x) #f))))))

(display (fdefinition)) (newline)
(display (fdefinition foo)) (newline)
(display (fdefinition bar)) (newline)
; Résultat :
;((bar (y) . 4) (foo (x) + x 3))
;((x) + x 3)
;((y) . 4)
```

variables libres seront toujours associées à la bonne définition même si elles ne sont plus à portée lors de l'appel à la macro fdefinition. Le code source 5.3 à la page 75 montre ce mécanisme d'hygiène à l'œuvre.

Code source 5.3 : L'hygiène permet de distinguer deux identificateurs ayant le même nom (Racket)

```
#lang racket

(define-transparent (foo x) (+ x 3))
(define-transparent (baz z) (foo (+ 1 z)))

; fooScope retourne la définition de l'identificateur
; en tête de l'ASA du corps de la fonction donnée
; en paramètre
(define-syntax fooScope
  (lambda (x)
    (syntax-case x ()
      ((_ name)
       (letrec ((x (assoc #'name fdefinitions
                          free-identifier=?))
                (fooSymbol (car (syntax->list (cddr x))))
                (y (assoc fooSymbol fdefinitions
                          free-identifier=?)))
         #`(if '#,y (cdr '#,y) #f))))))

(define (testScope)
  ; Nouvelle définition pour foo
  (define-transparent (foo x) (+ x 1))
  (begin
   (display (fdefinition)) (newline)
   (display (fooScope baz))))

(testScope)

; Résultat. On note 2 définitions pour foo.
; La dernière est la bonne pour la variable libre de baz
;((foo (x) + x 1) (baz (z) foo (+ 1 z)) (foo (x) + x 3))
;((x) + x 3)
```

Ainsi il est possible en Racket d'obtenir un mécanisme de fonctions transparentes à l'aide des macros. Notons que nous avons donné que des exemples

de variables globales, mais qu'il est possible de généraliser le mécanisme aux variables locales. Ce mécanisme peut également se généraliser aux objets de façon à ce que toutes les variables libres d'une fonction puissent être accessibles. Nous allons comparer cette approche avec celle d'Abitbol.

Avantages

Les avantages de l'approche de Racket par rapport celle d'Abitbol sont les suivants :

- Ce mécanisme permet d'inclure les fonctions transparentes lors de la métaprogrammation dans le langage Racket sans modification du compilateur.
- Seules les fonctions définies avec define-transparent auront leur définition d'incluse à la métaprogrammation. Si une fonction comprend une variable libre coûteuse à calculer, alors il est possible de l'exclure de la phase de métaprogrammation en utilisant une définition standard et opaque pour celle-ci. Cela permet de résoudre le problème décrit à la section 3.5.1 sur les performances de notre algorithme d'expansion de macros avec un mode d'évaluation strict.

Désavantages

Les désavantages de l'approche de Racket par rapport à celle d'Abitbol sont les suivants :

- Ce mécanisme nécessite l'intervention du programmeur. Il ne s'agit pas seulement de modifier la déclaration d'une fonction dont on veut obtenir la définition à la métaprogrammation, mais également de toutes

les variables libres si celles-ci doivent être inspectées également. Le nombre de modifications à faire peut donc être important.

- Dans notre mécanisme pour Abitobl, la paresse garantit que si l'on élimine l'usage d'une variable lors de la métaprogrammation cette valeur ne sera plus calculée. Dans le mécanisme pour Racket, si l'on a plus besoin d'une définition il faut retourner modifier la déclaration de la variable. Une intervention manuelle est donc requise également lorsqu'on ne souhaite plus utiliser la variable lors de la métaprogrammation. De plus, rien n'indique qu'une déclaration avec define-transparent n'est plus utilisée.

- Le code source existant n'est pas toujours accessible au programmeur pour que celui-ci puisse modifier les déclarations et utiliser define-transparent. En effet, les développeurs d'une librairie externe pourraient refuser de modifier leurs déclarations.

- Le code source devient plus complexe à comprendre pour un programmeur non initié aux fonctions transparentes. Pourquoi certaines définitions utilisent-elles define et d'autres define-transparent ?

- Ce mécanisme exige la manipulation explicite d'objets syntaxique. Nous avons vu à la section précédente que le mécanisme d'Abitbol peut également fonctionner avec de simples identificateurs.

- Les macros Racket ne sont pas composables et ne peuvent pas être passées en paramètre à une autre macro. Abitbol utilise les fonctions comme macros et donc celles-ci sont composables.

Au final cette implantation permet d'obtenir les avantages des fonctions transparentes lors de la phase de métaprogrammation moyennant un effort

supplémentaire du programmeur. Bien qu'il soit moins facile de maintenir le code source avec ce mécanisme nécessitant un appel explicite à define-transparent, il permet au programmeur de contrôler les définitions qui seront incluses lors de la phase de métaprogrammation dans un langage strict.

Chapitre 6

Conclusion

Avec ce mémoire, nous avons soutenu la thèse que les fonctions devraient être transparentes lors de la phase de métaprogrammation. Nous avons démontré que cela procure de nombreux avantages et n'impacte pas les performances à l'exécution.

Nous avons vu au chapitre Travaux antérieurs que certains langages présentent déjà cette caractéristique lors de l'évaluation et que cela leur procure un grand avantage pour le calcul symbolique. Nous avons également réalisé qu'Abitbol va plus loin en fournissant explicitement l'environnement de fermeture qui est absent dans Mathematica et Python. Le chapitre Travaux antérieurs nous a également permis de constater que la métaprogrammation telle que faite actuellement ne permet pas de réifier automatiquement le code source tel que nous le proposons.

Le chapitre Abitbol a permis de mettre en lumière que les fonctions transparentes sont faciles à implanter. Aussi, nous avons vu que cette fonctionnalité n'est robuste qu'avec un langage pur, mais qu'il est réaliste de l'implanter dans un langage faisant la distinction entre constantes et variables. Nous

avons également fourni un exemple d'un mécanisme de métaprogrammation permettant d'éviter le problème d'hygiène tout en conservant la possibilité de manipuler des fonctions lors de cette phase de la compilation. Ce mécanisme repose sur la paresse du langage mais le chapitre Discussion fournit également un mécanisme qui fonctionne dans un langage strict.

Au chapitre Applications nous avons donné deux exemples d'application des fonctions transparentes lors de la phase de métaprogrammation. Ces exemples ont permis d'illustrer divers avantages dont :

- Le passage de fonctions à un datatype et vice-versa.
- La possibilité d'écrire des optimisations sur mesure et ne nécessitant pas la robustesse nécessaire à celles incluses dans un compilateur.

Surtout, ces applications ont répondu à notre motivation principale qui était d'économiser du temps au programmeur, libérer ce dernier de la gestion des détails et encapsuler des connaissances d'experts dans des interfaces simples. La première application de manipulation symbolique démontre qu'Abitbol peut répondre à ces objectifs en déléguant des calculs mathématiques aux macros plutôt qu'au programmeur. Un tel outil aurait été grandement apprécié par l'auteur lorsqu'il travaillait dans le courtage électronique d'options sur actions. La deuxième application permet d'écrire rapidement et simplement des parsers sans avoir à se concentrer sur la performance du code, mais plutôt sur sa sémantique.

Lors de la discussion, nous avons illustré la simplicité sémantique de notre approche et montré qu'elle n'exige pas de travail supplémentaire de la part du programmeur. Il y a toutefois un coût en maintenance supplémentaire puisque la sémantique entre l'exécution et la compilation peut être différente si les fonctions transparentes sont absentes de l'exécution. Lors de la discussion

nous avons également argumenté qu'il existe une méthode simple de rendre hygiénique la métaprogrammation avec fonctions transparentes.

6.1 Travaux futurs

Nous terminons ce mémoire en donnant quelques pistes de travaux futurs.

Nous n'avons pas abordé le thème de la compilation séparée ou d'un système de modules. La présence de fonctions transparentes force à maintenir la version originale du code source, car elle pourrait être demandée par le programmeur via la transparence des fonctions. Or un des buts de la compilation séparée est de pouvoir compiler un module et n'avoir qu'à importer la version compilée.

Également, bien que très simple sur le plan sémantique nous ignorons s'il est réaliste d'intégrer les fonctions transparentes aux compilateurs actuels tel que GHC (Glasgow Haskell Compiler) pour Haskell ou celui de Racket (et non pas sous forme de macros). Tel que vu au chapitre Discussion notre approche à l'avantage de pouvoir utiliser le code existant. Il serait donc nettement plus utile d'ajouter les fonctions transparentes à un langage existant et que celles-ci soient automatiquement disponibles au programmeur.

Bibliographie

[1] Aho, A. V. *Compilers : Principles, Techniques and Tools 2/e*. Pearson Education, 2003.

[2] Bawden, A. First-class macros have types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000), ACM, pp. 133–141.

[3] Clinger, W., and Rees, J. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1991), ACM, pp. 155–162.

[4] Dmitriev, S. Language oriented programming : The next programming paradigm. *JetBrains onBoard 1*, 2 (2004).

[5] Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and symbolic computation 5*, 4 (1993), 295–326.

[6] Flatt, M. Composable and compilable macros : : you want it when ? In *ACM SIGPLAN Notices* (2002), vol. 37, ACM, pp. 72–83.

[7] Flatt, M., Culpepper, R., Darais, D., and Findler, R. B. Macros that Work Together. *Journal of Functional Programming 22*, 02 (2012), 181–216.

[8] Graham, P. *On lisp*. Prentice Hall, 1994.

[9] Hagan, P. S., Kumar, D., Lesniewski, A. S., and Woodward, D. E. Managing smile risk. *The Best of Wilmott* (2002), 249.

[10] Haug, E. G. *The complete guide to option pricing formulas*. McGraw-Hill Companies, 2007.

[11] Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. A history of Haskell : being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007), ACM, pp. 12–1.

[12] Hutton, G., and Meijer, E. Monadic parser combinators. Tech. Rep. NOTTCS-TR-96-4, University of Nottingham, 1996.

[13] Johnson, S. C. *Yacc : Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[14] Jones, E., Oliphant, T., and Peterson, P. SciPy : Open source scientific tools for Python. *http ://www. scipy. org/* (2001).

[15] Kelsey, R., Clinger, W., and Rees, J. Revised5 Report on the Algorithmic Language Scheme. Tech. rep., Feb. 1998.

[16] Kiselyov, O. Macros that compose : Systematic macro programming. In *Generative Programming and Component Engineering* (2002), Springer, pp. 202–217.

[17] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), ACM, pp. 151–161.

[18] Leijen, D. *Parsec, a fast combinator parser*. 2001.

[19] Leijen, D., and Meijer, E. Parsec : Direct style monadic parser combinators for the real world.

[20] Lesk, M. E., and Schmidt, E. *Lex : A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[21] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), ACM, pp. 333–343.

[22] Mainland, G. Why it's nice to be quoted : quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop* (2007), ACM, pp. 73–82.

[23] Marlow, S., and others. Haskell 2010 language report.

[24] Pitman, K. M. Special forms in Lisp. In *Proceedings of the 1980 ACM conference on LISP and functional programming* (1980), ACM, pp. 179–187.

[25] Python. `https://www.python.org/`. Accessed : 2015-03-15.

[26] Robison, A. D. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande* (2001), ACM, pp. 1–10.

[27] Sheard, T., and Jones, S. P. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (2002), ACM, pp. 1–16.

[28] Shutt, J. N. *Fexprs as the basis of Lisp function application or $ vau : the ultimate abstraction*. PhD thesis, Brown University, 2010.

[29] Smith, B. C. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1984), ACM, pp. 23–35.

[30] Sperber, M., R Kent, D., Flatt, M., and Van Straaten, A. Revised6 Report on the Algorithmic Language Scheme. Tech. rep., Sept. 2007.

[31] Steele, G. L. *Common LISP : the language*. Digital press, 1990.

[32] Taha, W., and Sheard, T. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science 248*, 1 (2000), 211–242.

[33] Team, S. D. SymPy : Python library for symbolic mathematics, 2015.

[34] Van Der Walt, S., Colbert, S. C., and Varoquaux, G. The NumPy array : a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[35] Wadler, P. Monads for functional programming. In *Advanced Functional Programming*. Springer, 1995, pp. 24–52.

[36] WAND, M. The theory of fexprs is trivial. *Lisp and Symbolic Computation 10*, 3 (1998), 189–199.

[37] WARD, M. P. Language-oriented programming. *Software-Concepts and Tools 15*, 4 (1994), 147–161.

[38] WOLFRAM, S. *The MATHEMATICA® book, version 4*. Cambridge university press, 1999.