UFR d'Informatique

le cnam
Conservatoire national
des arts et métiers

Université Paris Diderot

Rapport de Stage

Auteur :
Elie Canonici Merle

Référent :
Pierre Courtieu
Enseignant chercheur au CNAM

Master Ingénierie des Algorithmes et des Programmes
Parcours Langages et Programmation
Université Paris Diderot

15 septembre 2017

Je tiens tout d'abord à remercier Pierre Courtieu et Christophe Genolini pour la confiance qu'ils m'ont accordée et la liberté qui m'a été laissée d'explorer des idées nouvelles. Je tiens égalemment à remercier les enseignants chercheurs du CNAM pour leur bonne humeur qui y fait régner une ambiance plaisante.

Enfin je souhaite remercier les enseignants de Paris Diderot qui m'ont permis de découvrir quels étaient les aspects de l'informatique pour lesquels j'avais de l'affection sans le savoir.

Table des matières

Chapitre 1

Présentation du cadre du stage : Zébrys/CNAM

Zébrys est une start-up basée à Toulouse dont le secteur d'activité consiste à développer des outils liés à l'utilisation du langage de programmation R, notamment un IDE spécialisé pour le développement en R et la manipulation des données statistiques associées. Un des objectifs de Zébrys, outre la mise sur le marché d'un environement de développement adapté, est de mettre à disposition de la communauté R++, un langage inspiré de R mais bénéficiant de performances accrues et de garanties dont R ne disposent pas à ce jour.

Dans ce but Zébrys s'est rapprochée du milieu académique, en la personne de Pierre Courtieu, enseignant chercheur au CNAM au sein du laboratoire CEDRIC.

1.1 Contexte

Zébrys est composée de 5 salariés travaillant sur l'IDE spécialisé susmentionné et se tourne vers la conception de R++. C'est dans cette démarche que s'est monté le projet de prototyper un ensemble d'outils, dans le but de prendre la mesure de la tâche et de disposer d'une expérience suffisante pour faire les choix nécessaires à la conception d'un langage qui aurait vocation à vivre avec l'héritage de R, pour le meilleur plutôt que pour le pire. Il s'agit donc dans un premier temps d'un travail visant à déterminer les limitations imposées par R et sa sémantique afin de faire les choix les plus judicieux possibles vis-à-vis de R++ et proposer à terme une réelle amélioration sans perdre en chemin la communauté déjà bien établie et grandissante de R.
R en quelques chiffres, c'est plus de 10 000 paquets disponibles en téléchargement sur le CRAN [1], soit plusieurs millions de lignes de codes qui font son succès et l'établissent si ce n'est comme le langage le plus utilisé pour le calcul dans le domaine des statistiques tout du moins comme l'un d'entre eux. Il s'agit aussi et surtout d'un langage employé en majorité par des utilisateurs venant d'horizons divers et variés, parfois très éloignés des sciences informatiques. Nous avons donc former une équipe de deux stagiares pour mener à bien ce travail, Rémi Bossut de l'École nationale supérieure d'électrotechnique, d'électronique, d'informatique, d'hydraulique et des télécommunications de Toulouse, et moi même. Le travail a été divisé entre nous de la manière suivante :

- Rémi devait implémenter l'arbre de syntaxe abstrait sur lequel nous étions amené à travailler et un prototype de compilateur de R vers du C++
- J'étais chargé de l'analyse lexicale et syntaxique, l'implémentation d'un outil d'analyse statique, la création d'un système de type et l'implémentation de son inférence.

Nous avons donc travaillé à distance mais en maintenant une communication constante entre Paris et Toulouse. C'est dans ce cadre que s'est effectué mon stage, sous la direction de Pierre Courtieu dans les locaux du laboratoire CEDRIC du CNAM, en partenariat avec Zébrys.

1. CRAN (The Comprehensive R Archive Network) est un réseau de serveurs web et ftp stockant des versions identiques et à jour de code et de documentation pour R.

Chapitre 2

Présentation du sujet

R est un langage se voulant adapté au calcul statistique, à la manipulation et à la visualisation des données associés à ces calculs. Ayant déjà une longue histoire et disposant de plusieurs milliers de packages prêts à l'emploi pour l'utilisateur qu'il soit néophyte ou experimenté, l'écosystème de R est dense, ce qui explique en partie le succès du langage. Il ne parait à priori pas raisonnable d'expliquer que ces millions de lignes de code sont tout simplement obsolètes et qu'il faudrait désormais s'adapter à un nouveau langage à la syntaxe et à la sémantique diamétralement différentes de tout ce à quoi les programmeurs R sont habitués. Une telle approche serait certainement vouée à ne jamais dépasser le stade de prototype et ne prendrait pas en considération la population visée par R.

Pour comprendre la philosophie qui réside derrière R et ses utilisateurs, il convient de se plonger dans un petit bout de son histoire. R est un langage qui est né en 1993, date de démarrage du projet de recherche de Ross Ihaka et Robert Gentlemen à l'université d'Auckland, qui a vu son aboutissement par la mise à disposition de la première version officielle de R en 2000. Il s'est construit comme une dérivation du langage S qui avait été développé par Rick Becker, John Chambers, Doug Dunn, Paul Tukey, et Graham Wilkinson dans les laboratoires de Bell.

R a hérité de l'approche de S en ce sens qu'il existe une ambiguité intraseque à ces deux langages. Ils sont tous deux plus que des langages de programmation, ils résident à la frontière floue entre système langage et environement. Leur philosophie et de mettre à disposition un 'outil' sous la forme d'un environement intéractif, où les utilisateurs ne se voient pas consciemment en train de programmer, jusqu'au point où eventuellement leurs besoins deviennent suffisemment spécifiques et sophistiqués pour que les fonctionnalités fournies ne suffisent plus et qu'ils aient alors à leur disposition un langage et un système suffisamment expressifs pour les satisfaire.

2.1 Objectifs sur le long terme

Le but final est pour Zébrys de parvenir à concevoir R++, qui se veut être un langage capable d'assurer au mieux la rétrocompatibilité avec la base de code R existant à ce jour tout en apportant la garantie d'un typage statique sans être intrusif et des performances supéreures à celles que peut connaitre un programme R à l'heure actuelle. Le problème se décompose alors sous plusieurs aspects :

— Définir la syntaxe de R++
— Définir une sémantique pour R++
— Définir un système de type disposant d'une inférence de type décidable
— Compiler efficacement R++

Ces trois points n'étant bien entendu pas indépendants les uns des autres, avec la contrainte supplémentaire qu'il n'est pas raisonnable de s'éloigner dramatiquement de R, de sa syntaxe et de sa sémantique.

Une fois ce constat réalisé, et puisqu'on se pose la question de la rétrocompatibilité, il convient de se poser les questions suivantes :

— Qu'est-ce-que la syntaxe de R au juste ?
— Quelle est sa sémantique ?
— Quel sous-ensemble de R est raisonnablement typable ?
— Peut-ont raisonnablement ne pas supporter certaines features du langage ?
— Peut-on compiler efficacement R, si non, quels sont les freins à une telle compilation ?

2.2 Conserver la syntaxe

Dans un premier temps, il est nécessaire d'avoir une idée de ce qu'est la syntaxe de R. Il n'existe malhereusement que peu voir pas de documentation sur le sujet. Le langage ne s'est à priori jamais doté d'une grammaire au format BNF et l'analyseur lexical et syntaxique de l'implémentation open source de R est pour le moins rudimentaire, au sens où les règles de productions de la grammaire sont des plus ambiguë et seul l'analyse lexicale dépendante du contexte parvient à y donner du sens. Les premières lignes correspondent à des instructions visant à taire les conflits décalage/réduction et réduction/réduction inherant à la grammaire employée. Il est donc difficile dans ces conditions de disposer d'informations suffisantes sur la syntaxe exacte de R et on en est donc réduit à une suite d'essais/erreurs pour parvenir à déterminer la grammaire reconnue.

2.3 Conserver la sémantique

La sémantique de R est pour le moins perturbante, le langage dispose en effet d'un ensemble de features pour le moins étranges, non pas qu'elles le soient indépendemmant les unes des autres mais il est rare de les voir combiner ensemble, de manière non exhaustive :

- Les affectations sont des expressions comme les autres et sont des effets de bord dans le scope courant.
- Les arguments d'une application sont passés sous forme de promesses contennant une référence vers leur environement de définition et l'objet syntaxique de l'expression passée en argument.
- Il coexiste plusieurs systèmes de programmation orientée objet pour le moins insolites avec une résolution des appels de 'méthodes' peu orthodoxe.
- Les propriétés refléxives du langage semble ne connaitrent aucune limite et permettent par exemple de modifier intégralement le comportement d'un fonction déja définie, aussi bien ses arguments, que le corps de la fonction ou son environement d'execution.

2.4 Des chiffres pour décider

On est assez vite amené face aux bizarreries du langage à se demander si tel ou tel element syntaxique ou sémantique est vraiment employé dans le code R ou si il s'agit uniquemment d'un artefact qui aurait été conservé mais jamais utilisé. Il est alors intéressant pour pouvoir trancher sur la question de disposer d'une analyse de la base de code disponible sur le CRAN quand cela s'avère pertinent. On a à disposition quelques 10 000 packages au travers desquels on peut espérer parvenir à faire le tour de toutes les subtilités effectivement utilisées de R ne serait-ce que syntaxiquement mais aussi parfois sémantiquement quand une analyse statique est en mesure d'y répondre.

2.5 Typage statique d'un sous ensemble de R

On se pose au final la question de parvenir à déterminer un sous-ensemble de R que l'on serait en mesure de typer statiquement, en gardant à l'esprit qu'un tel système de type devrait disposer d'une inférence totale et décidable.

2.6 Langage et outils

Zébrys emploie exclusivement le C++ pour ses besoins de développement, il a donc été nécessaire de s'adapter à ce cadre de travail et la quasi totalité du code réalisé est en C++11. Ce qui impose comme nous allons le voir des contraintes fortes sur ceraines des solutions implémentées. L'analyse lexicale et syntaxique a été réalisée à l'aide de flex et bison.

Chapitre 3

Arbre de syntaxe abstrait

Que l'on cherche à compiler un programme, à l'interpréter, à la typer, ou à effectuer quelque opération que ce soit dessus, la première étape consiste à disposer d'une structure de données adaptée. Pour cela l'implémentation d'un arbre de syntaxe abstrait est la solution la plus adéquate. Selon le langage dans lequel on cherche à réaliser cette solution on a plus ou moins de facilités à obtenir une représentation compacte et extensible.

Un AST est un DAG dont les noeuds sont des structures de données spécialisées en fonction de l'élément qu'ils représentent, et les langages disposant de structures de données et de structures de contrôle adaptées à un raisonnement par cas permettent d'obtenir des solutions compactes pour de telles implémentations et leur manipulation, que ce soit par exemple OCaml et les types algébriques ou Scala et les case class, associés au pattern matching.

La premier jet d'AST reposait sur une hiérachie de classes implémentant pour chaque opération à réaliser une methode virtuelle redéfinie dans les classes filles. Une telle implémentation a le défaut de ne pas respecter le principe de séparation des préoccupations, nuisant à la modularité, l'extensibilité et la capacité à raisonner sur le code de manière générale.

3.1 Des Arbres en POO

C++ ne dispose pas des mêmes outils que Scala ou OCaml. Les solutions efficaces à ce type de problème sont bien connues en POO et l'application du patron de conception visiteur permet d'obtenir un ersatz de pattern matching. On pourrait prendre le partie d'implémenter toute opération sur l'arbre comme des méthodes virtuelles de la classe mère redéfinies dans les classes filles, mais toute nouvelle opération nécessite alors d'ajouter de nouvelles méthodes dans toutes les classes de l'arbre. La solution du visiteur est à double tranchant, car une extension de la structure de données implique d'ajouter du code pour les nouveaux noeuds dans tous les visiteurs, mais face à ses avantages telle que la localité de la définition des opérations et l'indépendance des données vis-à-vis des opérations les concernant, cela semble bénin.

C++ étant C++, le volume de code prend des proportions inattendues pour des implémentations qui devraient être simple. Prenons pour exemple un simple langage permettant de réaliser des additions sur des entiers.

```
type t = Add of t * t | Int of int

let rec compute = function
  | Int i -> i
  | Add (e1, e2) -> compute e1 + compute e2
```

Le même langage en C++ prend une autre dimension.

```
class Visitor;

class Node {
  virtual ~Node(){}
  virtual void accept(Visitor*) = 0;
};

class Add : public Node {
public:
  Add(Node* e1, Node* e2) : e1(e1), e2(e2) {}
  ~Add() { delete(e1); delete(e2); }
  Node* e1;
  Node* e2;
  void accept(Visitor* v) override { v->visit(this); }
};

class Int : public Node {
public:
  Int(int i) : i(i){}
  ~Int(){}
  int i;
  void accept(Visitor* v) override { v->visit(this); }
};

class Visitor {
public:
  virtual void visit(Add*) = 0;
  virtual void visit(Int*) = 0;
};

class Compute : public Visitor {
public:
  void visit(Add* a) override {
    a->e1->accept(this);
    int r = result;
    a->e2->accept(this);
    result += r;
  }
  void visit(Int* n) override { result = n->i; }
  int result = 0;
};

int compute(Node* n) {
  Compute c;
  n->accept(&c);
  return c->result;
}
```

On est forcé à adopter une approche où des opérations qu'on voudrait voir comme une simple fonction doivent être encodées sous la forme d'un objet. Il s'agit peut-être d'un travers de ma part de vouloir voir une fonction là où un programmeur objet pur et dur verra un système d'objets communiquant par messages, mais je suis tenté de ne pas admettre que cette vision soit adaptée au problème dans le cas présent.

Chapitre 4

Analyse lexicale et syntaxique

Afin de reconnaître la syntaxe de R et construire l'AST représentant le programme donné en entrée, on produit un analyseur lexical et syntaxique. L'analyse lexicale correspond à une opération de transformation du programme vers une séquence de lexèmes qui sont ensuite interprétés par l'analyse syntaxique pour leur donné du sens. Pour ce faire on peut utiliser un générateur d'analyseur lexical et un générateur d'analyseur syntaxique, il est plus facile de considérer ces deux étapes comme distinctes et successives mais la réalité est tout autre puisque les deux agissent de concert pour reconnaître le langage qu'elles décrivent et comme nous allons le voir il va nous falloir communiquer des informations entre ces deux composants.

Mon expérience en la matière était limitée à l'utilisation d'un peu de camlp4 et surtout de OCamllex et Menhir que nous avons utilisé pour reconnaître des langages avec relativemment peu d'ambiguités, mis à part les problèmes usuels qu'on est amené à rencontrer dans un langage de programmation comme le 'dangling else' ou les problèmes liés à l'associativité et la priorité des opérateurs arithmétiques les un vis-à-vis des autres. Outre l'apprentissage des petites différences qui existent entre OCamllex/Mehnir et Flex/Bison, la définition exacte du langage à reconnaître reste une difficulté à part entière.

De prime abord, la syntaxe de R ne semblait pas devoir poser de problèmes plus complexe qu'un autre langage de programmation, mais quelques points en particulier amènent à une analyse plus difficile qu'on ne le penserait et demande de réaliser des opérations que je n'avais jamais eu à considérer auparavant.

Une partie de la difficulté provient du fait que j'ai tendance à penser, peut-être à tort, qu'il ne devrait pas exister de conflits dans l'analyseur final. L'implémentation 'officiel' de R dispose d'un analyseur lexical et syntaxique qui reconnait le langage en faisant le choix de donner la responsabilité entière à l'analyse lexicale de s'adapter au contexte. Je préfère prendre le partie de permettre à l'analyse syntaxique de réagir au contexte courant et de modifier le comportement de l'analyse lexicale pour répondre à ses besoins. L'analyse syntaxique, de par sa nature, décide en fonction du contexte de l'interprétation des lexèmes, en ce sens l'analyse syntaxique dispose de la définition du contexte courant et il me parait conceptuellement inapproprié de donner la responsabilité à l'analyseur lexical de modifier sans information de l'analyse syntaxique sa stratégie de production des lexèmes quand cela dépend du contexte.

4.1 Tout est expression

La majorité des langages de programmation s'accordent à disposer d'une syntaxe telle qu'un programme est une séquence d'instructions formées d'expressions.

```
let x = 42
(*erreur*)
let x = let x = 42
```

```
int x = 42;
//erreur
int x = int x = 42;
```

R n'est pas du meme avis, et il n'existe tout simplement pas d'élément syntaxique qui ne soit pas une expression.

```
x <- x <- 42
```

Cela ne serait pas un problème en soit si il n'existait pas certaines constructions que nous allons voir par la suite.

4.2 Les bizarreries syntaxiques

Un des problèmes de la syntaxe de R est ce que l'on peut trouver comme cible d'une affectation. Les affectations étant des expressions et ne disposant pas du moindre lexème pour les préfixer cela entraine quelques difficultés.

```
f(x) #correct
f(x) <- 3 #correct
(function (x) x)(x) #correct
(function (x) x)(x) <- 3 #erreur de syntaxe
```

On est autorisé à faire apparaitre à gauche de l'affectation certaines constructions des expressions mais pas d'autres, et il nous faut alors différencier au sein des expressions entre les formes acceptées dans ce contexte et celles rejetées. Le contexte en question n'est connu qu'au moment ou le symbole <- devient le lexème de prédiction puisque les affectations ne sont pas préfixées par un lexème à part entière, et rien ne permet de savoir qu'on est dans une affectation et non pas dans une simple expression avant le symbole <-. Pour ce genre de cas on ne peut pas n'avoir qu'une seule règle de production des applications mais il nous en faut deux, l'une spécifiant les applications autorisées dans ce contexte et les autres, et considérer les deux règles de productions dans les contextes 'normaux'. Le problème ne se limite pas aux applications et toute la grammaire des expressions doit etre divisée entre les formes autorisées à gauche d'une affectation et les autres.

```
#string comme identificateur de variable
"abc" <- "abc"

x <- if(TRUE) x <- if(TRUE) x <- 3
#arbre identique
x <- (
    if(TRUE) {
        x <- (
            if(TRUE) {
                x <- 3
            })})
```

4.3 Des retours à la ligne

La syntaxe de R utilise le retour à la ligne pour déterminer la fin d'une expression, d'une manière à première vue similaire à ce qu'on peut trouver dans d'autres langages de script comme PYTHON

```
#expression x + 2
x + 2

#expression x suivie de + 2
x
+ 2
```

Le problème est que ce n'est pas toujours le cas. Dans certains contextes le retour à la ligne est autorisé sans qu'il ne signifie le fin de l'expression.

```
#expression x + 2
x +
2
```

On ne peut pas considérer les retours à la ligne comme du bruit puisqu'il servent de marqueurs de fin d'expression mais on doit pouvoir etre capable de reconnaitre la construction ci-dessus.

Une première solution que j'ai considérée était de chercher à rebondir sur une erreur de l'analyseur en rencontrant un lexème de retour à la ligne et de tenter de continuer comme si le retour à la ligne n'avait jamais été présent, mais une telle solution bien qu'elle montre des résultats n'est pas satisfaisante et est difficile à mettre en place sur la totalité du langage sans produire des conflits.

La deuxième solution que j'ai retenue est d'embrasser le problème et de spécifier dans toutes les règles de production où il est autorisé de voir apparaître un nombre arbitraire de retours à la ligne.

EXPR OP EXPR → EXPR OP EOL* EXPR

Mais le problème se corse face à des constructions syntaxiques similaires mais n'obéissant plus aux mêmes règles.

```
#expression x + 2
(x
+
2)

#une application
f(
x
,
y
)
```

Puisqu'une seule et unique expression peut apparaître entre parenthèses, l'utilisation du retour à la ligne pour marquer la fin d'une expression n'est plus nécessaire. Dans le cas d'une application 'f(' est suffisant pour déterminer que l'expression à une suite et comme les arguments sont séparés par des ',' on peut reconnaitre une application sans ambiguité si on ne considère plus les retours à la ligne comme un lexème. Tout contexte dans lequel une expression dispose d'un délimiteur de fin d'expression autre qu'un retour à la ligne obéit à la meme règle.

Il faut donc en fonction du contexte dans lequel se trouve l'analyse syntaxique considérer ou non les retours à la ligne dans l'analyse lexicale. On peut pour cela employer des actions à réaliser dans les règles de production elle-même en recontrant un lexème de prédiction plutôt qu'uniquement à la fin au moment d'effectuer une réduction. De cette manière on peut maintenir un compteur de la profondeur des parenthèses rencontrées. En rencontrant un retour à la ligne l'analyseur lexical teste la valeur courante du compteur pour savoir si il doit ou non produire un lexème.

Il faut alors considérer les accolades. En R les accolades délimitent une séquence d'expressions séparées par des point virgules ou des retours à la lignes. Cette séquence forme une expression à part entière qui peut donc apparaitre entre parenthèses.

```
#expression x suivie de + 2
({x
+
2})

#expressions identiques
x
+
2
```

Il faut donc remettre le compteur à 0 en rencontrant l'accolade ouvrante et restituer la valeur du compteur en rencontrant l'accolade fermante. En associant au lexème de l'accolade ouvrante la valeur du compteur on peut la retrouver à la fermeture d'accolade correspondante au cours de l'analyse syntaxique.

4.4 Des ELSE et des retours à la ligne

Face aux retours à la ligne comme délimiteurs d'expressions on rencontre un autre problème provennant de l'intéraction de ce choix avec les if then else.

```
{
    if (TRUE) TRUE
    else TRUE
}
```

'if (TRUE) TRUE' correspond à une expression syntaxiquement correcte, elle est suivie d'un retour à la ligne et on serait donc autorisé à appliquer une réduction. Dans ce cas la ligne suivante est une erreur, 'else TRUE' n'est pas une expression syntaxiquement correcte. Si la syntaxe de R avait fait ce choix il n'y aurait pas de problème, mais la syntaxe de R ne fait pas ce choix. Cela ne doit pas etre analysé comme 'if (TRUE) TRUE' suivie de l'expression incorrecte 'else TRUE' mais comme 'if(TRUE) TRUE else TRUE', qu'importe le nombre de retours à la ligne séparant les deux. Pour cela on aurait besoin d'un nombre indeterminé de lexèmes de prédiction pour pouvoir choisir la règle à appliquer en fonction de la présence d'un ELSE ou non. Nous générons un analyseur de type LALR(1), '$\forall$ k $\in$ N. LALR(k)' n'est pas une option, on pourrait par contre se diriger vers un parser GLR capable de résoudre ce type grammaire mais on va s'en tenir à LALR(1) et trouver une méthode pour 'tricher'.

La solution que j'ai retenue est d'aborder le problème du point de vue de l'analyse lexicale pour ne pas avoir à considérer le '$\forall$ k $\in$ N, k retours à la ligne précédant un else' au cours de l'analyse syntaxique.

En considérant que toute séquence de retours à la ligne suffixée d'un else qui n'est pas suivi d'un caractère pouvant apparaître dans un identificateur correspond au lexème ELSE on se sépare de ce problème. La contrainte sur ce qui suit le else est nécessaire car sinon on risque de couper un identificateur préfixé par else en deux comme par exemple elsex.

Le reste du langage ne pose pas de problème particulièrement notable et je n'en parlerais donc pas. Il est a noté néanmoins qu'il est compliqué de s'assurer que le langage reconnu est le bon, et seul l'analyse statique réalisée par la suite permet de penser que l'analyseur produit est correct ou au moins reconnait un langage dont l'intersection avec R semble satisfaisante.

Chapitre 5

Analyse statique

Une fois équipé de la structure de l'AST et d'un analyseur lexical et syntaxique capable de reconnaitre le langage et d'en déduire une représentation sous la forme du dit AST, on peut chercher à implémenter un outil d'analyse statique de programmes écrits en R.

L'interet est multiple, d'une part on aimerait bien disposer d'informations sur ce qu'il est possible de trouver dans les programmes R de la communauté, d'autre part en exposant l'analyseur lexical et syntaxique à autant de styles de programmation en R que possible on peut rencontrer des constructions syntaxiques qu'on aurait pas traitées et ainsi l'améliorer

Dans cette optique l'objectif était de développer un outil d'analyse statique s'appuyant sur les travaux précédents qu'on puisse facilemment étendre, et de le faire tourner sur les 10 000 packages disponibles via le CRAN.

Le problème se décompose de la manière suivante :

— Comment récuperer les packages et les sources qu'ils contiennent automatiquement depuis le CRAN ?
— Etant donnée une liste de fichiers et un ensemble d'analyses à réaliser quelle représentation des résultats choisir ?
— Quelle architecture est pertinente pour un tel outil ?

5.1 Récuperer les sources

Puisqu'on souhaite analyser les packages du CRAN il faut être en mesure de les télécharger, les extraire et identifier les fichiers sources qu'ils contiennent.

Le serveur du CRAN a à diposition des clients des urls accessibles pour télécharger via HTTP les packages disponibles. On a donc accès à un document HTML au formattage puremment utilitaire et facilemment traitable contenant toutes les informations dont on a besoins. On peut sans difficultés s'en tenir à un script bash qui télécharge le document HTML produit par le serveur Apache à l'aide de wget puis en extrait les liens de téléchargement à l'aide de grep. On a alors une liste complète des urls permettant de télécharger les archives, le noms des packages correspondants, la dernière date de mise à jour, etc. L'arborescence d'un package R est standardisée, une fois l'archive extraite, les sources sont accessibles au chemin "nompackage/R/unfichiersource.R". Encore une fois pas de difficultés pour déterminer l'ensemble des sources contenues dans une archive.

Une telle solution est efficace et rapide à mettre en place mais a la défaut de ne pas être portable, il peut être souhaitable de revenir sur cette partie ultérieurement pour en proposer une implémentation plus adaptée.

5.2 Représenter les résultats

Le choix d'une sortie sous forme de CSV a rapidemment semblé adéquat, le format est simple et a l'avantage de pouvoir être importé dans toute application tierce. On s'intéresse à des résultats par packages et non par fichier source et il faut donc être capable de fusionner les résultats d'une analyse sur un fichier A avec les résultats de la même analyse sur un fichier B. La première version que j'ai réalisée nécessitait de disposer d'une structure de CSV vide aux en-têtes prédéterminés à l'avance correspondant à chacune des analyses à effectuer qu'on remplissait avec les résultats.

Cela fonctionnait mais il était pénible de changer les analyses effectuées et en utilisant une structure de données partagée par toutes les analyses il est difficile de chercher à les paralléliser alors qu'elles se trouvent être conceptuellement indépendantes, si on cherche par exemple à compter le nombre d'occurences d'un certain élément dans un package, compter le nombre d'occurence dans le fichier A et dans le fichier B ne devrait pas

nous contraindre à adopter une solution séquentielle. En prenant un peu de recul sur le problème on parvient à obtenir une solution plus satisfaisante.

On a à disposition un certains nombres de fichiers sources en R appartenant à un même package et une batterie d'analyses à faire tourner dessus. Il est souhaitable par ailleurs de pouvoir facilemment ajouter ou enlever une analyse, le résultat ne devrait donc pas avoir une structure fixée à l'avance mais plutôt déduite des analyses effectuées.
On cherche donc à déterminer quel devrait être les propriétés d'un tel type de résultats. On adopte une vision telle qu'une analyse est une fonction dont le domaine est le type des AST et le codomaine est le type des résultats.

ANALYSE : AST $\rightarrow$ RESULTAT

On va alors pour N analyses différentes à effectuer disposer de N résultats qu'il va nous falloir recomposer, il nous faut donc une opération qu'on appellera $+_R$.

$+_R$: RESULTAT x RESULTAT $\rightarrow$ RESULTAT

Telle que si on dispose de A_1, ..., A_N de type ANALYSE :

$RESULTAT_{final}$ = A_1 (AST) $+_R$ ... $+_R$ A_N (AST)

De plus puisque nos analyses sont indépendantes les unes des autres, il serait souhaitable de disposer des propriétés suivantes sur leurs résultats et l'opération $+_R$:

$R_1 +_R R_2 = R_2 +_R R_1$

$R_1 +_R (R_2 +_R R_3) = (R_1 +_R R_2) +_R R_3$

Et en disposant d'un résultat vide R_e tel que :

R $+_R$ R_e = R

On décrit la structure d'un monoïde, et si le type de nos résultats équipé d'une fonction de composition respecte ces propriétés et si les analyses sont capables de produire d'elle meme le résultat qui leur correspond, il devient alors facile, non seulement de disposer de la possibilité d'ajouter ou de supprimer des analyses puisque le résultat final n'est obtenu qu'à partir de la recomposition des résultats de chaque analyse, mais aussi de paralléliser trivialemment les tâches effectuées puisque l'ordre dans lequel s'effectue la recomposition n'influe pas sur le résultat final.

Si on ne manipule que des valeurs dont les types disposent d'une opération de composition tels que ces propriétés sont respéctées, on peut obtenir un résultat sous forme de CSV à condition que les en-têtes de chaque analyse soient disjoints les uns des autres.

Par exemple si nos valeurs de résultats sont des entiers, prenons une analyse A et une analyse B à effectuer sur deux fichiers distincts F_1 et F_2. Si l'en-tete pour A est "A" et celui pour B est "B" et en choissisant de représenter une ligne de CSV comme une map des en-têtes vers les entiers. En disposant d'une fonction de fusion de deux map telle qu'en cas de collision entre deux clefs on additionne les valeurs correspondantes.

A(AST_{F_1}) = { ("A"$\rightarrow$ 2) }
A(AST_{F_2}) = { ("A"$\rightarrow$ 3) }
B(AST_{F_1}) = { ("B"$\rightarrow$ 0) }
B(AST_{F_2}) = { ("B"$\rightarrow$ 1) }
CSV_{final} = {("A" $\rightarrow$ 2 + 3), ("B" $\rightarrow$ 0 + 1)}

Quelque soit l'ordre dans lequel on effectue les analyses ou dans lequel on recompose les résultats on obtient le même résultat final, c'est cette propriété qui permet une parallélisation triviale des analyses.

Au final on n'exploitera qu'une parallélisation par fichier et non pas par analyse par fichier de manière à pouvoir partager des informations contextuelles entre les analyses d'un même fichier, telle que les variables libres ou liées à un certain point du programme.

5.3 Architecture

Il y a des opérations réccurentes dans toutes analyses statiques d'un programme, que ce soit des informations dont on a besoin dans des analyses différentes et qu'on aimerait ne pas avoir à recalculer mais qu'on souhaiterait partager, ou la traversée de l'arbre de syntaxe abstrait qui est à priori incontournable quoiqu'on souhaite réaliser.

On doit pouvoir réussir à factoriser les parcours de l'arbre induits par les différentes analyses, et séparer le contexte des analyses, les informations sur lesquelles elles se basent, de leur procédure de décision.

L'idée est donc la suivante, avoir un composant logiciel en charge de traverser l'arbre, un composant en charge de maintenir les informations sur le contexte courant, et un ensemble d'analyses qui se contentent de réagir en fonction des informations contextuelles et du noeud courant de la traversée.

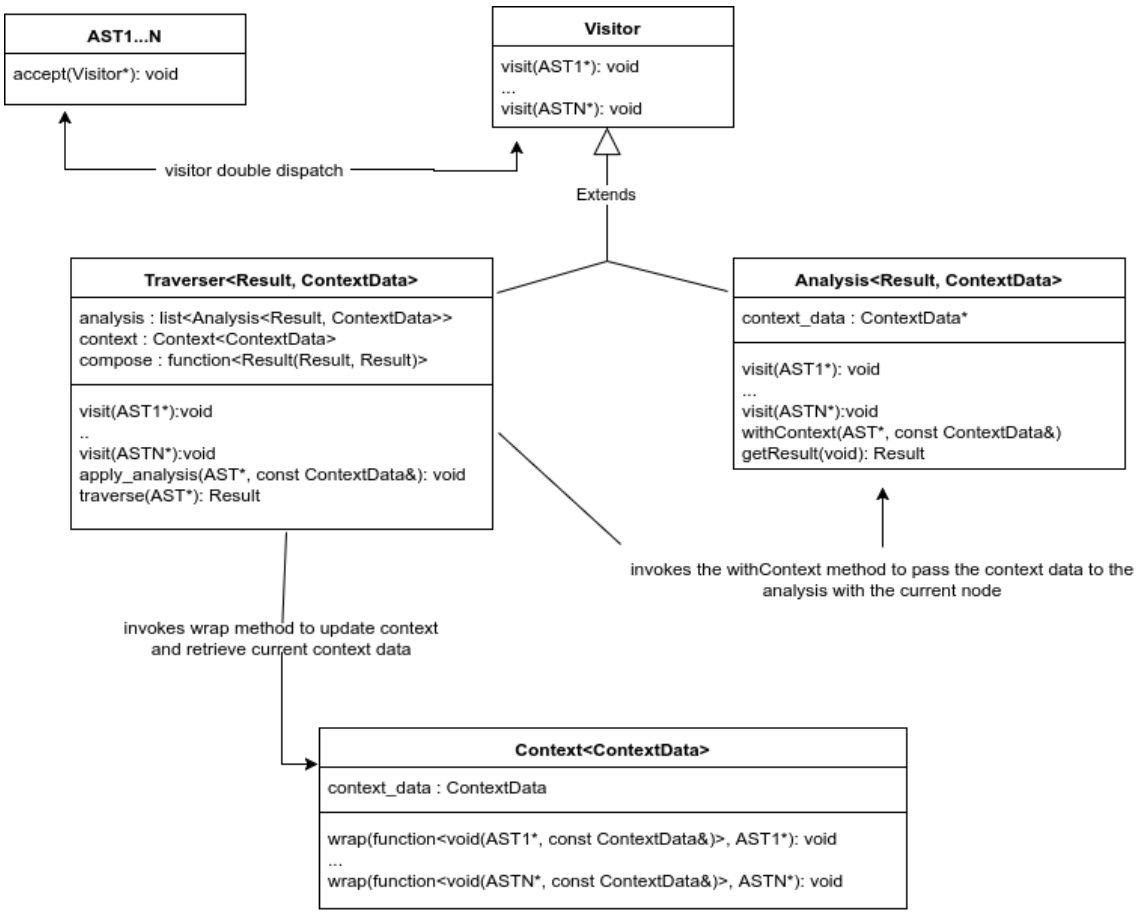

FIGURE 5.1: Architecture simplifiée

On va exploiter la pile des appels pour réagir à l'entrée et à la sortie d'un noeud au cours de la traversée. La traversée dispose à sa création d'un composant de gestion du contexte qui définit un ensemble de méthodes WRAP surchargées prenant en argument à la fois une fonction à appliquer sur un noeud et les informations de contexte, et un noeud, de cette manière le composant de gestion du contexte à la possibilité de mettre à jour les informations relatives au contexte courant avant et après l'appel de la fonction qu'on lui a passée. La traversée passe à WRAP une lambda dont le corps est l'application des analyses sur le noeud courant et la suite du parcours en profondeur de l'arbre.

On obtient une suite d'appels de la forme suivante :
— entrée dans TRAVERSER.VISIT
— entrée dans TRAVERSER.CONTEXT.WRAP
— possible mise à jour du contexte avant appel
— appel de la fonction passée en argument avec le noeud et le contexte courant
— possible mise à jour du contexte après appel
— retour de TRAVERSER.CONTEXT.WRAP
— retour de TRAVERSER.VISIT

On obtient ainsi un système ou toute opération de parcours de l'arbre est factorisée, le composant de contexte peut de manière indépendante de toute traversée prendre en charge le maintient d'information contextuelle, et les analyses ne décrivent qu'une procédure de décision en fonction d'informations contextuelles partagées sans dépendre directemment ni de la traversée ni du composant de gestion du contexte.

Cela améliore la relecture et la maintenance du code, on peut raisonner sous hypothèse de correction des autres composants. Il est facile de s'assurer que la traversée est correcte, que l'on pratique les bonnes opérations de mise à jour du contexte sous hypothèse que la traversée soit correcte, et que les analyses se comportent comme on le souhaite sous hypothèses que les information fournies soient les bonnes.

De plus l'implémentation d'une analyse supplémentaire est facilitée par le fait qu'on a juste besoin de développer les méthodes prenant en charge les noeuds pertinents, et déleguer au CONTEXT le maintient des informations additionnelles qui nous seraient nécessaires.

Il est alors possible de paralléliser les traversées des differents fichiers d'un package donné et de recomposer leurs résultats quand les analyses sont terminées.

5.4 Des statistiques sur les téléchargements

Une fois qu'on dispose de résultats sur un ensemble de packages R il est serait bon de pouvoir les mettre en perspective. Il est possible que le package truc s'amuse à redéfinir tous les opérateurs du langage, mais cela a peu d'importance si il n'a été téléchargé que 3 fois au cours des 5 dernières années, ce qui aurait tendance à indiquer que l'utilisation qu'il fait de R n'est pas très impactante.

Le CRAN met à disposition quotidiennement les logs du serveur de téléchargement sur son site. Ainsi d'une manière similaire à la façon dont on a procédé pour récupérer les packages on récupère ces logs, on les parse et on stocke les résultats dans une base de données. Le procédé est un peu long, chaque log faisant quelque 40 Mo, pour un an de données on doit télécharger 365 * 40 Mo = 14 Go de fichiers csv à traiter, bien que les informations qui nous sont utiles ne représentent qu'une fraction de cette quantité cela prend du temps. On obtient malgré tout des chiffres intéressants pour pondérer les résultats précédents.

5.5 Quelques chiffres

— FV : occurences de variables libres, sans doublons

— FV Capturée : affectations capturant la recherche d'une variable libre d'un scope inférieur, seule la première est comptée

— SB : occurences de super affectations explicites

— B : occurences d'affectations normales

— Downloads : nombre de téléchargements depuis le 01/01/2016

Package	FV	FV Capturée	SB	B	Downloads
Rcpp	1041	36	4	638	6 717 604
ggplot2	4739	193	3	2553	5 782 675
digest	239	1	3	121	4 900 199
stringr	281	10	0	128	4 858 895
stringi	1077	69	0	422	4 770 202
plyr	1114	24	14	557	4 540 625
jsonlite	796	20	3	243	4 381 448
nimble	15296	262	1231	7581	5 005

Le nombre de variables libres s'explique en partie par les dépendances à d'autres packages ou à la librairie standard, dans le cas d'une capture il n'est pas évident que cela est une incidence, il est tout à fait possible que la fonction où se trouve la variable libre ne soit plus jamais appelée à partir de ce point, cela peut aussi juste être symptomatique d'une dépendance mutuelle de deux définitions. Ces chiffres sont un échantillon de ce qui a été extrait des packages analysés, d'autres tests ont été effectués, allant de compter si on utilise plus souvent la flèche ou le égal pour produire une affectation à trouver si il existe des redéfinitions d'opérateurs ou des spécialisations des fonctions d'extractions. Il reste intéressant de chercher à étendre les tests réalisés par ces analyses.

Chapitre 6

Typage

R appartient à la famille des langages disposant d'un typage dynamique, et tout comme python ou javascript est amené au cours de l'execution d'un programme à tomber sur des erreurs de type irrécupérables entraînant un arrêt de l'execution. On cherche donc à définir un système de type qui nous permettent de déterminer statiquemment qu'un programme R ne sera pas amené à lever ces erreurs au moment de son execution. Bien entendu on ne s'attend pas à ce que l'ensemble des programmes R s'executant sans erreurs puissent être typés mais il est souhaitable de ne pas imposer de contraintes trop fortes sur les programmes considérés.

6.1 Préambule sur R

Avant toute discussion sur quelque système de type que ce soit, il est tout d'abord nécessaire de parler de la sémantique de R et des différents paradigmes qui y cohabitent, même si cela doit se faire de manière informelle et non exhaustive, cela fournit un tour d'horizon de quelques spécifités du langage et de ce à quoi on peut s'attendre face à du code R.

Au coeur de R il existe un noyau fonctionnel, le passage des arguments d'une fonction se fait toujours par valeur mais jamais par référence et le langage implémente un mechanisme de copy-on-write. Les fonctions sont des valeurs de première classe et en tant que telles peuvent être passées en argument ou servir de valeur de retour. R implémente un mechanisme de parresse limité aux arguments passés à l'appel d'une fonction dont la stratégie d'évaluation est un appel par nécessité. Il est important de noter qu'ici l'appel par nécessité n'est pas juste une optimisation dans l'implémentation d'un appel par nom, mais a un réel impact sur la sémantique générale du langage du fait du fonctionnement des environnements et du mechanisme de recherche associé.

R dispose égalemment d'une couche objet dans laquelle coexistent plusieurs systèmes différents qui se sont accumulés avec le temps, d'opération primitives fonctionnant de manière transparente à la fois sur des matrices, des vecteurs ou des listes, associées à des mechanismes de coercions implicites, et de fonctionnalités ayant trait à la programmation réflexive extrêmement développées.

6.1.1 Portée des variables, affectations et parresse

Bien que la portée des variables soit lexicale, toute variable libre d'une fonction ne voit pas sa valeur fixée par l'état de l'environnement au moment de la définition de cette fonction, mais par la valeur qui sera trouvée au moment de l'appel en cherchant à travers la hiérarchie des environnements englobant la définition, ce qui dépend donc des dernières affectations à avoir eu lieu dans ces environnements.

```
x <- "toto"
g <- function() {
    f <- function() x #x est libre
    print(f())        #affichera la valeur pour x visible au moment de cet appel
    x <- 1            #en appelant f on trouvera cette affectation
    return(f)
}

f <- g()              #affiche toto
x <- TRUE
f()                   #retourne 1
```

Ce comportement associé à la stratégie d'évaluation des arguments peut donner lieu à des situations plus ou moins complexes à appréhender.

```
f <- function (x) function (y) x
g <- f(x)       #x est inconnu mais on le passe sous forme de promesse
x <- 2          #x vaut 2
g(x <- "toto") #retourne 2
print(x)        #affiche 2 car la promesse x <- "toto" est ignoree
x <- TRUE       #la promesse ne sera pas recalculee
g(NULL)         #retourne toujours 2
```

Et la passage à l'ordre supérieur rend les choses d'autant plus complexes.

```
x <- "toto"
f <-  function(g) {
    h <- g(function () x)
    #selon la fonction g on trouvera soit "toto" maintenant
    #soit 3 plus tard  comme valeur pour x
    x <- 3
    return(h)
}
```

6.1.2 Affectation dans un scope supérieur

On n'a pas la possibilité de passer des arguments par référence mais on a à disposition un opérateur d'affectation particulier pour émuler une forme de mutabilité à travers les scopes.

```
f <- function() {
    x <- 1                #il existe un binding de x dans un scope superieur
    count <- function() {
        print(x)          #affiche le x visible
        #affecte x + 1 au premier binding de x visible depuis
        #le scope superieur
        x <<- x + 1
    }
}

count <- f() #on obtient un compteur
count()      #affiche 1
count()      #affiche 2
```

Il est à noter que si aucune association n'est trouvée pour x en remontant à travers les environnements, une nouvelle entrée apparait dans l'environnement global, donnant lieu à un mechanisme permettant de faire apparaitre de nouvelles définitions depuis un appel profond, ce qui peut s'avérer difficilement prévisible.

6.1.3 Récursion

Il n'existe pas de mot-clef pour la récursion qu'elle soit simple ou mutuelle, le phénomène se produit essentiellement du fait qu'au moment de l'appel il se trouve que la dernière définition visible pour l'appelant depuis cette position est la fonction recherchée.

```
odd <- function(x)
    if (x == 0) FALSE else even(x - 1)

even <- function(x)
    if (x == 0) TRUE else odd(x - 1)
```

Sans garantie sur le fait qu'une définition ultérieure ne vienne pas perturber la situation. Ce qui peut avoir ses avantages dans le cadre d'une session intéractive, une erreur dans la définition d'une de ces fonctions peut être corrigée en la redéfinissant sans redéfinir l'autre.

6.1.4 Arguments optionnels et appels de fonctions

Une fonction peut spécifier des expressions par défaut à associer à un ou plusieurs arguments qui ne seraient pas fournis au moment de l'appel. Ces expressions par défaut peuvent d'ailleurs se réveler dépendre de la valeur obtenue par l'évaluation de l'expression associée à un autre des arguments au moment de l'appel.

```
f <- function(x = y / 2, y = 2) x * y
#tous ces appels sont identiques et retournent 2
f()
f(1)
f(, 2)
f(, x = 1)
f(y = 2)
```

Au moment de l'appel d'une fonction les arguments fournis peuvent être associés aux paramètres de la fonction soit par nom soit par position, les deux pouvant être utilisés au sein du même appel. De plus, au moment de sa définition une fonction peut spécifier l'utilisation d'un nombre indeterminé d'arguments supplémentaire au moyen de l'ellipse noté '...'.

La résolution de l'association des arguments de l'appel avec les paramètres de la fonction se fait en séléctionnant tout d'abord les arguments qui ont été explicitement nommés puis les arguments restants sont associés par position avec les paramètres restants.

Les fonctions de la librairie standard de R font un usage extensif de ces arguments optionnels, cela s'explique en partie par la volonté du langage de vouloir faciliter l'accès aux fonctions de base en fournissant des fonctions dont on peut commencer par exploiter un ou deux arguments et en tirer un résultat satisfaisant dans la majorité des cas, puis pour remplir des besoins spécifiques être amenés à spécifier des valeurs autres que celles par défaut.

Pour certaines fonctions cela donne une documentation dense quand leur comportement est fortement paramétrable par des arguments qui décrivent parfois à eux tout seuls un mini langage. Exemple la fonction par [1] du package grachics.

1. https://www.rdocumentation.org/packages/graphics/versions/3.4.0/topics/par

6.1.5 Objet

Il existe au sein de R plusieurs systèmes ayant vocation à permettre une programmation orientée objet qui coexistent. Le système S3 qui exploite des fonctions dites 'génériques', sans définitions de classe, permet d'utiliser une forme de surcharge d'une fonction basée sur le type d'un des arguments.

```
print.A <- function(x) print("A")
print.B <- function(x) print("B")
x <- 42
print(x)        #affiche 42
class(x) <- "A"
print(x)        #affiche A
class(x) <- "B"
print(x)        #affiche B
```

Le système S4 dispose de définitions de classes et facilite la définition de fonction 'génériques' et permet une surcharge basée sur le type de plus d'un argument et un système d'héritage multiple.

```
setClass(
    "Person",
    slots = list(name = "character", age = "numeric")
)

setClass(
    "Employee",
    slots = list(boss = "Person"),
    contains = "Person"
)

#erreur, age attend une valeur de type numeric
alice <- new("Person", name = "Alice", age = "40")

alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss = alice)


setGeneric("isBossOf", function(x, y) standardGeneric("isBossOf"))

setMethod(
    "isBossOf",
    c(x = "Person", y = "Person"),
    function(x, y) {
        print("?")
        NULL
    }
)

setMethod(
    "isBossOf",
    c(x = "Person", y = "Employee"),
    function(x, y) y@boss@name == x@name
)

isBossOf(alice, john) # retourne TRUE
isBossOf(john, alice) # affiche '?' et retourne NULL
```

6.1.6 Réfléxivité

Les propriétés réfléxives du langage sont très puissantes, il est possible de manipuler toute portion du code comme une structure de donnée à partir de laquelle on peut produire du code executable et manipuler directemment les environnements d'execution.

```
add <- function (x, y) x + y          #une simple fonction
add(1,2)                              #retourne 3

formals(add) <- alist(x=0, y=0)       #modification des arguments
body(add) <- bquote(.(body(add)) + z) #on change le corps de la fonction
e = new.env()                         #on construit un nouvel environnement
assign("z", 42, e)
environment(add) = e                  #cet environnement est celui de la fonction add
add()                                 #retourne 42
```

6.1.7 Et si on pouvait tout changer

On notera la possibilité de redéfinir des éléments inattendus comme la parenthèse, le return, les opérateurs d'affectation et d'autres, qui s'avèrent n'être que du sucre syntaxique.

```
f <- function(x) return(x) + (2 + 2)

#retourne 2
f(2)

#on change les fonctions pour '(' et 'return'
'(' <- function(x) x + 2 - 2
'return' <- function(x) 7
#et pour +
'+' <- function(x, y) x * y

#retourne 42
f(pourquoipas)
```

6.2 Typage statique

On peut considérer plusieurs manières de chercher à typer R statiquemment, différentes approches mènent à des systèmes de types différents et imposent plus ou moins de contraintes sur la forme des programmes qu'on est capable de considérer.

A l'instar de typescript pour javascript on peut proposer un sur-ensemble du langage, augmenté d'annotations et de constructions syntaxiques fournies par l'utilisateur telles qu'on dispose d'informations supplémentaires sur les types pour déterminer qu'un programme est bien typé puis proposer une transformation de ce sur-ensemble vers le langage lui même. Une telle solution est efficace et permet d'étendre l'ensemble des programmes typables, mais elle a le désavantage d'imposer à ses utilisateurs de s'adapter à une nouvelle manière d'écrire leurs programmes et de requérir de leur part une transformation de leur base de code existante s'il souhaite bénéficier des avantages d'un typage statique.

Il est vraisemblable qu'à terme pour atteindre un ensemble raisonnable de ce qu'on est capable de typer une telle solution est à envisager, pour le moment on va s'intéresser à un sous-ensemble de R pour lequel on est capable de proposer un typage statique sans requérir d'informations de la part des utilisateurs, c'est-à-dire un sous-ensemble qui nous permette de définir un système de type disposant d'une inférence de type totale.

De plus il est nécessaire de parvenir à un système admettant un polymorphisme paramétrique, ce qui a l'avantage non négligeable de permettre la réutilisation d'une même définition dans des contextes différents grâce aux différentes instanciations possibles de son type.

Dans le cadre de ce stage je n'ai pas le temps de proposer un système de type qui couvre un ensemble suffisamment grand du langage pour dépasser le stade de jouet, ou de fournir les preuves associées à une telle entreprise, mais il met néanmoins en lumière certaines restrictions nécessaires qu'on se doit d'imposer sur le langage pour parvenir à nos fins si on ne souhaite pas voir les types devenir extrêmement compliqués ou se priver d'une inférence totale.

6.2.1 Sous-ensemble de R considéré

On va se limiter au noyau fonctionnel du langage augmenté des affectations. Comme on l'a vu précédemment, l'affectation de R n'a pas le même sens qu'un LetIn en ML et il est nécessaire de considérer sa sémantique particulière.

On n'autorise pas la redéfinition de certains éléments comme les parenthèses, les accolades ou les opérateurs d'affectation, et on ne gère pas non plus l'opération d'affectation hors du scope courant. On va également se limiter à des affectations qui ne peuvent pas être passées en argument d'un appel de fonction, bien qu'il soit certainement possible de les considérer. Je n'ai pas pris le temps de prendre en compte d'autre forme d'application que la plus simple et la plus classique, à savoir la passage de la totalité des arguments d'une fonction, de manière positionnelle, et sans valeurs par défaut pour les arguments dans la déclaration des fonctions.

Au final on s'intéresse donc à un mini langage disposant de traits fonctionnels, mais aussi d'une opération d'affectation introduisant une mutabilité locale des variables. Ce langage dispose d'une sémantique équivalente à celle de R.

$$e ::= \lambda(x,\ ...,\ x).\ e \mid e(e,\ ...,\ e) \mid e; e \mid x \leftarrow e$$

6.2.2 Intuitions

On est amené à chercher à donner un type à des fonction qui dispose de la forme suivante :

```
f <- function() x
```

Ces fonctions sont caractérisées par la présences de variables libres dont l'existence dans l'environnement et le type associé ne peuvent pas être déterminés au moment de la définition de la fonction mais ne seront connus qu'au moment de son application.

Dans ces conditions quel type donner à une telle fonction ? On a tout de même accès à un certain nombre d'informations par l'observation de sa définition.

- On sait qu'il s'agit d'une fonction :
  $f : ? \rightarrow ?$
- Son domaine est défini par le type de ses arguments, ici aucun :
  $f : () \rightarrow ?$
- Son codomaine est le type τ_x qu'on va trouver pour x a moment de son application, sans qu'on n'en sache plus sur la forme exacte qu'il doit respecter dans le cas présent :
  $f : () \rightarrow \tau_x$
- Cela est soumit à la condition qu'au moment de l'application de f on puisse trouver un x, sinon on aurait une erreur à l'execution, ce qui correspond au genre de comportement qu'on cherche à rejeter, ainsi il est nécessaire qu'il existe un tel x et quelque soit le type τ_x qu'on va lui trouver cela déterminera le type de f, on obtient l'implication suivante :
  $\forall \tau_x.(\exists x, x : \tau_x) \implies f : () \rightarrow \tau_x$

Ces fonctions ne disposent donc pas uniquemment d'un domaine et d'un codomaine mais aussi de contraintes sur l'état des environnements englobant leur définition lors de leur application et exprimant les conditions nécéssaires à leur bonne execution. La satisfaction de ces contraintes permet non seulement de garantir que certaines erreurs ne vont pas se produire, mais aussi de trouver les informations qui nous manque quand elle pourront être connues, par exemple ici le type de retour de f. En somme il existe des informations sur les types qu'on a immédiatement à disposition, et d'autres qui nous sont inconnues et dont la prise en compte doit être retardée pour décider de la forme exacte du type à un certain point du programme.

Il va nous falloir maintenir une information sur la profondeur de scope à laquelle un jugement de typage se place de manière à garder une trace de l'environnement responsable de la satisfaction de telles contraintes.

Il faut aussi prendre en compte la stratégie d'évaluation des appels de fonctions, on ne cherche pas à disposer de types tels qu'ils soient capables d'exprimer l'évaluation des promesses. On va plutôt chercher à s'abstraire

de l'évaluation des arguments en appel par nécessité en imposant que toute affectation à un symbole qui aurait été utilisé dans un argument ait un type qui soit compatible avec celui utilisé lors du passage de l'argument.

Pour disposer de polymoprhisme on va s'autoriser à généraliser sur le variables de type et de contraintes. Bien qu'on ne puisse pas accéder aux environnements des scopes supérieurs au cours de la dérivation de typage, les contraintes peuvent spécifier les instances de type nécessaires à profondeur n, de telle sorte qu'au moment où leur résolution est nécessaire on peut vérifier que le schéma de type dont on dispose peut s'instancier de manière à satisfaire ces contraintes.

Il existe des limitations dans ce qu'on est capable d'exprimer avec nos types. Si on dispose de la fonction suivante : $\lambda().\ (f(\lambda().x);\ x \leftarrow 2)$. On ne sait pas à priori quel va être le type de f et si f va appliquer la fonction passée en argument ou non. En cas d'application on s'attend à disposer d'un x dans le scope supérieur, dans le cas contraire c'est l'affectation suivante qui va capturer toute recherche de x. On pourrait chercher à se prémunir de ce genre d'indécision en demandant que quoiqu'il arrive on dispose de x et que ce soit un entier en donnant le type suivant à ce programme : $\forall c_f, c.() \rightarrow_{[\exists f:_0((()\rightarrow_{[c]} int)\rightarrow_{[c_f]} int\ \wedge\ c_f\ \wedge\ \exists x:_0 int]}\ int$ mais cela complique d'autant plus les règles de typages et on va prendre le partie de chercher à rejeter de tels programmes bien qu'il soit certainemment possible d'obtenir une meilleure solution.

L'implémentation de l'inférence de type dispose d'une solution pour prendre en compte les fonctions récursives que je pense correcte mais que je ne n'ai pas réussi à exprimer dans le système présenté et qui n'y apparait donc pas. En l'occurence, en présence de récursion on ne dispose pas de contraintes récursives dont on aurait besoin et on est donc amené à chercher à instancier un schéma de type vers un type de profondeur infinie.

Seules les fonctions disposent d'un scope à part entière et la fermeture du corps d'une fonction ferme ce scope. On gagne du coup une information importante qui est que toute contrainte portant sur ce scope doit être satisfaite, invalidée ou reportée vers le scope supérieur en fonction de l'état désormais final de l'environnment interne de la fonction.

6.2.3 Systême de type

Les jugements de typage sont de la forme suivante : $\Gamma, U, H \vdash_{b,n} e : \tau, \Gamma', U'$
- Γ est l'environnement de typage courant qui va se trouver enrichie par les affectation dans e en Γ'
- U est un ensemble d'association de symboles de variables x et de types τ qui correspond aux instances employées en position d'agruments qui va enrichie en U' par les occurences de symboles dans des arguments.
- H est l'ensemble des hypothèses sur les environnements englobant nécéssaires à la dérivation du jugement de typage.
- b est un marqueur pour déterminer que le jugement de typage porte sur une expression en position d'argument.
- n est la profondeur de scope courante à laquelle se place le jugement.
- e est l'expression à laquelle on cherche à donner le type τ.
- τ le type qu'on souhaite donner à e.

Un programme p de type τ est typé à partir du jugement $\emptyset, \emptyset, \emptyset \vdash_{false,0} p : \tau$

langage

$$e ::= x | exp \qquad exp ::= \lambda(x,\ ...,\ x).\ e \mid e(e,\ ...,\ e) \mid e; e \mid x \leftarrow e$$

Syntaxe des types, des contraintes et des environnements

$$\Gamma ::= x : \sigma,\ \Gamma\ |\alpha, \Gamma|\alpha_c, \Gamma|\ \emptyset \qquad H ::= \exists x :_n \tau,\ H \mid \alpha_c,\ H \mid \emptyset \qquad U ::= x : \tau,\ U \mid \emptyset$$

$$\sigma ::= \forall \alpha.\ \sigma\ |\forall \alpha_c.\ \sigma \mid \tau \qquad \tau ::= (\tau, ..., \tau) \rightarrow_{[c]} \tau \mid \alpha \qquad c ::= \exists x :_n \tau \mid c \wedge c \mid \alpha_c$$

Règles (1)

$$\textsc{Assign}\ \frac{\Gamma, U, H \vdash_{false,n} e : \sigma,\ \Gamma', U' \qquad \forall (x : \tau_x) \in U'.(x : \sigma,\ \Gamma'), \emptyset, H \vdash_{false,n} x : \tau_x, (x : \sigma,\ \Gamma'), \emptyset}{\Gamma, U, H \vdash_{false,n} x = e : \sigma, (x : \sigma,\ \Gamma'), U'}$$

$$\textsc{Seq}\ \frac{\Gamma, U, H \vdash_{b,n} e_1 : \tau_1, \Gamma_1, U_1 \qquad \Gamma_1, U_1, H \vdash_{b,n} e_2 : \tau_2, \Gamma_2, U_2}{\Gamma, U, H \vdash_{b,n} e_1; e_2 : \tau_2, \Gamma_2, U_2}$$

$$\textsc{Abs}\ \frac{\begin{array}{c}(x_0 : \tau_0, ..., x_n : \tau_n), \emptyset, H_f \vdash_{false,n+1} e : \tau_r, \Gamma_f, U_f \\ H_f \Rightarrow_n c_f \qquad \tau_{r'} = simpl_{n+1}(\Gamma_f \cup ftv(H_f) \cup fcv(H_f), \tau_r) \qquad WF((\tau_0, ..., \tau_n) \rightarrow_{[c_f]} \tau_{r'})\end{array}}{\Gamma, U, H \vdash_{b,n} \lambda(x_0, ..., x_n).e : (\tau_0, ..., \tau_n) \rightarrow_{[c_f]} \tau_{r'},\ \Gamma, U}$$

$$\textsc{App}\ \frac{\begin{array}{c}\Gamma, U, H \vdash_{b,n} e_f : (\tau_0, ..., \tau_m) \rightarrow_{[c_f]} \tau_r, \Gamma', U_f \qquad \Gamma', \emptyset, H \vdash_{true,n} e_0 : \tau_0, \Gamma', U_0 \\ ... \qquad \Gamma', \emptyset, H \vdash_{true,n} e_n : \tau_n, \Gamma', U_n \qquad H \vdash_n simpl_n(\Gamma' \cup ftv(H) \cup fcv(H), c_f)\end{array}}{\Gamma, U, H \vdash_{b,n} e_f(e_0, ..., e_n) : \tau_r,\ \Gamma', U_f \cup U_0 \cup ... \cup U_n}$$

$$\textsc{Var}\ \frac{}{\Gamma, U, H \vdash_{false,n} x : \Gamma(x), \Gamma, U} \qquad \textsc{Var}\ \frac{}{\Gamma, U, H \vdash_{true,n} x : \Gamma(x), \Gamma, (x : \Gamma(x)) \cup U}\ \#\forall(\Gamma(x)) = 0$$

$$\textsc{Var}\ \frac{}{\Gamma, U, H \vdash_{true,n} x : \Gamma(x), \Gamma, U}\ \#\forall(\Gamma(x)) > 0$$

$$\textsc{FreeVar}\ \frac{x \notin \Gamma \qquad H \vdash \exists x :_{n-1} \tau \qquad WF(\tau)}{\Gamma, U, H \vdash_{false,n} x : \tau, \Gamma, U} \qquad \textsc{FreeVar}\ \frac{x \notin \Gamma \qquad H \vdash \exists x :_{n-1} \tau \qquad WF(\tau)}{\Gamma, U, H \vdash_{true,n} x : \tau, \Gamma, (x : \tau) \cup U}$$

Règles (2)

$$\textsc{Gen}\ \frac{\Gamma, U, H \vdash_{b,n} e : \sigma, \Gamma', U' \qquad \alpha \notin ftv(\Gamma) \cup ftv(H)}{\Gamma, U, H \vdash_{b,n} e : \forall\alpha.\sigma, \Gamma', U'}$$

$$\textsc{GenC}\ \frac{\Gamma, U, H \vdash_{b,n} e : \sigma, \Gamma', U' \qquad \alpha_c \notin fcv(\Gamma) \cup fcv(H)}{\Gamma, U, H \vdash_{b,n} e : \forall\alpha_c.\sigma, \Gamma', U'}$$

$$\textsc{Inst}\ \frac{\Gamma, U, H \vdash_{false,n} e : \forall\alpha.\sigma,\ \Gamma', U'}{\Gamma, U, H \vdash_{false,n} e : [\alpha \Rightarrow \tau]\sigma,\ \Gamma', U'}$$

$$\textsc{InstVar}\ \frac{\Gamma, U, H \vdash_{true,n} x : \forall\alpha.\sigma,\ \Gamma, U}{\Gamma, U, H \vdash_{true,n} x : [\alpha \Rightarrow \tau]\sigma,\ \Gamma,\ (x : [\alpha \Rightarrow \tau]\sigma) \cup U}\ \#\forall(\sigma) = 0$$

$$\textsc{InstVar}\ \frac{\Gamma, U, H \vdash_{true,n} x : \forall\alpha.\sigma,\ \Gamma, U}{\Gamma, U, H \vdash_{true,n} x : [\alpha \Rightarrow \tau]\sigma,\ \Gamma, U}\ \#\forall(\sigma) > 0$$

$$\textsc{InstExp}\ \frac{\Gamma, U, H \vdash_{true,n} exp : \forall\alpha.\sigma,\ \Gamma, U'}{\Gamma, U, H \vdash_{true,n} exp : [\alpha \Rightarrow \tau]\sigma,\ \Gamma, U'}$$

$$\textsc{InstC}\ \frac{\Gamma, U, H \vdash_{false,n} e : \forall\alpha_c.\sigma, \Gamma', U'}{\Gamma, U, H \vdash_{false,n} e : [\alpha_c \Rightarrow c]\sigma, \Gamma', U'}$$

$$\textsc{InstCVar}\ \frac{\Gamma, U, H \vdash_{true,n} x : \forall\alpha_c.\sigma, \Gamma, U}{\Gamma, U, H \vdash_{true,n} x : [\alpha_c \Rightarrow c]\sigma, \Gamma,\ (x : [\alpha_c \Rightarrow c]\sigma) \cup U}\ \#\forall(\sigma) = 0$$

$$\textsc{InstCVar}\ \frac{\Gamma, U, H \vdash_{true,n} x : \forall\alpha_c.\sigma, \Gamma, U}{\Gamma, U, H \vdash_{true,n} x : [\alpha_c \Rightarrow c]\sigma, \Gamma, U}\ \#\forall(\sigma) > 0$$

$$\textsc{InstCExp}\ \frac{\Gamma, U, H \vdash_{true,n} exp : \forall\alpha_c.\sigma, \Gamma, U'}{\Gamma, U, H \vdash_{true,n} exp : [\alpha_c \Rightarrow c]\sigma, \Gamma, U'}$$

WF

$$\frac{}{WF(\alpha)} \qquad \frac{WF(\tau_r) \qquad WF(c) \qquad wf(\tau_0) \qquad ... \qquad wf(\tau_n)}{WF((\tau_0, ..., \tau_n) \rightarrow_{[c]} \tau_r)} \qquad \frac{}{WF(\alpha_c)}$$

$$\frac{WF(c_1) \qquad WF(c_2)}{WF(c_1 \wedge c_2)} \qquad \frac{wf(\tau)}{WF(\exists x : \tau)} \qquad \frac{}{wf(\alpha)} \qquad \frac{wf(\tau_0) \qquad ... \qquad wf(\tau_n) \qquad wf(\tau_r)}{wf((\tau_0, ..., \tau_n) \rightarrow_{[\alpha_c]} \tau_r)}$$

simpl

$$\frac{}{simpl_n(\Gamma, \alpha) \Rightarrow \alpha} \qquad \frac{}{simpl_n(\Gamma, \alpha_c) \Rightarrow \alpha_c}$$

$$\frac{simpl_n(\Gamma, \tau_r) \Rightarrow \tau_r' \qquad simpl_n(\Gamma, c) \Rightarrow c'}{simpl_n(\Gamma, (\tau_0, ..., \tau_n) \rightarrow_{[c]} \tau_r) \Rightarrow (\tau_0, ..., \tau_n) \rightarrow_{[c']} \tau_r'} \qquad \frac{simpl_n(\Gamma, c_1) \Rightarrow c_3 \qquad simpl_n(\Gamma, c_2) \Rightarrow c_4}{simpl_n(\Gamma, c_1 \wedge c_2) \Rightarrow c_3 \wedge c_4}$$

$$\frac{x \notin \Gamma}{simpl_n(\Gamma, \exists x :_n \tau) \Rightarrow \exists x :_{n-1} \tau} \qquad \frac{}{simpl_n(\Gamma, \exists x :_{n-1} \tau) \Rightarrow \exists x :_{n-1} \tau}$$

$$\frac{\Gamma, \emptyset, \emptyset \vdash_{false,n} x : \tau, \Gamma, \emptyset}{simpl_n(\Gamma, \exists x :_n \tau) \Rightarrow \alpha_c} \ \alpha_c \ chosen \ fresh$$

Transformation des hypothèses vers les contraintes

$$\frac{}{\emptyset \Rightarrow_n \alpha_c} \ \alpha_c \ chosen \ fresh \qquad \frac{H \Rightarrow_n c}{(\exists x :_n \tau), H \Rightarrow_n (\exists x :_n \tau) \wedge c} \qquad \frac{}{\alpha_c \Rightarrow_n \alpha_c}$$

Satisfaction des contraintes

$$\frac{\alpha_c \in H}{H \vdash_n \alpha_c} \ n > 0 \qquad \frac{}{H \vdash_0 \alpha_c} \qquad \frac{(\exists x :_n \tau) \in H}{H \vdash_n \exists x :_n \tau} \qquad \frac{H \vdash_n c_1 \qquad H \vdash_n c_2}{H \vdash_n c_1 \wedge c_2}$$

6.2.4 Récursion

Le problème face à des fonctions récursives et mutuellement récursives est que comme pour le reste la récursion n'apparait qu'au dernier moment à l'application. Prenons le programme suivant :
$f \leftarrow \lambda(x).f(x)$;
$f(2)$
On va inférer le type suivant pour f : $\forall \alpha, \beta, c_f.(\alpha) \rightarrow_{[\exists f :_0 (\alpha) \rightarrow_{[c_f]} \beta \wedge c_f]} \beta$
Puis au moment de l'application on va instancier ce type. Mais pour satisfaire l'existence de f, et la contrainte qu'on va en déduire il nous faut satisfaire à nouveau l'existence de f de la contrainte qui en est déduite, etc... Il faudrait exprimer explicitement dans le système de types l'existence de telle contraintes récursive au moment d'une instanciation. L'inférence de types se contente de maintenir une liste des contraintes qu'elle cherche à satisfaire et si en cherchant à satisfaire qu'il existe bien un f d'un certain type τ_f à profondeur n on rencontre une contrainte qu'il doit exister un f d'un type τ_f' à profondeur n, on s'autorise à ignorer cette contrainte sous condition que $\tau_f = \tau_f$' modulo les contraintes pour reconstruire une récursion monomorphe. On s'autorise à une égalité ne considérant pas les contraintes car on sait qu'on ne peut pas trouver un autre f à profondeur n et donc si cette égalité est vraie les contraintes sont identiques.

6.2.5 Inférence

Plusieurs solutions ont été implémentées pour obtenir une inférence de type qui reste relativement performante. L'inférence de type se décompose en deux étapes, une traversée du programme considéré pour produire un système de contraintes dont la satisfaction doit équivaloir au bon typage de ce programme puis la résolution de ce système à l'aide d'une unification syntaxique reposant sur une structure de données union-find. On utilise également le système de rang/nivau tel que décrit ici :
`http://okmij.org/ftp/ML/generalization.html`
De cette manière la généralisation n'implique pas un parcours de l'environnement pour trouver les occurences libres d'une variable.

Contraintes

$$C ::= Eq(\tau,\tau) \mid Inst(\tau,\alpha) \mid Solve(c,\Gamma) \mid Simpl(\tau,\alpha,\Gamma,n) \mid SimplC(c,\alpha_c,\Gamma,n) \mid WellFormed(\tau)$$

Génération de contraintes

$$e ::= x|\lambda(x,\ ...,\ x).\ e \mid e(e,\ ...,\ e) \mid e;e \mid x \leftarrow e$$

$$\frac{x \in \Gamma}{\Gamma,U,H,x,\alpha,false,n \rightarrow_{gen} Inst(\Gamma(x),\alpha),\Gamma,U,H}$$

$$\frac{x \notin \Gamma \qquad \alpha_2 = fresh\alpha(n-1)}{\Gamma,U,H,x,\alpha,false,n \rightarrow_{gen} Eq(\alpha,\alpha_2),\Gamma,U,(x,\alpha_2) \cup H}$$

$$\frac{x \in \Gamma}{\Gamma,U,H,x,\alpha,true,n \rightarrow_{gen} Inst(\Gamma(x),\alpha),\Gamma,(x,\alpha) \cup U,H}$$

$$\frac{x \notin \Gamma \qquad \alpha_2 = fresh\alpha(n-1)}{\Gamma,U,H,x,\alpha,true,n \rightarrow_{gen} Eq(\alpha,\alpha_2),\Gamma,(x,\alpha_2) \cup U,(x,\alpha_2) \cup H}$$

$$\frac{\Gamma,U,H,e_1,fresh\alpha(n),b,n \rightarrow_{gen} C_1,\Gamma_1,U_1,H_1 \qquad \Gamma_1,U_1,H_1,e_2,\alpha,b,n \rightarrow_{gen} C_2,\Gamma_2,U_2,H_2}{\Gamma,U,H,(e_1;e_2),\alpha,b,n \rightarrow_{gen} C_1;C_2,\Gamma_2,U_2,H_2}$$

$$\frac{\Gamma,U,H,e,\alpha,false,n \rightarrow_{gen} C,\Gamma',U',H' \qquad \forall(x,\alpha_x) \in U'.C = C;Inst(\alpha,\alpha');Eq(\alpha',\alpha_x)}{\Gamma,U,H,x \leftarrow e,\alpha,false,n \rightarrow_{gen} C,(x,\alpha).\Gamma',U',H'}$$

$$\frac{\Gamma,U,H,e_1,fresh\alpha(n),b,n \rightarrow_{gen} C_1,\Gamma_1,U_1,H_1 \qquad \Gamma_1,U_1,H_1,e_2,\alpha,b,n \rightarrow_{gen} C_2,\Gamma_2,U_2,H_2}{\Gamma,U,H,(e_1;e_2),\alpha,b,n \rightarrow_{gen} C_1;C_2,\Gamma_2,U_2,H_2}$$

$$\frac{\begin{array}{c}\alpha_r,\alpha_0,...,\alpha_n,\ chosen\ fresh\ with\ rank\ n \\ (x_0,\alpha_0;...;x_n,\alpha_n),\emptyset,\emptyset,e,\alpha_r,false,n+1 \rightarrow_{gen} C,\Gamma_f,U_f,H_f\end{array}}{\begin{array}{c}\Gamma,U,H,\lambda(x_0,...,x_n).e,\alpha,b,n \rightarrow_{gen} \\ C;Simpl(\alpha_r,\alpha_s,\Gamma_f,n+1);Eq(\alpha,(\alpha_0,...,\alpha_n) \rightarrow_{[H_f]} \alpha_s);WellFormed(\alpha),\Gamma,U,H\end{array}}$$

$$\frac{\begin{array}{c}\alpha_f,\alpha_0,...,\alpha_n,\alpha_c,\alpha_c'\ chosen\ fresh\ with\ rank\ n \\ \Gamma,U,H,e_f,\alpha_f,b,n \rightarrow_{gen} C_f,\Gamma',U',H' \qquad \Gamma',U',H',e_0,\alpha_0,true,n \rightarrow_{gen} C_0,\Gamma',U_0,H_0 \\ ... \qquad \Gamma',U_{n-1},H_{n-1},e_n,\alpha_n,true,n \rightarrow_{gen} C_n,\Gamma',U_n,H_n\end{array}}{\begin{array}{c}\Gamma,U,H,e_f(e_0,...,e_n),\alpha,b,n \rightarrow_{gen} \\ C_f;C_0;...;C_n;Eq((\alpha_0,...,\alpha_n) \rightarrow_{[\alpha_c]} \alpha,\alpha_f);SimplC(\alpha_c,\alpha_c',\Gamma',n),\Gamma',U_n,\alpha_c\text{'} \cup H_n\end{array}}\ n > 0$$

$$\frac{\begin{array}{c}\alpha_f,\alpha_0,...,\alpha_n,\alpha_c,\alpha_c'\ chosen\ fresh\ with\ rank\ 0 \\ \Gamma,U,H,e_f,\alpha_f,b,0 \rightarrow_{gen} C_f,\Gamma',U',H' \qquad \Gamma',U',H',e_0,\alpha_0,true,0 \rightarrow_{gen} C_0,\Gamma',U_0,H_0 \\ ... \qquad \Gamma',U_{n-1},H_{n-1},e_n,\alpha_n,true,n \rightarrow_{gen} C_n,\Gamma',U_n,H_n\end{array}}{\begin{array}{c}\Gamma,U,H,e_f(e_0,...,e_n),\alpha,b,0 \rightarrow_{gen} \\ C_f;C_0;...;C_n;Eq((\alpha_0,...,\alpha_n) \rightarrow_{[\alpha_c]} \alpha,\alpha_f);Solve(\alpha_c,\Gamma'),\Gamma',U_n,H_n\end{array}}\ n = 0$$

Réolution des contraintes simplifiée

$$Eq(\tau_1,\tau_2) \rightarrow \tau_1 =_\tau \tau_2 \qquad Inst(\tau_1,\alpha_2) \rightarrow \alpha_2 =_\tau instantiate(\tau_1,rank(\alpha_2))$$

$$Solve(c,\Gamma) \rightarrow contains\_only\_variables(simpl_0(c,\Gamma)) \qquad Simpl(\tau_1,\alpha_2,\Gamma,n) \rightarrow \alpha_2 =_\tau simpl_n(\tau_1,\Gamma)$$

$$SimplC(c,\alpha_c,\Gamma,n) \rightarrow \alpha_c =_c simpl_n(c,\Gamma) \qquad WellFormed(\tau) \rightarrow WF(\tau)$$

6.2.6 Améliorations possibles

Outre la simplification possible des règles, l'expression formelle de la résolution du typage des fonctions récursives, de vraies explications sur la résolution des contraintes dans l'inférence de type et l'extension du mini langage considéré vers un ensemble plus conséquent, j'ai quelques idées que j'aimerais explorer et qui pourraient s'avérer intéressantes.

Dans une première passe d'analyse sur le programme, si on réalisait par exemple une analyse de vivacité on pourrait parfois lever certaines contraintes comme celle sur les symboles passés en arguments, ou chercher à déterminer qu'une affectation sur un symbole donné est la dernière de son scope, ces informations peuvent aider à simplifier les problèmes posés.

Pour se rapprocher de R, il semble nécessaire de disposer de certaines règles de sous-typage, les fonctions et leurs arguments par défauts en bénéficieraient mais aussi les listes qui correspondent plus à des records dynamiques.

Sur la fin du stage on a cherché à typer certaines des fonctions de base au comportement particulier, que ce soit le plus ou la fonction c. Ces fonctions sont caractérisés par un type de retour qui dépend du type des arguments passés en entrée. Par exemple la fonction c qui est une fonction de concaténation, peut être appelée avec des types différents et un nombre d'arguments variable et retourne la borne supérieure des types passé en entrée selon un ordre précis. Cela amène à chercher à étendre les contraintes qu'on embarque dans les types des fonctions avec des prédicats supplémentaires.

```
v <- c(TRUE, 3, "4") #produit un vecteur de string
v <- c(v,v,NULL,v,v,NULL,v,v,NULL, NA) #les NULL ne sont pas pris en compte
```

Chapitre 7

Conclusion

Au cours de ce stage on a donc réalisé un analyseur lexical et syntaxique, un outil d'analyse statique des programmes et un système de types sur un sous-ensemble du langage R.

On a vu quelques uns des défis que représente la syntaxe de R du point de vue de son analyse, et le développement de ces éléments a mit en avant certains des problèmes inhérant à la sémantique particulière de R. La stratégie d'évaluation des arguments lors des appels de fonctions, combinée à l'aspect mutable des environnements imposent des dispositions particulières aussi bien du point de vue du système de types, qu'en ce qui concerne toute tentative de compilation efficace du langage qui doit prendre en compte ces éléments, ce qui représentent un frein certain à la performance.

La suite de ces travaux est de continuer à travailler sur le système de type et son inférence afin de pourvoir les preuves nécessaires et chercher à l'étendre vers un sous-ensemble plus large. Il semble qu'il puisse s'avérer intéressant d'en déterminer une version plus contraignante de telle sorte qu'elle permette de diriger une compilation performante du langage.

J'ai le projet de poursuivre dans l'exploration de ce sujet à travers une thèse en partenariat avec Zébrys et le CNAM dans le but de parvenir à des résultats plus probant sur les problèmes soulevés par le typage et la compilation d'un langage comme R.