Université Paul Sabatier
Master Informatique et Télécommunication
Parcours **S**ystèmes **I**nformatiques et **G**énie **L**ogiciel

Rapport de stage

Institut de Recherche en Informatique de Toulouse
Responsable de stage : Martin Strecker
Tuteur ENSEEIHT : Marc Pantel

Vérification de structures de pointeurs

Mathieu Giorgino

Résumé:

La vérification de programmes impératifs est souvent longue et difficile tandis qu'elle est souvent plus facile pour les programmes fonctionnels purs (*i.e.* sans effet de bords). Cependant, les programmes fonctionnels ne sont pas toujours aussi efficaces que leurs homologues impératifs utilisant des pointeurs.

Le but de ce stage était d'étudier la possibilité de générer des programmes impératifs efficaces à partir de programmes fonctionnels purs facilement vérifiables. Ce rapport présente cette étude et une solution générant du code Scala à partir de code Isabelle utilisant des monades.

Avril - Septembre 2008

Remerciements

Je remercie l'ensemble de l'équipe ACADIE qui m'a chaleureusement accueilli, et je tiens à remercier plus particulièrement Martin Strecker pour son soutien, la confiance qu'il m'a porté et ses nombreux conseils. C'est grâce à lui que j'ai découvert le domaine de la vérification et que je travaille dans cette équipe, et je pense qu'il aurait vraiment été dommage de rater cela.

Je remercie aussi ma famille et mes amis qui ont fait de moi de ce que je suis, pour le meilleur en tous cas.

Table des matières

Remerciements i

1 Introduction 1
1.1 Contexte . . . 1
1.2 Objectif . . . 2
1.3 Muabilité et partage . . . 2
1.4 Exemple . . . 3
1.5 Muabilité . . . 5
1.6 Méthodes formelles . . . 6
1.6.1 La preuve assistée . . . 6
1.6.2 Le Model-checking . . . 7
1.6.3 L'Interprétation abstraite . . . 7

2 Outils et Concepts 9
2.1 Isabelle . . . 9
2.1.1 Utilisation basique . . . 9
2.1.2 Syntaxe et transformations . . . 12
2.2 Scala . . . 12
2.2.1 "Case Classes" . . . 13
2.2.2 Attributs . . . 13
2.3 (O)Caml . . . 14
2.4 Les Monades . . . 14
2.4.1 Utilisation . . . 15

3 Des pointeurs certifiés 17
3.1 Syntaxe . . . 17
3.2 Introduction . . . 17
3.3 De Caml vers Scala . . . 19
3.3.1 Types . . . 19
3.3.2 Langage . . . 20
3.3.3 Résultat . . . 22
3.4 Introduction des monades . . . 24
3.5 Des monades en Isabelle . . . 25

3.6 D'Isabelle vers Scala 29
3.6.1 D'Isabelle vers OCaml 30
3.6.2 Règles 31

4 Conclusion **37**
4.1 Améliorations futures 37
4.1.1 Extension à un modèle orienté objet 37
4.1.2 Méta-modélisation 37
4.1.3 Génération de code à partir d'Isabelle 38
4.1.4 Certification de la traduction 38

Chapitre 1

Introduction

1.1 Contexte

J'ai effectué ce stage à l'IRIT[1], au sein de l'équipe ACADIE[2] et sous la direction de Martin Strecker, Maître de Conférences à l'Université Paul Sabatier.

J'avais déjà effectué deux stages pour cette équipe avec pour objectif la vérification de programmes impératifs. Ils consistaient tous deux à effectuer une vérification de transformations de graphes, en identifiant les structures de données avec pointeurs à des graphes et les programmes les manipulant à des transformations de ces graphes. Le premier stage nous a appris que l'absence de contrainte pour les graphes rendait la vérification difficile et le second que la vérification automatique de transformation de graphes très contraints[3] était trop complexe, aussi bien en temps qu'en espace mémoire.

Ce stage avait alors pour but le problème inverse – permettre de générer un programme impératif avec pointeurs efficace à partir d'un programme manipulant des structures de données arborescentes et vérifié à l'aide d'un assistant de preuve. Il s'insérait dans les projets TopCased[4], Spacify[5]et Geneauto[6].

[1]Institut de Recherche en Informatique de Toulouse

[2]Assistance à la Certification d'Applications DIstribuées et Embarquées

[3]Les graphes étaient en fait des arbres auxquels étaient ajoutés des arcs/pointeurs dépendant sémantiquement de la structure et des données de l'arbre sous-jacent. Nous utilisions alors PALE (Pointer Assertion Logic Engine).

[4]Projet français visant à fournir, à l'aide de la plateforme de développement Eclipse, des moyens d'analyse d'exigences, modélisation, simulation de modèles, implémentation, test, validation, rétro-ingénierie et gestion de projet.

[5]Projet initié par l'ANR (Agence Nationale de la Recherche) visant au développement d'un environnement de conception pour le logiciel embarqué critique.

[6]Projet européen visant la réalisation d'un générateur de code embarqué temps-réel optimisé et vérifié.

1.2 Objectif

La programmation purement fonctionnelle consiste en l'évaluation de fonctions plutôt qu'en des changements d'états. Elle n'autorise aucun effet de bord et permet ainsi d'évaluer les expressions dans un ordre quelconque et de permettre la transparence référentielle, *i.e.* le fait que chaque expression puisse être remplacée par son résultat sans changer le comportement du programme. De cette façon, les structures de données manipulées sont arborescentes et il est beaucoup plus facile de raisonner sur ces programmes et de prouver qu'ils sont corrects, par exemple à l'aide d'assistants de preuve, mais aussi de les compiler plus efficacement. En effet, grâce à la transparence référentielle, les compilateurs peuvent changer l'ordre d'évaluation des expressions (évaluation stricte, évaluation paresseuse, stratégie non déterministe) ou partager des données.

Cependant cette simplicité peut avoir un coût à l'exécution : les programmes sont quelques fois plus lents et plus gourmands en mémoire que leurs homologues impératifs, malgré les nombreuses optimisations apportées par les compilateurs.

De plus les habitudes font que les langages impératifs sont les plus utilisés et donc les plus optimisés et développés, et il est certain qu'en informatique comme ailleurs l'inertie est une des choses les plus difficiles à contrer.

L'objectif de ce stage était donc d'étudier la possibilité de traduire ces programmes fonctionnels facilement vérifiables en des programmes impératifs efficaces. Nous allons voir que cela serait possible en transformant des appels de fonctions en des déréférencements de pointeurs et en permettant de modifier certaines données, *i.e.* en rendant le partage plus explicite et en permettant la muabilité.

1.3 Muabilité et partage

Définitions

Pour éviter tout malentendu, rappelons la définition des termes muabilité et partage nécessaires à la suite :

Muabilité : Une donnée muable a la capacité de pouvoir être modifiée après sa création.

Partage : Le partage de données est le fait de partager, à l'aide de pointeurs, de même données dans différentes structures.

Bien que les définitions ne prêtent pas à confusion, ces concepts ne sont pas toujours clairs car applicables à n'importe quel langage permettant de manipuler des structures de données et donc aussi bien aux langages de haut-niveau qu'aux langages machine. Les langages de haut-niveau étant au final compilés, et cette compilation pouvant apporter des optimisations non forcément décidables par la sémantique du langage, des données non muables ou non partagées pourraient le devenir après compilation. Bien entendu, la préservation de la sémantique par ces transformations est normalement assurée par le compilateur.

Dans la suite, il ne sera donc fait mention de ces concepts qu'au niveau le plus haut des langages auxquels nous nous intéresserons, sauf indication contraire.

Relations

Partage et muabilité sont des concepts distincts mais pourtant fortement liés. A bas niveau ils servent tous deux, de façon totalement indépendante, à économiser l'espace en évitant les copies. Néanmoins, **à un niveau suffisamment élevé** pour que l'efficacité du code soit indépendante de son implémentation (ce qui serait souhaitable mais inexistant actuellement) ils sont indissociables. En effet, si l'efficacité du code n'importe pas, partager des données n'a aucun sens si l'on ne peut y apporter des modifications, car il est impossible de distinguer des données partagés de données non partagées si elles sont non modifiables. De même, il est impossible de distinguer une donnée modifiée d'une copie de celle-ci. Pourtant, la combinaison des deux est très utile car elle permet la création d'objets logiques, *i.e.* des objets uniques auxquels tout le monde peut avoir accès, et dont tout le monde peut modifier les propriétés.

1.4 Exemple

Pour pouvoir effectuer la traduction d'un programme fonctionnel vers un programme impératif, il nous faut d'abord étudier les différences entre ces deux paradigmes et nous commençons donc par l'analyse d'un même exemple de programme écrit dans ces deux-ci.

Ces deux instances de ce programme sont écrites en OCaml (*c.f.* section 2.3) et en Scala (*c.f.* section 2.2) dans les listings respectifs 1.1 et 1.2. Ce programme permet d'obtenir le type d'une expression à partir des types de ses sous-expressions. Il définit des expressions dans lesquelles il est possible d'effectuer des sommes entre des variables et des constantes entières ou booléennes (pour permettre des erreurs de typages). Nous en étudierons une version plus complète dans la section 3 et nous allons surtout nous concentrer ici sur les structures de données fonctionnelles et impératives représentées respectivement par les figures 1.1 et 1.2.

Sur ces représentations, la différence au niveau des structures de données est évidente : arborescentes pour le programme fonctionnel et sous forme de graphe(s) composé(s) d'arbres et de pointeurs additionnels (flèches pointillées) pour le programme impératif. Le programme impératif utilise le partage de données en considérant qu'une variable n'a qu'une seule identité, qu'elle soit dans une déclaration ou une expression. Les accès en lecture ou en écriture dans la version impérative sont alors plus rapides car utilisant directement des pointeurs à bas niveau plutôt qu'une forme explicite de référence, les noms de variables, qui nécessite l'appel d'une fonction (`get_tp`) là où le programme impératif utilise les possibilités de la mémoire à accéder directement à ses adresses.

Cependant, un détail pourrait ne pas vous avoir échappé : il est tout à fait possible d'effectuer ces modifications au niveau fonctionnel car le partage est pos-

Listing 1.1 – example.ml

```
0  type tp =
     | IntT
     | BoolT
     | Error

5  type decl = string * tp

   type expr =
     | Var of string
     | ConstI of int
10   | ConstB of bool
     | Add of expr * expr

   let rec tp_expr decls e =
     match e with
15   | Var vname  -> get_tp decls vname
     | ConstI v   -> IntT
     | ConstB v   -> BoolT
     | Add (l, r) ->
       let tpl = tp_expr env l in
20     let tpr = tp_expr env r in
       match (tpl, tpr) with
       | (IntT, IntT) -> IntT
       | _            -> Error
   (* get_tp decls vn : get the type
25    of variable vn in decls *)
```

Listing 1.2 – example.scala

```
abstract class Tp
  case object IntT extends Tp
  case object BoolT extends Tp
  case object Error extends Tp

class Decl(varName:String, tp:Tp)

class Expr
  case class Var(d:Decl) extends Expr
  case class ConstI(v:Int) extends Expr
  case class ConstB(v:Boolean) extends Expr
  case class Add(l:Expr, r:Expr) extends Expr

def tp_expr(e:Expr):Tp =
  e match {
  case Var(d)    => d.tp
  case ConstI(v) => IntT
  case ConstB(v) => BoolT
  case Add(l, r) =>
    val tpl = tp_expr(l);
    val tpr = tp_expr(r);
    (tpl, tpr) match {
      case (IntT, IntT) => IntT
      case _            => Error
    }
}
```

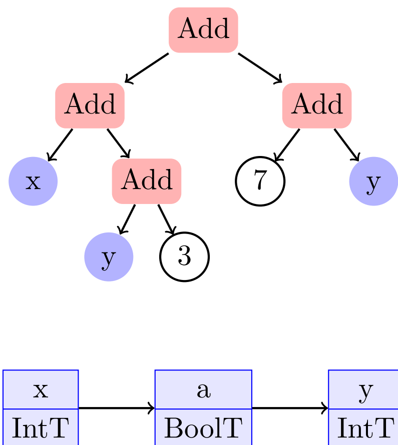

FIG. 1.1 – Structure de données fonctionnelle (arborescente)

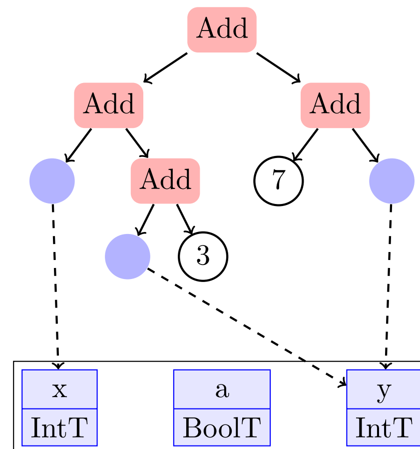

FIG. 1.2 – Structure de données impérative (noeud partagé)

sible. Il suffirait en effet de remplacer dans `Var` la chaîne de caractère par une déclaration et d'utiliser les déclarations pour initialiser l'expression. Mais il y a bien une raison pour l'avoir fait ainsi et la différence est ailleurs : la muabilité. Il suffirait en effet que l'on veuille pouvoir modifier les types des variables dans l'expression pour que le partage ne soit plus possible dans la version fonctionnelle.

De plus, dans les langages fonctionnels le partage n'est pas naturel : les données partagées ne peuvent être distinguées de celles qui ne le sont pas que par la façon dont est écrit le code. Il pourrait alors peut-être y avoir un moyen de rendre le partage des données plus explicite.

La différence étant la muabilité, nous allons alors l'étudier.

1.5 Muabilité

La question qui se posait était : Pourquoi les programmes fonctionnels sont-ils plus lents et plus gourmands en mémoire que les programmes impératifs ? En fait, cela peut s'expliquer par l'immuabilité des données propre au paradigme fonctionnel.

En effet cette propriété nécessite souvent la duplication de certaines parties des structures manipulées (par exemple lors de la concaténation de listes ou de l'insertion d'un noeud dans un arbre) pour conserver l'accès à la structure de départ. Pourtant cette structure de départ n'est souvent plus utilisée par la suite et la partie non partagée sera récupérée par le ramasse-miettes plutôt que d'avoir été modifiée directement.

La muabilité permet donc d'optimiser l'utilisation de la mémoire mais ce n'est pas tout : elle permet aussi un partage non plus seulement matériel/structurel mais aussi logique/sémantique. En effet, le fait de partager une donnée muable permet d'en faire une unité logique, un "objet" étant identique pour tous ceux l'utilisant et dont la modification est visible par tous.

Évidemment, l'immuabilité est l'essence même du paradigme fonctionnel et tenter d'y échapper conduirait à une remise en cause de tous les résultats obtenus pour celui-ci. Il y a cependant deux niveaux qui coexistent : le niveau du langage et de sa sémantique d'un coté, et le niveau exécutable généré par le compilateur de l'autre ; et respecter l'immuabilité au premier niveau ne force pas à la respecter au second, pourvu que la sémantique et donc le résultat soit préservé.

Il est donc possible de générer à partir de programmes purement fonctionnels, et en ajoutant si nécessaire des indications supplémentaires, des programmes de même sémantique et utilisant la muabilité. Il existe d'ailleurs certains langages qui permettent de gérer des données muables et des effets de bords de façon purement fonctionnelle :

Haskell utilisant les monades[1] issues de la théorie des catégories.

Clean, Mercury utilisant un système de type nommé "Unique types" ou types linéaires[2] issue de la logique linéaire.

Ainsi, un programme peut être vérifié à un niveau purement fonctionnel puis transformé/compilé en un programme impératif efficace. Il suffit alors que la transformation ait été démontré correcte pour que ce programme impératif soit certifié.

De plus, dans la sémantique des langages impératifs, la muabilité entraîne l'utilisation d'une notion d'état, qui n'est pas nécessaire pour les langages fonctionnels purs. Nous évitons alors d'avoir à l'utiliser lors de la vérification de programmes fonctionnels, ce qui conduit à ne plus considérer que des réductions d'expressions, d'où la simplification de la vérification.

La mise en oeuvre paraissant possible, nous nous y sommes attaqué.

1.6 Méthodes formelles

Ces méthodes, basées sur les mathématiques, aident à la spécification, au développement et à la vérification du logiciel et du matériel. Ces méthodes resteront pourtant marginales car considérées comme trop coûteuses jusqu'à l'explosion du vol inaugural de la fusée Ariane 5 en 1996 qui fera naître l'intérêt du monde industriel pour l'analyse statique.

1.6.1 La preuve assistée

La preuve assistée permet la vérification de spécifications formelles de systèmes.

Les assistants de preuve

Les assistants de preuve sont utilisés pour démontrer, interactivement ou non, des théorèmes. Ils se basent sur une logique permettant d'exprimer des théorèmes (hypothèses et résultat) qu'ils peuvent, soit résoudre complètement automatiquement, soit de manière assistée. Le choix de la logique est important selon les objectifs que l'assistant est censé atteindre. En effet, certaines logiques, comme l'arithmétique de Presburger, sont décidables et permettent donc de décider pour tout prédicat, s'il est vrai ou faux. Cependant, cela ne préjuge en rien de la complexité algorithmique nécessaire à cette décision, et celle-ci peut tout aussi bien être rédhibitoire.

La spécification formelle

La garantie de la qualité d'un système ne peut passer que par une spécification précise des besoins. Cependant, une spécification et une vérification manuelle ne sont que très rarement exhaustives et donc complètes. La spécification formelle permet de spécifier voire implémenter des systèmes de façon plus rigoureuses, sous forme de prédicats qui pourront ainsi être vérifiés par un assistant de preuve.

Les deux méthodes les plus utilisés industriellement sont :

la notation Z [3] qui est un langage de spécification utilisé pour décrire et modéliser les systèmes informatiques.

B [4] qui est à la fois une méthode de développement (la méthode B) et un langage formel de spécification.

1.6.2 Le Model-checking

Le Model Checking consiste à vérifier si un modèle donné, le système lui-même ou une abstraction du système, satisfait une propriété, appelée spécification et souvent formulée en termes de logique temporelle (CTL, LTL, ...). Il opère par analyse de l'ensemble des états atteignables du système, pouvant être vus sous forme de traces. Cette analyse peut se faire de trois façons différentes, ayant conduit à trois catégorie de méthodes différentes qui sont utilisés selon les besoins :

Les méthodes explicites qui représentent la totalité du système sous forme explicite, ce qui limite la taille du système pouvant être vérifié.

Les méthodes symboliques qui représentent le système sous forme abstraite, ce qui augmente les capacités de vérification à des systèmes de plus grande taille mais toujours limité.

Les méthodes bornées qui se limitent dans la longueurs des traces à analyser, ce qui permet de vérifier des systèmes de très grandes tailles mais de façon non exhaustive.

1.6.3 L'Interprétation abstraite

L'interprétation abstraite est une théorie uniformisant un grand nombre de méthodes permettant l'analyse statique de programmes. Ainsi, elle fournit en quelque sorte un moule pour les définir tout en les garantissant correctes.

Cette théorie se repose sur les treillis qui sont utilisés pour représenter les ensembles de valeurs que peuvent prendre les variables du programme. On peut ainsi rapidement vérifier des programmes sur des points précis tels que l'absence de dépassement de capacité ou de division par zéro, ou sur d'autres points plus généraux tels que la vivacité, la terminaison ou les relations entre variables.

Chapitre 2

Outils et Concepts

Ce chapitre est consacré à la présentation des outils ou concepts utilisés lors du stage et nécessaires à la compréhension de certains aspects de ce rapport.

2.1 Isabelle

Isabelle [5] est un assistant de preuve générique c'est à dire qu'il permet d'exprimer des formules mathématiques dans n'importe quelle théorie logique à l'aide de la méta-logique Isabelle/Pure. Il est surtout utilisé pour la formalisation de preuves mathématiques et en particulier la vérification formelle de programmes informatiques.

Nous avons choisi la distribution Isabelle/HOL ("Higher Order Logic") utilisant une logique classique du second ordre. C'est la plus développé actuellement, fournissant une librairie étendue de théorèmes mathématiques.

La suite de cette section présente le langage de façon succincte (2.1.1) ainsi qu'une fonctionalité intéressante utilisé par la suite : la définition de syntaxe et de transformations syntaxiques (2.1.2). Une documentation beaucoup plus complète dans la langue de Shakespeare peut-être trouvé sur son site [5].

2.1.1 Utilisation basique

Isabelle permet de déclarer ou de définir des fonctions et des types à l'aide d'un **langage fonctionnel** puis de vérifier des propriétés sur ceux-ci par l'écriture de **théorèmes**. Tout cela s'effectue à l'aide d'un langage textuel mais permettant l'affichage des symboles mathématiques tels que $\forall$, $\in$, $\Rightarrow$ ou λ. Effectivement les interfaces telles que Proof General (*c.f.* Fig. 2.1), une interface pour assistant de preuves basée sur Emacs, peuvent ensuite permettre l'affichage de ces symboles pour rendre la lecture des théorèmes facile.

Ces déclarations, définitions et preuves sont regroupées dans des modules appelés théories. Une théorie nommée `T` utilisant les théories `B1` ... `Bn` se présente de la façon suivante :

```
File  Edit  Options  Buffers  Tools  Isabelle  Proof-General  X-Symbol  Help
  assumes law2: "∀ a. a ⊵ returnM = a"
  assumes law3: "∀ x f g. (x ⊵ f) ⊵ g = x ⊵ (λ v. ((f v) ⊵ g))"
  assumes then_law: "∀ x y. x ⊳ y = x ⊵ (λ k. y)"

(**** The state transformer monad ****)

datatype ('v, 's) state_trans = ST "'s ⇒ 'v × 's"

types
  'n var_decls = "'n list"
  ('n, 'v) var_values = "'n ⇒ 'v option"
  ('n, 'v) state = "'n var_decls × ('n, 'v) var_values"

fun runST :: "('v, 's) state_trans ⇒ 's ⇒ 'v × 's"
where
  "runST (ST m) = m"

lemma [simp]: "∀ a. ST (runST a) = a"
apply (auto)
apply (case_tac a)
by(simp)

constdefs
  returnST :: "'v ⇒ ('v, 's) state_trans"
  "returnST m == ST (λ e. (m,e))"

constdefs
  bindST :: "('v1, 's) state_trans ⇒ ('v1 ⇒ ('v2, 's) state_trans) ⇒ (('v2, '
--:-- cell_trans_monad.thy    6% L30    (Isar script XS:isabelle/s Scripting)--------

proof (prove): step 2

goal (1 subgoal):
 1. ⋀a fun. a = ST fun ⟹ ST (runST a) = a

-u:-- *goals*     All L1     (proofstate)--------------------------------------------
```

FIG. 2.1 – Une théorie Isabelle/HOL dans Proof General

Listing 2.1 – Une théorie en Isabelle

```
theory T imports B1 ... Bn
begin
 Déclarations, définitions et preuves
end
```

On peut alors définir :

– des types :

```
types
  ('a, 'b) t = "'a list ⇒ ('b × c)"
  x = "(nat, bool) t"
```

– des types algébriques :

```
datatype 'a liste =
  Nil
| Cons "'a" "'a liste"
```

– des constantes :

```
constdefs
  ouex :: "bool ⇒ bool ⇒ bool" (infixr "⊕" 5)
  "a ⊕ b == (¬a ∧ b) ∨ (a ∧ ¬b)"

consts isempty :: "'a liste ⇒ bool"
primrec
  "isempty Nil == True"
  "isempty (Cons a as) == False"
```

– des fonctions :

```
fun f :: "nat ⇒ nat"
where
  "f 0 = 0"
| "f x = x + 1"
```

Les expressions utilisés dans les définitions sont semblables à celles des langages fonctionnels avec :

– des constantes :
  – entières : “0”, “1”, “92”
  – booléennes : “True”, “False”
  – ...
– des applications de fonctions/constructeurs, préfixées ou infixées : “max 1 2”, “1 + 3”, “Cons 1 (Cons 2 (Cons 3 Nil))”, “[1, 2, 3]”
– des définitions de variables : “let a = 5 in a + a”
– du filtrage par motif : “case xs of [] => True | x#xs' => False”

Nous avons donc pour le moment les possibilités d'un langage fonctionnel, auxquelles s'ajoute la définition de théorèmes ou de lemmes :

```
theorem f0 : "f(0) = 0"
```

```
lemma fcroissante : "∀ x y. x > y ⟶ f(x) > f(y)"
```

suivis de leurs démonstration avec :
- une seule étape :

```
by (simp)
```

- plusieurs étapes en une :

```
by (case_tac "x", simp_all)
```

- plusieurs étapes en... plusieurs étapes :

```
apply (case_tac "x")
apply (simp_all)
done
```

2.1.2 Syntaxe et transformations

Isabelle permet aussi de définir sa propre syntaxe ce qui peut se révéler très pratique. Un premier exemple a été présenté pour la définition des constantes :

```
constdefs
  ouex :: "bool ⇒ bool ⇒ bool" (infixr "⊕" 5)
  "a ⊕ b == (¬a ∧ b) ∨ (a ∧ ¬b)"
```

Les mots clés `infixr`, `infixl` et `infix` permettent de créer des opérateurs infixes respectivement associatifs à droite, associatifs à gauche ou non associatifs.

Il est aussi possible de définir des transformations syntaxiques :

```
syntax "_if" :: "[bool, 'a, 'a] ⇒ 'a" ("(_ ?/ _ :/ _)")

translations
  "a ? b : c" == "if a then b else c"
```

Ici nous définissons une nouvelle syntaxe pour le `if then else` en décrivant tout d'abord cette syntaxe et en donnant son type à l'aide du mot clé `syntax`, puis en décrivant la transformation à l'aide du mot-clé `translations`.

On peut alors écrire :

```
normal_form "True ? 5 : 1"
```

qui nous donne 5 pour résultat, `normal_form` permettant de mettre une expression sous forme normale.

2.2 Scala

Scala [6] est un langage multi-paradigme combinant les possibilités des langages fonctionnels et impératifs orientés objets. Il unifie en effet les concepts fonctionnel et orienté objets dans un langage où toutes les fonctions et toutes les valeurs sont

des objets. Il est interfaçable de manière directe avec Java et $C^{\#}$, les langages orientés objets les plus utilisés actuellement.

Nous l'avons donc utilisé en premier lieu pour ses capacités impératives orientées objets, nous permettant de mettre en place du partage et de la modification de données, puis pour sa proximité avec les langages fonctionnels, facilitant la transformation à partir d'Isabelle, qui utilise un langage fonctionnel. C'est en effet cette proximité mais aussi ses types génériques qui se sont imposés face aux plus utilisés Java, C^{++}, ou $C^{\#}$.

2.2.1 "Case Classes"

Une caractéristique intéressante des langages fonctionnels est la présence des types algébriques, qui permettent de définir des types de données que l'on peut ensuite filtrer à l'aide de motifs. Scala présente un système du même ordre nommé "Case Classes". On définit et utilise ces classes de la manière suivante :

```
sealed abstract class Expr
  case class Var(name:String) extends Expr
  case class Add(e1:Expr, e2:Expr) extends Expr

Add(Var "x", Var "y") match
  Add(e1, e2) => e1
```

Elles sous-typent alors une classe commune comme les constructeurs des types algébriques ont un type commun. Cependant, les "case classes" ont aussi leurs propres types. Le fait de les déclarer en tant que "`case`" permet d'effectuer du filtrage par motif sur celles-ci, et le fait de déclarer la classe supérieure "`sealed`" permet d'empêcher l'ajout d'autres constructeurs à ce type en dehors du contexte de cette classe, en plus de permettre au compilateur d'afficher des warnings lorsque le filtrage n'est pas complet.

2.2.2 Attributs

Il peut être intéressant de pouvoir définir des attributs sans savoir si ceux-ci sont internes à l'objet ou calculés. En effet cela permet d'avoir une interface unifiée en cachant les détails d'implantation et donc plus simple à utiliser. Scala permet de faire cela d'une façon très naturelle et homogène.

La définition d'une valeur (attribut non muable) se fait à l'aide du mot-clé `val`. Elle peut être simulé (virtuelle) par la définition d'une fonction sans argument.

La définition d'une variable (attribut muable) à l'aide du mot-clé `var` est équivalente à la définition de deux fonctions :

- un "getter" : `def x :T`
- et un "setter" : `def x_= (y :T) :Unit`

Ces définitions permettent d'utiliser les syntaxes `a.x` et `a.x = y` où la seconde remplace `a.x_=(y)`.

2.3 (O)Caml

Objective Caml [7] est un langage initialement fonctionnel (Caml) auquel ont été ajouté des extensions impérative et orienté objet. Il dispose d'un système de types puissant, doté de polymorphisme paramétrique et d'inférence de types.

Nous l'avons utilisé pour ses capacités fonctionnelles, et notre meilleure connaissance de celui-ci par rapport aux autres langages fonctionnels existants, tels que Haskell ou SML.

2.4 Les Monades

Les monades sont un concept issue de la théorie des catégories, introduit en tant qu'outil informatique par Moggi [8] puis vulgarisé par Wadler [1] et leur utilisation par le langage Haskell [9]. Elles permettent d'associer, de façon transparente à l'utilisation, un état à, selon l'interprétation que l'on en fait, un conteneur ou un calcul. Cet état peut-être totalement quelconque et peut ainsi représenter par exemple un simple état d'erreur, une mémoire partagée, ou la totalité du monde extérieur au programme. Les programmes manipulant un état à l'aide de monades restent donc purement fonctionnels, car la monade et donc l'état sont effectivement bien qu'implicitement transmis.

De plus, les monades étant issues de théories mathématiques, de nombreux résultats peuvent être utilisés ou obtenus, et elles sont aussi suffisamment abstraites pour être génériques, ce qui permet une réutilisation importante de code. En effet, il est possible de définir des structures de contrôle applicables à toutes les monades, tout en pouvant aussi en définir de nouvelles pour des monades particulières.

Les monades définissent ainsi un formalisme pour la gestion d'états, quels qu'ils soient.

Haskell est actuellement le langage le plus connu pour son utilisation intensive des monades, notamment pour gérer les effets de bords indispensables à tout programme : les entrées/sorties. En effet, Haskell est un langage paresseux, *i.e.* n'effectuant les calculs que lorsque qu'ils sont indispensables, et de nombreux problèmes surviennent alors lors de tentatives d'accès au monde extérieur au programme, car l'ordre d'exécution à l'intérieur du programme n'est pas forcément l'ordre des instructions. Les monades permettent dans ce cas de forcer un ordre d'exécution par la simple transmission implicite d'un état ayant en Haskell le type "RealWorld".

Mais les monades sont aussi utiles pour la gestion des données muables en permettant de la même manière que pour les entrées/sorties d'ordonner l'exécution. Il devient ainsi possible de partager des données muables ou d'implanter des tableaux ou des tables de hachage. On bénéficie finalement de nombreux avantages des langages impératifs tout en restant dans un cadre purement fonctionnel.

2.4.1 Utilisation

Définition

Une monade se compose d'un ensemble de 3 éléments appelé triplet de Kleisli :

un constructeur de type `M` qui permet de construire à partir d'un type `t` le type `M t` de la monade. Lorsque le type `t` est une fonction, il est plus facile de considérer la monade comme un calcul sur `t` avec une dépendance à un état, tandis que s'il ne l'est pas il est plus facile de considérer la monade comme un conteneur d'une valeur `t` associée à un état.

une opération "`return`" (ou "`unit`") de type "`'a => M 'a`" qui permet de construire une monade de type `M t` à partir d'une valeur/fonction de type `t`.

une opération "`bind`" notée $\trianglerighteq$ de type "`M 'a => ('a => M 'b) => M 'b`" qui permet, en considérant la monade comme un conteneur, d'effectuer des calculs sur la valeur qu'elle contient. Si l'on considère une monade comme un calcul, alors cette opération permet de séquencer les calculs, en fournissant au second calcul le résultat du premier. Dans tous les cas, l'état peut-être modifié en fonction de la définition de la monade.

Une autre opération permet de séquencer les calculs lorsque l'on ne souhaite pas garder le résultat du premier calcul : "`then`" notée $\triangleright$ de type "`M 'a => M 'b => M 'b`" définissable à partir de "`bind`".

La notation do

Cependant, l'utilisation des monades sous cette forme n'est pas facile et il est possible de définir une syntaxe beaucoup plus simple et très semblable à du code impératif par une simple transformation syntaxique. Cette syntaxe est appelée "notation do" dans le langage Haskell.

Les règles de transformation de la notation do vers la notation fonctionnelle sont définies ainsi :

BIND

$$\|do\ \texttt{x} \leftarrow a;\ b\| \quad \rightarrow \quad a \trianglerighteq (\lambda\ x.\ \|b\|)$$

THEN

$$\|do\ a;\ b\| \quad \rightarrow \quad a \triangleright \|b\|$$

DO

$$\|do\ a\| \quad \rightarrow \quad a$$

et permettent d'écrire ce code :

```
do
  x ← f 1;
  write x;
```

```
  return x
```

à la place de celui-ci :

```
f 1 ⊵ λ x.
write x ▷
return x
```

Les lois

Évidemment, toute définition ne convient pour la création d'une monade et celle-ci doit avant tout respecter trois lois :

Identité à droite : `∀ f v. return v ⊵ f = f v`

Identité à gauche : `∀ a. a ⊵ return = a`

associativité : `∀ x f g. (x ⊵ f) ⊵ g = x ⊵ (λ v. f v ⊵ g)`

Les deux premières assurent que "`return`" conserve toute l'information contenue dans son argument tandis que la troisième assure l'associativité de "`bind`", ce que l'on voit mieux en remplaçant `f` par "`λ v. f v`" :

```
∀ x f g. (x ⊵ λ v. f v) ⊵ g = x ⊵ (λ v. f v ⊵ g)
```

ou en notation do en remplaçant aussi g par "λ w. g w" :

```
do w ← do v ← x;
          f v
   g w
```

=

```
do v ← x;
   w ← f v;
   g w
```

Chapitre 3

Des pointeurs certifiés

Dans ce chapitre, la partie 3.2 commencera par présenter un programme introduisant le problème. Nous nous intéresserons alors à une traduction directe de ce programme vers Scala dans la partie 3.3. Ensuite nous verrons dans la partie 3.4 une optimisation de cette première traduction qui permettra d'introduire l'utilisation des monades. Pour permettre la vérification de programmes utilisant des monades, nous étudierons la possibilité de les utiliser en Isabelle dans la partie 3.5. Enfin, dans la partie 3.6 nous présenterons la traduction des monades vers Scala.

La syntaxe qui sera utilisée pour les règles de traduction est présentée dans la partie 3.1.

3.1 Syntaxe

Dans les règles de transformations sont adoptés quelques conventions syntaxiques pour en simplifier l'écriture et la lecture :

- NAME $A \rightarrow B$: désigne la règle de transformation de A vers B nommée "name".
- $\langle T \rangle$: désigne la représentation du type OCaml T en Scala.
- $\|X\|$: désigne la représentation du code OCaml X en Scala.
- a : les termes en *italique* sont des termes génériques.
- `a` : les termes écrits en `monospace` sont présents tels quels dans le code.
- a^t : indique que le terme a est de type t.

3.2 Introduction

Pour illustrer les propos, commençons par un exemple de programme fonctionnel pur écrit en OCaml (sans utilisation des extensions impérative ou orientée objet). Il permet de décorer l'arbre syntaxique d'une expression avec les types de ses sous-expressions. Il définit une structure modélisant un programme composé de déclarations de variables entières ou booléennes et d'une expression permettant

d'effectuer des sommes entre ces variables et des constantes (Des conjonctions auraient aussi pu être ajouté mais elles n'apportaient rien par rapport aux sommes. Leur définition a été mise en commentaire et le code s'y rapportant a été supprimé).

Il est composé des définitions :

– des structures et types utilisés :

Listing 3.1 – typing.ml

```
(* Variable names type *)
type varName = string

(* Type constructors *)
type tp =
  | IntT
  | BoolT

(* Value constructors *)
type value =
  | IVal of int
  | BVal of bool

(* Decorated expression tree constructors *)
type 'a expr =
  | Var of 'a * varName
  | Cst of 'a * value
  | Add of 'a * 'a expr * 'a expr
(*| And of 'a * 'a expr * 'a expr*)

(* Variable declaration type *)
type decl = varName * tp

(* Program constructor *)
type 'a prog =
  | Prog of (decl list) * 'a expr
```

– de fonctions auxiliaires permettant de récupérer les types des constantes, des variables et des expressions :

Listing 3.2 – typing.ml

```
(* Get the type of a value *)
let type_of_value = function
  | IVal _ -> Some IntT
  | BVal _ -> Some BoolT

(* Get the type of a variable *)
let rec type_of_var decls vn =
  match decls with
    | [] -> None
    | (vn1, tp1) :: decls' ->
      if vn1 = vn then Some tp1
      else type_of_var decls' vn
```

```
   (* Get the type of a typed expression *)
40 let type_of_tp_expr = function
     | Var (t, _) -> t
     | Cst (t, _) -> t
     | Add (t, _, _) -> t
```

– de la fonction principale "tp_expr" qui parcourt récursivement l'expression qu'elle prend en argument pour en retourner une nouvelle instance typée :

Listing 3.3 – typing.ml

```
   (* Returns the expression and its sub-expressions decorated with
      their types *)
45 let rec tp_expr e decls =
     match e with
       | Var (_, vn) -> Var (type_of_var decls vn, vn)
       | Cst (_, v) -> Cst (type_of_value v, v)
       | Add (_, a, b) ->
50       (let ta = tp_expr a decls in
         let tb = tp_expr b decls in
           match (type_of_tp_expr ta, type_of_tp_expr tb) with
             | (Some IntT, Some IntT) -> Add(Some IntT, ta, tb)
             | (_, _) -> Add(None, ta, tb))
```

– et d'une dernière fonction permettant de typer un programme :

Listing 3.4 – typing.ml

```
55 (* Returns the expression of the program decorated with types *)
   let tp_prog (Prog(decls, e)) = tp_expr e decls
```

Ce programme de base étant présenté, nous pouvons passer à sa traduction en Scala.

3.3 De Caml vers Scala

La première étape de la traduction se devait d'être la traduction de l'ensemble du langage d'origine, en l'occurrence OCaml, vers le langage cible, en l'occurrence Scala, sans se préoccuper des optimisations qui allaient être apportées par la suite.

3.3.1 Types

Toute traduction entre langages typés statiquement commence par une traduction de types. Ceux-ci étant effectivement l'image homogène du langage sous-jacent.

TP__CONSTR
$\langle (b_1, b_2, ...)\ a \rangle \quad \rightarrow \quad \langle a \rangle \left[\langle b_1 \rangle , \langle b_2 \rangle , ... \right]$

TP__ARROW
$\langle a \Rightarrow b \rangle \quad \rightarrow \quad \langle a \rangle \Rightarrow \langle b \rangle$

TP__TUPLE
$\langle (a_1, a_2, ...) \rangle \quad \rightarrow \quad \left(\langle a_1 \rangle , \langle a_2 \rangle , ... \right)$

TP__IDENT[a]
$\langle M_1.M_2....id \rangle \quad \rightarrow \quad \texttt{M_}M_1.\texttt{M_}M_2....\texttt{T_}id$

TP__VAR[b]
$\langle 'a \rangle \quad \rightarrow \quad x$

[a] Ajout des préfixes M_ et T_
[b] Nommage des variables de type en fonction du contexte

Les langages étant assez semblables au niveau des types, il n'y a pas grand chose d'intéressant si ce n'est l'ajout de préfixes M_ et T_ permettant un élargissement de l'espace de nommage dans le langage cible, la majorité des constructions de départ d'espaces de nommage différents ayant pour équivalent les classes ou les objets.

Les suffixes ajoutés sont alors :

M_ pour les noms de modules

T_ pour les noms de types, utilisé en particulier pour les types algébriques

C_ pour les noms des constructeurs des types algébriques

aucun mais première lettre en minuscule pour les fonctions/méthodes, les variables et les attributs.

3.3.2 Langage

Vient ensuite la traduction du langage en lui même, en commençant par les éléments de haut niveau, les définitions de module, de variables ou fonctions et de types.

Définitions

MODULE

$\|$`module` m = `struct` x `end ;`$\|$ $\longrightarrow$ `object M_`m{ $\|x\|$ }

LET

$\left\| \texttt{let}\ f\ a_1^{t_1}\ a_2^{t_2}\ ...=e \right\|$ $\longrightarrow$ `def` f[Variables de type]$(a_1\ :\ \langle t_1\rangle)(a_2\ :\ \langle t_2\rangle)...$ = { $\|e\|$ }

TYPE_DEF

$\|$`type` $(b_1, b_2, ...)\ a\ =\ x\|$ $\longrightarrow$ Définition de type[3.3.2]

Les variables de type des fonctions dans LET sont récupérées à partir du type inféré de f.

Définition de type

TYPE_ABSTRACT

$\left\| \texttt{type}\ (b_1, b_2, ...)\ a\ =\ t^{\texttt{type}} \right\|$ $\longrightarrow$ `type T_`a = $\langle t\rangle$

TYPE_VARIANT[a]

$$\left\| \begin{array}{l} \texttt{type}\ (b_1, b_2, ...)\ a\ = \\ \mid C_1\ \texttt{of}\ (t_{1,1},\ t_{1,2},\ ...) \\ \mid ... \\ \mid C_n\ \texttt{of}\ (t_{n,1},\ t_{n,2},\ ...) \\ \mid ... \\ \mid C_m \\ \mid ... \end{array} \right\| \longrightarrow$$

```
abstract class T_a[+⟨b1⟩, +⟨b2⟩ ...]
  case class C_C1[⟨b1⟩, ⟨b2⟩, ...]($1 : ⟨t1,1⟩, $2...)
    extends T_a[⟨b1⟩, ⟨b2⟩ ...]
  ...
  case class C_Cn[⟨b1⟩, ⟨b2⟩, ...]($1 : ⟨tn,1⟩, $2...)
    extends T_a[⟨b1⟩, ⟨b2⟩ ...]
  ...
  case object C_Cm extends T_a[Nothing, Nothing...]
  ...
```

TYPE_RECORD

$$\left\| \begin{array}{l} \texttt{type}\ (\langle b_1\rangle,\ \langle b_2\rangle,\ ...)\ a\ = \\ \{f_1 : \langle t_1\rangle, f_2 : \langle t_2\rangle, ...\} \end{array} \right\| \longrightarrow \texttt{class T_}a[\langle b_1\rangle,\ \langle b_2\rangle,\ ...](\texttt{val}\ f_1 : \langle t_1\rangle,\ \texttt{val}\ f_2 : \langle t_2\rangle,\ ...)$$

[a] Les signes "+" sont ajoutés pour indiquer que les paramètres de types sont covariants.

Ici, les $\$i$ utilisés pour les noms des arguments des constructeurs des "case classes" pourraient préter à confusion, ressemblants à une instruction spéciale de Scala mais le \$ est un caractère comme les autres et il est utilisé ici pour générer des noms ne pouvant pas exister dans le code OCaml.

Expressions

Puis la traduction des expressions.

MATCH

$$\left\| \begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \mid p_1\ -\!>\ e_1 \\ \mid p_2\ -\!>\ e_2 \\ \mid \dots \end{array} \right\| \rightarrow \begin{array}{l} \|e\|\ \texttt{match}\ \{ \\ \quad \texttt{case}\ \|p_1\|\ \texttt{=>}\ \|e_1\| \\ \quad \texttt{case}\ \|p_2\|\ \texttt{=>}\ \|e_2\| \\ \quad \texttt{case}\ \dots \\ \} \end{array}$$

LET__IN

$$\|\texttt{let}\ a\ =\ e1\ \texttt{in}\ e2\| \rightarrow \begin{array}{l} \{ \\ \quad \texttt{val}\ a\ =\ \|e1\|\,; \\ \quad \|e2\|\,; \\ \} \end{array}$$

...

$$\dots \rightarrow \dots$$

Scala étant aussi un langage fonctionnel, les expressions se ressemblent beaucoup et les traductions sont pour la plupart évidentes. Les transformations concernant les monades sont définies dans la partie 3.6.

3.3.3 Résultat

Par la traduction directe, voici donc le code que nous obtenons :

Listing 3.5 – Typing.scala

```
0 object Typing{
    type T_varName = String
    abstract class T_tp
      case object C_IntT extends T_tp
      case object C_BoolT extends T_tp
5   abstract class T_value
      case class C_IVal($1: Int) extends T_value
      case class C_BVal($1: Boolean) extends T_value
    abstract class T_expr[+A]
      case class C_Var[A]($1: A, $2: T_varName) extends T_expr[A]
10    case class C_Cst[A]($1: A, $2: T_value) extends T_expr[A]
      case class C_Add[A]($1: A, $2: T_expr[A], $3: T_expr[A]) extends
          T_expr[A]
    type T_decl = (T_varName, T_tp)
    abstract class T_prog[+A]
      case class C_Prog[A]($1: List[T_decl], $2: T_expr[A]) extends
          T_prog[A]
15  def type_of_value(args : T_value) = {
      args match {
        case C_IVal(_) =>
          Some(C_IntT)
        case C_BVal(_) =>
20        Some(C_BoolT)
      }
    }
    def type_of_var[A, B](decls : List[(A, B)])(vn : A) : Option[B] = {
```

```
    decls match {
      case List() =>
        None
      case ::((vn1, tp1), decls$) =>

        if((vn1.==) (vn)){
          Some(tp1)
        }else{
          (type_of_var (decls$)) (vn)
        }
    }
  }
  def type_of_tp_expr[A](args : T_expr[A]) = {
    args match {
      case C_Var(t, _) =>
        t
      case C_Cst(t, _) =>
        t
      case C_Add(t, _, _) =>
        t
    }
  }
  def tp_expr[A](e : T_expr[A])(decls : List[(T_varName, T_tp)]) :
      T_expr[Option[T_tp]] = {
    e match {
      case C_Var(_, vn) =>
        C_Var((type_of_var (decls)) (vn), vn)
      case C_Cst(_, v) =>
        C_Cst(type_of_value (v), v)
      case C_Add(_, a, b) =>
        {
          val ta = (tp_expr (a)) (decls);
          val tb = (tp_expr (b)) (decls);
          (type_of_tp_expr (ta), type_of_tp_expr (tb)) match {
            case (Some(C_IntT), Some(C_IntT)) =>
              C_Add(Some(C_IntT), ta, tb)
            case (_, _) =>
              C_Add(None, ta, tb)
          }
        }
    }
  }
  def tp_prog[A](args : T_prog[A]) = {
    args match {
      case C_Prog(decls, e) =>
        (tp_expr (e)) (decls)
    }
  }
}
```

3.4 Introduction des monades

La traduction directe de OCaml vers Scala étant effectuée, il est maintenant possible d'optimiser le résultat, par une transformation sémantique. Nous reprenons l'exemple typing.ml (Listings 3.1, 3.2, 3.3 et 3.4) dans lequel le second argument de "tp_expr" (Listing 3.3), la liste de déclarations de variables, va nous intéresser. En effet, dans un langage impératif avec pointeurs cet argument disparaîtrait certainement car il est plus efficace que les noms des variables, utilisés comme identificateurs, soient remplacés par des pointeurs vers une structure, unique pour chaque variable, contenant son nom et son type, *i.e.* ses propriétés. Cette structure serait alors partagée entre les différentes utilisations de la variable dans l'expression. Cela permet un gain certain en temps d'accès aux types en remplaçant les évaluations de la fonction "type_of_var" (ligne 32 du Listing 3.2) par autant de déréférencements de pointeurs[1].

Voilà alors les modifications qui seraient apportées :

– Le remplacement du pointeur vers le nom de variable par un pointeur vers la déclaration de la variable :

Listing 3.6 – Typing2.scala

```
   abstract class T_expr[+A]
10   case class C_Var[A]($1: A, $2: T_decl) extends T_expr[A]
     case class C_Cst[A]($1: A, $2: T_value) extends T_expr[A]
     case class C_Add[A]($1: A, $2: T_expr[A], $3: T_expr[A])
         extends T_expr[A]
```

– Le remplacement de la recherche de la déclaration par un déréférencement de pointeur :

Listing 3.7 – Typing2.scala

```
25 def type_of_var(d : T_decl) : Option[T_tp] = {
     Some(d._2)
   }
```

– La suppression du second argument de "tp_expr" dans sa définition et ses applications (et accessoirement le changement du nom de la variable *vn* en *d*) :

Listing 3.8 – Typing2.scala

```
40 def tp_expr[A](e : T_expr[A]) : T_expr[Option[T_tp]] = {
     e match {
       case C_Var(_, d) =>
```

[1]Évidemment, pour cet exemple simple, il serait tout a fait possible d'effectuer cette optimisation de façon purement fonctionnelle en remplaçant "varName" par "decl" dans "Var of varName" (ligne 15) et en utilisant ensuite les variables crées pour les déclarations dans l'expression. Cependant, il serait alors impossible de modifier les données partagées. Ainsi si une autre fonction effectuait une modification des types, bien que cela n'ai pas de sens dans ce cas, il y aurait un compromis à faire entre les efficacités en lecture et en écriture.

```
          C_Var(type_of_var (d), d)
        case C_Cst(_, v) =>
45        C_Cst(type_of_value (v), v)
        case C_Add(_, a, b) =>
          {
            val ta = tp_expr (a);
            val tb = tp_expr (b);
50          (type_of_tp_expr (ta), type_of_tp_expr (tb)) match {
              case (Some(C_IntT), Some(C_IntT)) =>
                C_Add(Some(C_IntT), ta, tb)
              case (_, _) =>
                C_Add(None, ta, tb)
55          }
          }
      }
    }
    def tp_prog[A](args : T_prog[A]) = {
60    args match {
        case C_Prog(decls, e) =>
          tp_expr (e)
      }
    }
65 }
```

De cette manière, les accès aux types des variables sont plus rapides. Mais comment intégrer, de façon générique, cette modification lors d'une traduction automatique ? Il faut pour cela analyser les modifications effectuées. La modification principale est la suppression du second argument de “tp_expr”, la liste de déclarations de variables. Cet argument était en fait transmis à chaque appel récursif de “tp_expr” et formait un état des déclarations, qui est ensuite contenu dans la mémoire pour la version impérative. Notre souhait est de pouvoir en quelque sorte déclarer cet argument comme un état, ou plus généralement de pouvoir gérer des états. Or, pour gérer des états dans un langage fonctionnel, l'idée d'utiliser des monades (cf. 2.4) vient naturellement, pour peu qu'on en connaisse leur existence. Elles permettent effectivement de gérer plus facilement des états du coté fonctionnel, en plus d'identifier les états pour les traduire de façon différente.

Ainsi notre choix s'est porté sur les monades, que nous avons dû modéliser dans un environnement permettant d'effectuer une vérification formelle, et c'est Isabelle que nous avons choisi.

3.5 Des monades en Isabelle

Nous avons vu dans 2.4 que les monades permettent de gérer des états de façon purement fonctionnelle. C'était donc ce qu'il nous fallait et ce que nous avons utilisé. Plus précisément, nous avions besoin d'une monade permettant de transmettre et de modifier des états. Nous avons alors commencé à développer notre propre solution jusqu'à découvrir que Peyton Jones et Wadler, contributeurs majeurs à la conception du langage Haskell, ont défini une monade de transformation

d'état dans ce langage [10]. Nous nous en sommes alors inspiré, d'autant plus que de nombreuses similitudes existaient déjà.

Pourtant, cette monade n'est pas parfaitement sûre d'utilisation, car des références à l'état transmis par la monade peuvent être utilisés dans des états différents, en exécutant les monades à l'aide de runST puis en les réutilisant dans une autre. Par la suite, ils ont découvert qu'un polymorphisme de type de second ordre (ou première classe) permettait une utilisation sûre de cette monade [11]. Cependant, le polymorphisme de type de second ordre rend l'inférence de type indécidable et Isabelle ne le permet pas. Nous nous sommes donc contenté de cette version qui n'est pas vraiment gênante dans notre cas car les programmes peuvent tout de même être vérifiés explicitement.

Voici la définition à laquelle nous sommes finalement arrivés en Isabelle. Comme toute monade, elle consiste en son type :

```
datatype ('v, 's) state_trans = ST "'s ⇒ 'v × 's"
```

auquel on rajoute l'opération "runST"permettant d'appliquer un état à la monade de transformation d'état et d'obtenir le résultat :

```
fun runST :: "('v, 's) state_trans ⇒ 's ⇒ 'v × 's"
where
  "runST (ST m) = m"
```

puis les opérations classiques "return", "bind" et optionnellement "then" suffixés dans ce cas particulier par "ST" :

```
constdefs
  returnST :: "'v ⇒ ('v, 's) state_trans"
  "returnST m ≡ ST (λ s. (m, s))"

constdefs
  bindST :: "('v1, 's) state_trans ⇒ ('v1 ⇒ ('v2, 's) state_trans)
      ⇒ (('v2, 's) state_trans)" (infixr "⊵" 55)
  "bindST m f ≡ ST (λ r. (let (x, s) = runST m r in runST (f x) s))"

constdefs
  thenST :: "('v1, 's) state_trans ⇒ ('v2, 's) state_trans ⇒ ('v2,
      's) state_trans" (infixr "⊳" 55)
  "thenST m1 m2 ≡ ST (λ r. let (x, s) = runST m1 r in runST m2 s)"
```

Il est alors possible de vérifier que la monade que nous venons de définir est correcte, *i.e.* qu'elle vérifie les trois axiomes :

```
lemma st_left_identity: "∀ v f. (returnST v) ⊵ f = f v"
lemma st_right_identity: "∀ a. a ⊵ returnST = a"
lemma st_assoc: "∀ x f g. (x ⊵ f) ⊵ g = x ⊵ (λ v. ((f v) ⊵ g))"
```

puis de définir une transformation syntaxique en Isabelle pour pouvoir utiliser la notation do :

```
syntax
```

```
  "_bind" :: "[pttrn, 'a, 'b] ⇒ 'c"
                                             ("(_/ ←/ _;/ _)" [0, 0, 10]
                                                10)
  "_then" :: "['a, 'b] ⇒ 'b"
                                             ("(2do _;/ _)" [0, 10] 10)
  "_end" :: "['a, 'b] ⇒ 'b"
                                             ("(1do _;/)" [10] 10)

translations
  "x ← p; q" == "p ⊵ (λ x. q)"
  "do p; q" == "p ⊳ q"
  "do p;" => "p"
```

La monade de transformation d'état était alors définie en Isabelle et utilisable, mais elle était encore générique concernant son état. Nous avons alors défini :

– des références vers des valeurs de type $'v$:

```
datatype
  'v ref = Ref nat
```

– une fonction vérifiée permettant de générer des références toujours différentes :

```
consts
  list_max :: "'n ref list ⇒ nat"
primrec
  "list_max [] = 0"
  "list_max (r # rs) = (case r of Ref x ⇒ max x (list_max rs))"

lemma list_max_leq: "Ref x ∈ set vs ⟶ x ≤ list_max vs"
by (induct vs, auto, split ref.split, auto)

lemma list_max_Suc: "Ref (Suc (list_max vs)) ∉ set vs"
by (fastsimp dest: list_max_leq)
```

– le type d'un état et d'une monade de transformation d'état portant sur des références :

```
types
  ('n, 'v) state = "'n list × 'n ⇒ 'v option"
  ('v, 'ret) ref_trans = "('ret, ('v ref, 'v) state)
      state_trans"
```

puis quelques opérations permettant d'utiliser tout cela :

– une fonction d'allocation des références :

```
constdefs
  alloc :: "('v, 'v ref) ref_trans"
  "alloc == ST (λ (vs, m). (let n = Suc (list_max vs) in (Ref
      n, ((Ref n)#vs, m))))"
```

– une fonction de lecture :

```
constdefs
  read :: "'v ref ⇒ ('v, 'v) ref_trans"
  "read n ≡ ST ( λ (vs, m).
    (case m n of Some v ⇒ (v, (vs, m))))"
```

– une fonction d'assignation :

```
constdefs
  write :: "['v ref, 'v] Rightarrow ('v, unit) ref_trans" (infix
      ":=" 5)
  "write n v ≡ ST ( λ (vs, m).  ((), (vs, m(n ↦ v))))"
```

– une fonction de création :

```
constdefs
  new :: "'v ⇒ ('v, 'v ref) ref_trans"
  "new v ≡ (x ← alloc; do x := v; returnST x)"
```

et il est alors possible d'écrire des programmes l'utilisant :

```
fun
  foo :: "[nat, nat] ⇒ (nat, (nat × nat)) ref_trans"
where
  "foo wx wy = (
    x ← alloc;
    y ← new 2;
    do x := 1;
    do y := wy;
    do x := wx;
    vx ← read x;
    vy ← read y;
    returnST (vx, vy)
  )"
```

et de les vérifier :

```
  lemma "∀ s x y. ∃ s'. runST (foo x y) s = (( x, y), s')"
  by (simp add: Let_def ref_trans_monad_defs max_def)
```

Et voici ce que donne le programme de la section 3.2 écrit à l'aide de la monade :

```
0  theory typing imports ref_trans_monad

   begin

   datatype tp =
5    IntT
   | BoolT

   types
     varName = string
10   decl = "varName * tp"
     'a decl_trans = "(decl, 'a) ref_trans"
```

```
datatype "value" =
  IVal "nat"
| BVal "bool"

datatype 'a expr =
  Var "'a" "decl ref"
| Cst "'a" "value"
| Add "'a" "'a expr" "'a expr"
(*| And "'a" "'a expr" "'a expr"*)

fun tp_value :: "value ⇀ tp"
where
  "tp_value (IVal _) = Some IntT"
| "tp_value (BVal _) = Some BoolT"

fun type_of_var :: "decl ref ⇒ (tp option) decl_trans"
where
  "type_of_var d = (do
     x ← read d;
     returnST (let (vn, tp) = x in Some tp);)"

fun type_of_expr :: "'a expr ⇒ 'a"
where
  "type_of_expr (Var tp _)   = tp"
| "type_of_expr (Cst tp _)   = tp"
| "type_of_expr (Add tp _ _) = tp"

fun tp_expr :: "'a expr ⇒ (tp option expr) decl_trans"
where
  "tp_expr (Var _ d)   = (do
    tp ← type_of_var d;
    returnST (Var tp d);)"
| "tp_expr (Cst _ v)   = returnST (Cst (tp_value v) v)"
| "tp_expr (Add _ a b) = (do
    ta ← tp_expr a;
    tb ← tp_expr b;
    returnST (case (type_of_expr ta, type_of_expr tb) of
      (Some IntT, Some IntT) ⇒ Add (Some IntT) ta tb
    | (_, _)                 ⇒ Add None ta tb);)"

end
```

3.6 D'Isabelle vers Scala

La dernière étape aurait du consister en l'écriture d'un traducteur d'Isabelle vers Scala. Il existe d'ailleurs dans Isabelle un moyen pour extraire le code exécutable d'une théorie vers d'autres langages, les langages supportés étant pour le moment SML, Haskell et OCaml. Pour ce faire, Isabelle utilise un code intermédiaire générique appelé Thin-gol à partir du quel il doit être possible d'implémenter la fin de la traduction vers le langage de son choix. Cependant, Isabelle et le tra-

ducteur sont tout d'abord écrit en SML, langage que nous ne connaissons que très peu, et pressés par le temps, nous n'avons de plus pas compris comment les traductions finales étaient effectuées, et si il aurait été possible de le faire pour Scala.

Néanmoins, lors des premières expériences nous avions développé un traducteur de OCaml vers Scala composé :

- d'un parseur réduit de OCaml
- d'un traducteur entre les arbres syntaxiques de OCaml et de Scala
- et d'un générateur de Scala

Nous avons alors décidé de développer la traduction en utilisant en premier lieu le code OCaml généré par Isabelle puis la traduction de OCaml vers Scala. Il manquait cependant une partie importante à cette traduction qui était l'inférence de types, certain types étant explicitement nécessaires en Scala. Nous avons alors récupéré les sources du parseur et de l'inférence de type du compilateur OCaml pour récupérer un arbre syntaxique typé du code OCaml, et générer à partir de celui-ci un arbre syntaxique du code Scala à générer.

La traduction directe de OCaml vers Scala ayant été présenté dans la partie 3.3, nous allons nous concentrer ici sur la partie intéressante de la traduction : la transformation des monades en Scala.

3.6.1 D'Isabelle vers OCaml

Le développement de la traduction avec une étape en OCaml a certainement permis de finir rapidement la traduction mais a tout de même présenté un inconvénient majeur : les types n'étaient pas totalement conservés lors de la génération de OCaml par Isabelle. En effet, il n'est pas nécessaire, pour avoir du code exécutable, de préserver l'intégralité des types. Ainsi, lorsqu'en Isabelle nous exprimons explicitement les types des fonctions, le code OCaml généré n'en tient pas compte. cela pourrait paraître tout à fait anodin, puisque les types sont inférés en OCaml, mais cela ne l'est pas si les types rajoutent du sens, comme dans le cas des types des références (*c.f.* Listing 3.5, section 3.5), impossibles à inférer car les références ne sont en fait que des entiers, que l'on type de manière à assurer que les valeurs associées soient du bon type lors de la récupération à l'aide de "`read`".

Il a alors fallu trouver une solution, et nous avons choisi de rajouter les types nécessaires au code OCaml généré en modifiant simplement les fonctions générés à partir de la théorie "`ref_trans_monad`". Cela a été codé pour être effectué de façon automatique mais pas très propre car modifiant du code généré... La modification consiste simplement à ajouter des annotations de types aux deux fonctions `new` et `read` :

```
let rec newa _V v =
  bindST alloc (fun x ->
  thenST (write _V x v) (
  returnST x));;

let rec read n =
   St (fun ((vs, m) as a) ->
     let Some v = m n in
     (v, (vs, m)));;
```

→

```
let rec newa _V (v :'a) =
  bindST alloc (fun (x :'a ref) ->
  thenST (write _V x v) (
  returnST x));;

let rec read (n :'a ref) =
   St (fun ((vs, m) as a) ->
     let Some (v :'a) = m n in
     (v, (vs, m)));;
```

De même, pour effectuer une traduction spécifique des monades, il fallait pouvoir les distinguer des autres données manipulées. La solution a dans ce cas été plus simple car il a suffit de transformer le type de la monade pour en faire un type algébrique nous permettant de la reconnaître (*c.f.* Listing 3.5, section 3.5).

3.6.2 Règles

Les détails d'implémentations étudiés, il ne reste plus qu'à définir les transformations spécifiques à la traduction des monades.

Pour effectuer la traduction, nous introduisons une classe State_trans_monad importée dans les classes générées et contenant la base nécessaire à la transformation :

```
0  package pack;

   object State_trans_monad {

     class Ref[V]() {
5
       var v:V = _;

       def this(v:V) = {
         this()
10       this.v = v;
       }

     }
   }
```

Elle ne contient pour le moment que la classe `Ref[V]` permettant l'indirection des références vers les valeurs. Celle-ci dispose de deux constructeurs permettant d'initialiser ou non l'attribut `v`.

Types

La première transformation concerne les types. La transformation doit en effet conserver leur cohérence avec le code pendant la traduction. Voici les règles correspondantes :

TP__ERASE

$$\langle (r, s)\ \text{state_trans} \rangle \quad \to \quad \langle r \rangle$$

TP__REF

$$\langle (\text{nat}, n)\ \text{ref} \rangle \quad \to \quad \text{Ref}[\langle n \rangle]$$

La règle TP__ERASE remplace simplement le type d'une monade de transformation d'état par son type de retour $\langle r \rangle$. Au premier abord étrange car elle semble remplacer une fonction par une constante, elle ne fait en fait que rendre l'état de départ implicite en le remplaçant par l'état existant dans les programmes impératifs. $\langle r \rangle$ est en fait le type d'une fonction sans argument modifiant l'état interne.

La règle TP__REF remplace le type des références, qui ne sont que des entiers, par le type de l'objet "`Ref`" introduisant un pointeur vers une valeur.

Structure du langage

Il faut ensuite transformer le code, en commençant par les opérations basiques des monades, permettant de structurer le code :

BIND

$$\left\| x^{r_1} \leftarrow t_1^{(r_1,s)\ \texttt{st}};\ t_2^{(r_2,s)\ \texttt{st}} \right\| \quad \to \quad \texttt{val}\ x^{\langle r_1 \rangle} = \|\ t_1\ \|^{\langle r_1 \rangle};\ \|\ t_2\ \|^{\langle r_2 \rangle}$$

THEN

$$\left\| t_1^{(r_1,s)\ \texttt{st}};\ t_2^{(r_2,s)\ \texttt{st}} \right\| \quad \to \quad \|\ t_1\ \|^{\langle r_1 \rangle};\ \|\ t_2\ \|^{\langle r_2 \rangle}$$

RETURN

$$\|\ \texttt{return}\ t^r\ \| \quad \to \quad \|\ t\ \|^{\langle r \rangle}$$

RUN

$$\left\| \texttt{runST}\ m^{(r,s)\ \texttt{st}}\ s^s \right\| \quad \to \quad \|\ m\ \|^{\langle (r,s)\ \texttt{st} \rangle \equiv \langle r \rangle}$$

Les transformations BIND, THEN et RETURN sont assez directes par rapport à la notation do (*c.f.* section 2.4.1).

L'opération "`runST`" permet d'extraire des données de la monade en l'exécutant à partir d'un état de départ. Il paraîtrait possible d'effectuer sa traduction par la règle RUN mais par manque de temps, l'étude n'a pu être effectué de façon plus approfondie et son utilisation est pour le moment interdite dans le code exécutable (mais autorisée dans les preuves) par son absence de traduction. Elle n'est cependant pas nécessaire, car tout le code fonctionnel et monadique peut être écrit à l'intérieur d'une monade.

Primitives du langage

La structure du code ne serait rien sans les primitives nécessaires à la manipulation de l'état :

ALLOC

$$\left\| \texttt{alloc}^{(a,\ a\ \texttt{ref})\ \texttt{ref_trans}} \right\| \quad \rightarrow \quad \texttt{new Ref}[\langle a \rangle]()$$

NEW

$$\left\| \texttt{new}^{'a\ \rightarrow\ ('a\ \texttt{ref})\ \texttt{state_trans}}\ v^a \right\| \quad \rightarrow \quad \texttt{new Ref}(\| v \|)$$

WRITE

$$\| \texttt{write}\ x\ v \| \equiv \| x \ \ \texttt{:=}\ v \| \quad \rightarrow \quad \| x \| .v = \ \| v \|$$

READ

$$\| \texttt{read}\ x \| \quad \rightarrow \quad \| x.v \|$$

Ainsi, `alloc` est traduit en `Ref(null)`, ce qui permet d'allouer une variable non initialisée, mais pouvant être initialisée par la suite à l'aide de `write`, traduit en opération d'affectation de l'attribut `v` de son premier argument, instance de la classe `Ref`. Ces deux étapes peuvent être regroupées en une seule à l'aide de `new`. Enfin `read` permet de récupérer la valeur pointée par la référence, *i.e.* récupérer l'attribut `v` de son argument.

Résultat

Et voici enfin le code généré à partir du programme Isabelle "`tp_expr`" présenté dans le Listing 3.5 duquel ont été retiré les définitions ajoutés par Isabelle :

```
0  package pack ;

   import State_trans_monad._

   object Tp_expr{
5    object M_Typing{
       abstract class T_tp
         case object C_BoolT extends T_tp
         case object C_IntT extends T_tp
       abstract class T_value
10       case class C_BVal($1: Boolean) extends T_value
         case class C_IVal($1: Int) extends T_value
       abstract class T_expr[+A]
         case class C_Add[A]($1: A, $2: T_expr[A], $3: T_expr[A])
             extends T_expr[A]
         case class C_Cst[A]($1: A, $2: T_value) extends T_expr[A]
15       case class C_Var[A]($1: A, $2: Ref[(List[M_List.T_char],
             T_tp)]) extends T_expr[A]
       def tp_case[A](f1 : A)(f2 : A)(x2 : T_tp) : A = {
         (f1, f2, x2) match {
           case (f1, f2, C_IntT) =>
             f1
20         case (f1, f2, C_BoolT) =>
             f2
         }
```

```
    }
    def type_of_var[A, B](d : Ref[(A, B)]) : Option[B] = {
25    val x = d.v;
      {
        val a = x;
        val (vn, aa) = a;
        Some(aa)
30    }
    }
    def tp_value(args : T_value) : Option[T_tp] = {
      args match {
        case C_BVal(uv) =>
35        Some(C_BoolT)
        case C_IVal(uu) =>
          Some(C_IntT)
      }
    }
40  def type_of_expr[A](args : T_expr[A]) : A = {
      args match {
        case C_Add(tp, uw, ux) =>
          tp
        case C_Cst(tp, uv) =>
45        tp
        case C_Var(tp, uu) =>
          tp
      }
    }
50  def tp_expr[A](args : T_expr[A]) : T_expr[Option[T_tp]] = {
      args match {
        case C_Add(uw, a, b) =>
          {
            val ta = tp_expr (a);
55          {
              val tb = tp_expr (b);
              {
                val (x, xa) = (type_of_expr (ta), type_of_expr (tb));
                x match {
60                case None =>
                    C_Add(None, ta, tb)
                  case Some(aa) =>
                    aa match {
                      case C_IntT =>
65                      xa match {
                          case None =>
                            C_Add(None, ta, tb)
                          case Some(ab) =>
                            ab match {
70                            case C_IntT =>
                                C_Add(Some(C_IntT), ta, tb)
                              case C_BoolT =>
                                C_Add(None, ta, tb)
                            }
75                      }
                      case C_BoolT =>
```

```
                        C_Add(None, ta, tb)
                    }
                }
80            }
            }
          }
        case C_Cst(uv, v) =>
          C_Cst(tp_value (v), v)
85      case C_Var(uu, d) =>
          {
            val tp = type_of_var (d);
            C_Var(tp, d)
          }
90    }
    }
  }
}
```

Chapitre 4

Conclusion

Ce rapport a présenté comment il était possible, à partir de programmes purement fonctionnels vérifiables dans des assistant de preuves tels qu'Isabelle, de générer des programmes impératifs faisant l'usage de pointeurs permettant des gains en vitesse d'exécution et espace mémoire utilisé. Ce projet semble prometteur mais reste pourtant encore largement limité et de nombreuses améliorations sont envisageables.

4.1 Améliorations futures

4.1.1 Extension à un modèle orienté objet

Le modèle du langage impératif utilisable est pour l'instant assez limité, ne permettant que l'utilisation de références vers un seul type de valeur par monade. Il faudrait pouvoir définir un état plus générique, à l'image d'une mémoire, mais avec un typage plus fort que cette dernière. Les structures de contrôle et primitives sont elles aussi limités, ne permettant que l'utilisation de références mais aucun concept plus avancé tel que les objets.

Nous avons essayé de définir un état permettant la gestion d'objets mais les parties les plus difficiles étaient l'héritage et le sous-typage. De nombreux exemples de formalisation de langages objets existent, comme dans [12] ou [13]. Cependant, ces formalismes sont très peu intégrés aux langages dans lesquels ils sont manipulés, ce que l'on appelle "Deep embedding" [14] ; les constructions du langages étant modélisés par des types algébriques. Pourtant, il serait souhaitable de pouvoir représenter un langage objet en utilisant les constructions du langage de modélisation, ce qui est appelé "Shallow embedding", permettant une vérification poussée des types.

4.1.2 Méta-modélisation

Une fonctionnalité intéressante pourrait aussi être la génération des structures de données fonctionnelles en Isabelle et objets en Scala à partir d'un méta-modèle

UML défini par exemple dans Eclipse dans des fichiers Ecore. Les accès à ces structures de données pourraient alors aussi être générés de façon optimisé en Scala et compatible avec la version Isabelle.

4.1.3 Génération de code à partir d'Isabelle

Nous utilisons pour le moment une traduction intermédiaire par Caml, avec les avantages mais aussi avec les inconvénients que cela comporte. Il pourrait être intéressant, et plus facilement maintenable et certifiable d'effectuer cette traduction directement du code Isabelle vers Scala, notamment en utilisant les possibilités offertes à ce niveau par Isabelle.

4.1.4 Certification de la traduction

Cette traduction, écrite en Caml et donc dans un syntaxe proche de celle d'Isabelle, devrait être vérifiée pour assurer une préservation de la sémantique dans tous les cas. En effet, vérifier des programmes ne sert à rien si ce sont finalement les programmes générés à partir de ceux-ci de façon non sûre qui sont exécutés.

Bibliographie

[1] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1990.

[2] Philip Wadler. Linear types can change the world. In *Programming Concepts and Methods*, pages 347–359. North, 1990.

[3] http://www.afm.sbu.ac.uk/z/z.html. The Z notation.

[4] Dominique Cansell and Dominique Méry. Computing and informatics, vol. 22, 2003, 1–31, 2003.

[5] http://isabelle.in.tum.de. Isabelle proof assistant website.

[6] http://www.scala-lang.org. Scala programming language website.

[7] http://caml.inria.fr. Caml programming language website.

[8] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1) :55–92, 1991.

[9] http://www.haskell.org. Haskell programming language website.

[10] Simon L Peyton and Jones Philip Wadler. Imperative functional programming, 1993.

[11] Simon L Peyton Jones. State in Haskell, August 20 1996.

[12] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java : A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[13] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4) :619–695, 2006.

[14] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety : Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223, pages 305–320, 2004.