2M11.2785.6

Université de Montréal

La compilation de patrons de filtrage sous Erlang

par

Patrick Piché

Département d'informatique et de recherche opérationnelle

Faculté des arts et sciences

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

mars 2000

© Patrick Piché, 2000

QA
76
U54
2000
v.023

Page d'identification du jury

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :
La compilation de patrons de filtrages sous Erlang

présenté par :
Patrick Piché

a été évalué par un jury composé des personnes suivantes :

Guy Lapalme
(président du jury)

Jean Vaucher

Marc Feeley
(directeur de recherche)

Mémoire accepté le: 13 juin 2000

Sommaire

Le présent travail traite du problème de compilation de patrons de filtrage (*pattern matching*) sous le langage fonctionnel Erlang, développé par Ericsson Telecom. La principale motivation dans l'élaboration de cet algorithme est son intégration dans notre compilateur Erlang vers Scheme (Etos). Les principales contraintes sont tout à fait classiques, nous recherchons l'efficacité sur les plans spatiaux et temporels.

Afin d'atteindre notre objectif, nous avons élaboré un sous-langage qui permet de centraliser le filtrage de patrons sous une seule forme syntaxique. La réduction de Erlang vers ce sous-langage constitue donc une partie importante de nos travaux. D'autre part, nous avons découvert que ce sous-langage possède d'autres propriétés intéressantes dû au fait qu'il s'agisse d'un langage de plus bas niveau. Notons entre autres la possibilité de créer des programmes beaucoup plus efficaces qu'en Erlang pur.

Une fois le mécanisme de filtrage isolé, nous avons procédé à la compilation de ce dernier vers Scheme. Afin d'atteindre cet objectif, nous avons analysé le problème du filtrage pour en identifier les principales composantes. Ainsi, nous avons abordé la prédiction des tests par la rétention d'information structurelles sur les données et la génération d'arbres de décision efficaces. Nous avons implanté des algorithmes qui intègrent par ailleurs les possibles contraintes induites par les expressions de garde.

La recherche d'un générateur d'arbre de décision optimal étant vaine vue la complexité exponentielle du problème, nous avons accordé une attention particulière à l'adaptation d'heuristiques intéressantes qui permettent d'optimiser encore davantage notre méthode dont les résultats intéressants sont dévoilés.

Table des matières

Sommaire iii
Table des matières iv
Liste des figures x
Introduction 1
Chapitre I Introduction au langage Erlang 3
1.1 Classification du langage 4
1.2 Les types de donnée 4
1.3 La syntaxe 4
1.4 Propriétés particulières 5
1.4.1 Modularité 5
1.4.2 Concurrence 6
1.4.3 Distribution 6
1.4.4 Tolérance à la faute 7
1.5 De Erlang vers Scheme 8
Chapitre II Erlang comparé à Scheme 9
2.1 Situation dans le processus global de compilation 10
2.2 Implications des propriétés particulières de Erlang 11
2.2.1 Modularité 11
2.2.2 Concurrence 11
2.2.3 Distribution 11
2.2.4 Tolérance à la faute 12
2.3 Représentation des données 12
2.3.1 Entiers 12
2.3.2 Caractères et chaînes de caractères 13
2.3.3 Nombres à virgule flottante 13
2.3.4 Atomes 13
2.3.5 Listes 13
2.3.6 Tuples 14

2.3.7 Fonctions ... 14
2.3.8 Références ... 14
2.3.9 Identificateurs de processus (PIDs) ... 15
2.3.10 Ports ... 16
2.3.11 Binaires ... 16
2.4 Formes syntaxiques ... 17
2.4.1 Constantes ... 17
2.4.2 Variables ... 18
2.4.3 Les patrons ... 18
2.4.4 Les gardes ... 19
2.4.5 La forme `fun` ... 19
2.4.6 Les déclarations de fonctions nommées ... 20
2.4.7 Les applications de fonction et les opérateurs ... 20
2.4.8 La forme `case` ... 21
2.4.9 L'opérateur de match ... 22
2.4.10 La forme `begin` ... 22
2.4.11 La forme `if` ... 23
2.4.12 La forme `cond` ... 23
2.4.13 Les formes `some_true` et `all_true` ... 23
2.4.14 La forme `catch` ... 24
2.4.15 La forme `try` ... 24
2.4.16 La forme `receive` ... 25
2.5 Simplification du langage ... 26
Chapitre III Réduction du langage Erlang ... 27
3.1 Situation dans le processus global de compilation ... 29
3.2 Données ... 29
3.3 Formes syntaxiques ... 29
3.3.1 Constantes et variables ... 29
3.3.2 Patrons ... 30
3.3.3 Filtrage de patrons: `ecase` ... 30

3.3.4 Application de fonction 30
3.3.5 Capture d'exception: ucatch 30
3.3.6 Définition de fonctions anonymes: afun 31
3.3.7 Déclaration de fonctions nommées 32
3.4 Compilation de Erlang vers UE 32
3.4.1 Les déclarations de fonctions nommées 33
3.4.2 La forme case 33
3.4.3 L'opérateur de match 33
3.4.4 La forme begin 34
3.4.5 La forme if 34
3.4.6 La forme cond 35
3.4.7 La forme some_true 35
3.4.8 La forme all_true 36
3.4.9 La forme catch 36
3.4.10 La forme try 37
3.4.11 La forme receive 37
3.4.12 La forme fun 38
3.4.13 Les opérateurs autres que l'opérateur de match 39
3.4.14 Les patrons 39
3.4.15 Les gardes 39
3.5 Propriétés du langage UE 40
3.5.1 Centralisation du filtrage de patrons 40
3.5.2 Centralisation des liaisons de variables 40
3.5.3 Brèche de sécurité 41
3.5.4 Aucun traitement d'exception implicite 42
3.6 Mode de compilation non sécuritaire 43
3.7 UE comme langage de transition de Erlang vers Scheme 45
Chapitre IV UE vers Scheme: implantation efficace du filtrage 46
4.1 Situation dans le processus global de compilation 47
4.2 Passage de UE à Scheme : la transformation μ 47

4.2.1 Système de gestion des exceptions 47
4.3 Compilation des formes syntaxiques UE 48
4.3.1 Constantes et variables 48
4.3.2 Application de fonction 48
4.3.3 La forme `ucatch` 49
4.3.4 La forme `afun` 50
4.3.5 Déclaration de fonctions nommées 50
4.3.6 La forme `ecase` 51
4.4 Traitement du filtrage 52
4.4.1 Le filtrage de patrons Erlang versus l'unification de Prolog 52
4.5 Liaison des variables 53
4.5.1 Le statut des variables 53
4.5.2 Le moment de liaison et les patrons non linéaires 54
4.6 Gestion du flot de contrôle 57
4.6.1 La duplication de code 58
4.6.2 La fusion des expressions communes 59
4.6.3 Évaluation efficace des conditions : la prédiction des tests 60
4.6.4 L'ordre des tests 61
4.7 Vers des solutions 62
Chapitre V La prédiction des tests 64
5.1 Les expressions de test 65
5.1.1 Tests sur les types de données Scheme de Gambit 66
5.1.2 Sémantique des expressions de test 67
5.1.3 Syntaxe 67
5.1.3.1 Constantes 67
5.1.3.2 Tests 67
5.1.3.3 L'opérateur `bind` 67
5.1.3.4 L'opérateur `bnd?` 68
5.1.3.5 Opérateurs `and` et `all` 68
5.1.3.6 Opérateurs `or` et `one` 68

5.1.3.7 Opérateur `not` 68

5.1.3.8 Forme `if3` 68

5.1.3.9 Opérateurs `kt` et `kf` *(known true, known false)* 69

5.1.3.10 La forme `any` 69

5.1.3.11 L'opérateur `exec` 70

5.1.4 Règles de construction 71

5.1.4.1 L'expression de test des liaisons 72

5.1.4.2 L'expression de test de l'acceptation du sélecteur 73

5.1.5 Extraction des liaisons par réduction des expressions L 75

5.1.6 Extraction des tests par évaluation partielle des expressions A 75

5.2 L'arbre de décisions 76

5.2.1 Génération 77

5.2.2 Retrait de la duplication de code 79

5.2.3 Fusion des expressions communes 79

5.3 Prédictions 80

5.3.1 La prédiction des expressions de test composées 81

5.3.1.1 L'opérateur `bnd?` 81

5.3.1.2 Opérateurs `all` et `one` 81

5.3.1.3 Opérateurs `and` et `or` 82

5.3.1.4 Opérateur `not` 83

5.3.1.5 Forme `if3` 83

5.3.1.6 La forme `any` 83

5.3.2 La prédiction des tests 83

5.4 L'expansion des tests 84

5.4.1 Tests de base 84

5.4.2 Tests de comparaison et sur structures complexes 85

5.4.2.1 Comparaisons 86

5.4.2.2 Structures complexes 86

5.4.3 Génération des expression de test spéciales des gardes 87

5.5 Résumé 88

Chapitre VI L'ordre des tests 89
6.1 Étalons utilisés 90
6.2 Première stratégie 91
6.3 Seconde stratégie 91
6.3.1 Objectif 91
6.3.2 Heuristique de convenance (*relevance*) 92
6.3.3 Heuristique du facteur de branchement (*branching factor*) 94
6.3.4 Heuristique du facteur d'arité (*arity factor*) 100
6.3.5 Heuristique du facteur de complexité 100
6.3.6 Le problème des listes 102
6.4 Résultats comparatifs 103
Conclusion 105
Remerciements 107
Annexe 1 Grammaire du langage UE 108
Annexe 2 Programmes étalons 110
`epoptest.erl` 111
`litescm.erl` 112
`poker.erl` 116
`type.erl` 118
Annexe 3 Programme d'exemple 119
`epop.erl` 120
Références 130

Liste des figures

Figure 2.1: Le processus global de compilation 10
Figure 2.2: Constantes Erlang vs constantes Scheme 18
Figure 2.3: Patrons 19
Figure 2.4: Patrons augmentés 19
Figure 3.1: Phase de réduction 29
Figure 4.1: Compilation vers Scheme 47
Figure 5.1: Hiérarchie des tests sur les données 66
Figure 5.2: Résumé des fonctions de transformation 72
Figure 5.3: Règles de construction des expressions L 72
Figure 5.4: Règles de construction des expressions A à partir des patrons 73
Figure 5.5: Règles de construction des expressions A à partir des gardes 73
Figure 5.6: Fonctions de construction des expressions de garde 74
Figure 5.7: Règle de construction des expressions A 74
Figure 5.8: Règles d'extraction des tests 75
Figure 5.9: Règles d'extraction des tests 76
Figure 5.10: Arbre de décision généré 78
Figure 5.11: Exemple de table de nœuds 79
Figure 6.1: Arbre de décision optimal, langage statiquement typé 96
Figure 6.2: Clauses candidates après le choix du test et son résultat 97
Figure 6.3: Clauses candidates après le choix du test et son résultat 98
Figure 6.4: Arbre de décision optimal 99
Figure 6.5: Arbre de décision non optimal 99
Figure 6.6: Règles de complexité des tests 101
Figure 6.7: Règles de complexité des termes 102
Figure 6.8: Résultats comparatifs des deux stratégies 103

Introduction

Le langage Erlang [AVW96] a récemment été développé par Joe Armstrong, Robert Virding et Cläes Wikström au sein de la compagnie de télécommunication suédoise Ericsson. Étant un langage très récent, Erlang ne possède que très peu de compilateurs présentement disponibles sur le marché. Il est donc naturel que les implantations existantes puissent être grandement améliorées.

De son coté, Scheme est un langage bien établi et il existe déjà plusieurs implantations du langage sur le marché [ISR99] dont certaines semblent surpasser de loin les meilleurs compilateurs Erlang. C'est pourquoi il devient intéressant de compiler Erlang vers Scheme. Nous avons développé un tel compilateur à l'Université de Montréal, que nous avons appelé Etos et dont les résultats sont très prometteurs [FL97], particulièrement lorsqu'utilisé conjointement avec le compilateur Gambit-C [GC98].

Une grande difficulté dans la compilation de Erlang vers Scheme est l'implantation efficace du filtrage de patrons. En effet, le contrôle de flot conditionnel est entièrement réalisé par filtrage sous Erlang. Cette fonctionnalité n'étant pas présente dans Scheme, il devient impératif de l'implanter dans le compilateur. L'implantation efficace du filtrage est importante en ce qui concerne l'efficacité globale du code généré puisque chaque décision prise lors de l'exécution en dépend.

Ce mémoire porte donc en grande partie sur l'implantation efficace du filtrage de patrons sous Erlang. Notre travail se compose de deux grands volets: la réduction du langage Erlang et la compilation vers Scheme du filtrage Erlang. Il s'agit en fait de la description du processus de compilation que nous avons développé pour améliorer Etos. Il s'agit en premier lieu de clairement identifier et d'isoler les constructions sujettes au filtrage et ensuite de passer à la compilation proprement dite.

Le premier chapitre constitue une introduction au langage fonctionnel Erlang. Nous y décrivons les principales caractéristiques du langage ainsi que sa syntaxe.

Le second chapitre présente une comparaison du langage Erlang avec le langage Scheme afin d'illustrer les similitudes entre les deux langages tout en soulignant quelques aspects qui les différencient.

Le troisième chapitre traite de la réduction du langage Erlang vers un sous-langage de niveau plus bas. Nous y verrons comment cette réduction permet de centraliser le filtrage de patrons sous une seule forme syntaxique. Nous verrons aussi que le sous-langage possède d'autres propriétés intéressantes dû au fait que le traitement d'erreur n'y est plus implicite.

Le quatrième chapitre est consacré à la compilation efficace du sous-langage vers Scheme. L'attention sera portée sur la compilation de l'unique forme syntaxique permettant le filtrage. Nous y présenterons une analyse des différents aspects entourant la problématique du filtrage.

Le cinquième chapitre traite du problème de la prédiction des tests par la conservation de l'information recueillie à l'issue des tests effectués. Nous y verrons comment les règles régissant les liens structurels des données peuvent être explicités au compilateur par l'utilisation des expressions de tests.

Le sixième chapitre traite de l'élaboration d'heuristiques permettant l'optimisation de l'arbre de décision par le choix approprié de l'ordre d'évaluation des tests. Nous adapterons l'algorithme de Baudinet et MacQueen aux fonctionnalités spécifiques du langage Erlang. Des résultats intéressants en terme de performance et de taille du code généré y sont aussi présentés.

Chapitre I

Introduction au langage Erlang

Ce chapitre présente une brève introduction au langage Erlang. Nous parlerons d'abord de la classification du langage et présenterons ses types de données et sa syntaxe ainsi que ses propriétés particulières. Le deuxième chapitre aborde la syntaxe plus en détail afin de comparer le langage Erlang au langage Scheme.

1.1 Classification du langage

Erlang est un langage fonctionnel concurrent qui fut développé chez Ericsson dans un contexte de télécommunication. Le langage n'est pas purement fonctionnel puisqu'il supporte les processus concurrents ainsi que la communication entre ces processus. La gestion de la mémoire y est automatisée avec un *garbage collector*. Erlang est un langage basé sur les expressions, possède une portée lexicale, implante de façon efficace la récursivité terminale, est typé dynamiquement et passe ses paramètres par valeur. Il est clair que le langage partage de nombreuses ressemblances avec le langage Scheme.

1.2 Les types de donnée

Le langage Erlang offre plusieurs types de donnée qui sont : les nombres (les entiers sans limite de précision ainsi que les nombres à virgule flottante), les caractères (Unicode est supporté), les atomes (comme les symboles de Scheme), les listes, les tuples (comme les vecteurs de Scheme), les fonctions, les ports (un canal de communication avec les processus externes et le système d'exploitation), les PIDs (les identificateurs de processus), les références (des identificateurs globalement uniques) et les binaires (des tableaux de bytes). L'arithmétique de Erlang est générique (n'importe quel mélange de nombres) et les opérateurs de comparaison peuvent comparer n'importe quel mélange de types.

1.3 La syntaxe

La syntaxe de Erlang est inspirée par celle de Prolog [COL82]. Par exemple, `[x,y,z]`, `[]` et `[H|T]` dénotent des listes, les variables commencent avec une lettre en majuscule et les atomes avec une lettre minuscule, le filtrage de patrons est utilisé pour définir les fonctions et extraire les données des structures. Erlang n'offre pas d'unification complète comme Prolog. En effet, une variable n'est pas un objet qui peut être contenu

dans une donnée. Ceci implique par ailleurs que le *backtracking* n'est pas nécessaire sous Erlang et que l'implantation du filtrage de patrons sous Erlang soit différente que celle de l'unification sous Prolog.

Il est à noter que les patrons peuvent être augmentés avec des gardes, permettant de contraindre davantage le filtrage. Le seul moyen de lier une variable à une valeur est d'utiliser le filtrage de patrons (dans les paramètres d'une fonction et dans plusieurs autres constructions qui seront présentées plus loin). La portée des liaisons inclue l'ensemble des expressions qui la suivent et est bornée par le corps de la définition d'une fonction.

Voici un exemple simple de définition de la fonction `member` qui démontre l'utilisation du mécanisme de filtrage de patrons pour la définition de fonctions et la liaison des variables :

```
member(X, [X|_]) -> true;
member(X, [_|T]) -> member(X,T);
member(_, [])    -> false.
```

Voici maintenant un autre exemple qui démontre comment l'augmentation des patrons par des gardes permet de définir la fonction de Fibonacci :

```
fib(X) when X<2 -> X;
fib(X)          -> fib(X-1) + fib(X-2).
```

1.4 Propriétés particulières

Puisque Erlang a été développé dans un contexte de télécommunication, il possède plusieurs propriétés intéressantes qui lui permettent de se démarquer sur plusieurs plans par rapport aux autres langages. Nous présentons ici les principales propriétés du langage.

1.4.1 Modularité

Erlang supporte un système simple de modules qui fournit la gestion des espaces de noms. Chaque module spécifie son nom, l'ensemble des fonctions qu'il exporte et les fonctions qu'il importe des autres modules. La forme `lists:map` indique la fonction `map` du module `lists`.

1.4.2 Concurrence

Le langage Erlang est un langage concurrent. Les processus locaux et distants sont créés dynamiquement avec la fonction `spawn` et interagissent en envoyant des messages (des données Erlang) vers leurs boites aux lettres qui mettent en série et prennent en charge la mise en mémoire tampon des messages qui arrivent. Les messages sont drainés de façon asynchrone depuis la boite aux lettres avec la forme spéciale `receive` qui extrait de la boite aux lettre le prochain message qui est accepté par le patron spécifié dans la construction (une période d'attente limite peut aussi être spécifiée).

Voici un exemple d'utilisation des mécanismes de concurrence sous Erlang :

```
-module(m). % Déclaration d'un nom de module
client(Pid) -> receive
                   {Pid,end} ->   io:write("finish"),
                                  io:nl()
                   {Pid,data,X} -> io:write(X),
                                   io:nl(),
                                   client(Pid);
               end.
server() -> ClientPid = spawn(m,client,[]),
            {ClientPid,data,"Hello world"} ! ClientPid,
            {ClientPid,end} ! ClientPid.
```

Ici, la fonction `server` créé un processus avec la fonction `spawn` qui appelle la fonction `client`. Ensuite, deux messages sont mis dans la boîte aux lettres de ce processus avec l'opérateur d'envoi `!`. De son coté, le processus qui applique la fonction `client` retire les messages qui lui sont envoyés en filtrant sa boîte aux lettres grâce à la construction `receive`. Les messages sont reçus dans l'ordre et le programme affiche donc `"Hello world"` suivi de `"finish"`.

1.4.3 Distribution

Erlang est un système pouvant être distribué sur plusieurs machines faisant partie d'un même réseau. Les données de Erlang peuvent donc être transmises d'un système à un autre. La distribution sous Erlang est implantée dans certaines fonctions prédéfinies qui permettent d'accéder aux autres systèmes. Aucune forme syntaxique ne réfère à la

distribution. Tout se déroule donc dans un ensemble de fonctions de la librairie d'exécution. Un système Erlang isolé demeure tout de même un outil de développement très intéressant.

1.4.4 Tolérance à la faute

Le langage Erlang étant grandement destiné aux systèmes fonctionnant en temps réel, il devient alors important d'inclure une certaine tolérance à la faute afin d'empêcher que les erreurs puissent figer le système. Il peut souvent être très coûteux que de seulement réinitialiser un système. Nous n'avons qu'à penser à un relais téléphonique peu accessible, où l'intervention humaine implique des coûts importants. La tolérance à la faute est donc très ancrée dans la syntaxe de Erlang. L'ensemble des erreurs pouvant survenir lors de l'exécution de code Erlang soulève des exceptions qui peuvent être gérées à même l'application générée. Ainsi, l'application gérant le relais pourrait décider de réinitialiser son système si jamais elle rencontre un problème grave, détruire le processus mal en point ou simplement faire un rapport à la centrale pour d'éventuelles corrections au logiciel.

Les exceptions sont gérées en utilisant les constructions `throw`, `exit`, `catch` et `try`. L'évaluation de `throw` *X* transfert le contrôle au `catch` englobant qui est dynamiquement le plus proche et qui retourne *X*. Des exceptions prédéfinies existent pour les BIFs(*Built-In Functions*).

Voici une nouvelle version de la fonction `member` qui retourne la portion de la liste depuis la première occurrence de l'élément recherché jusqu'à la fin:

```
memberlist(X,L=[X|_])  -> L;
memberlist(X, [_|T])   -> member(X,T);
memberlist(_, [])      -> throw(nomatch).
```

Cette fonction pourrait être utilisée à l'intérieur d'un catch pour traiter le cas d'erreur:

```
member(X,Y) -> case catch memberlist(X,Y)
                 nomatch -> false;
                 _ -> true
               end.
```

1.5 De Erlang vers Scheme

Ce chapitre a présenté le langage Erlang de façon sommaire. Un exemple d'application Erlang est présenté à l'annexe 3. Le prochain chapitre entre dans les détails de la syntaxe en comparant Erlang à Scheme. Cette comparaison est justifiée par le fait que c'est vers le langage Scheme que nous voulons compiler Erlang.

Chapitre II

Erlang comparé à Scheme

Le langage Erlang fut développé chez Ericsson dans un contexte de télécommunication. Erlang a été conçu pour faciliter la programmation de systèmes distribués, concurrents, temps réel et étant tolérants à la faute. Il s'agit d'un langage fonctionnel dont la syntaxe rappelle celle des langages ML et Prolog. Puisque les langages fonctionnels permettent la création de programmes hautement structurés [HUG90], des gains considérables en terme de temps d'implantation peuvent être réalisés par leur utilisation. Nous décrivons ici plus en détail le langage Erlang en le comparant à un autre langage fonctionnel: Scheme, qui est un langage probablement mieux connu.

2.1 Situation dans le processus global de compilation

Erlang est le langage à la source du processus global de compilation. Notre but ultime est la génération de code natif très performant. Notre solution se décompose en quatre phases de compilation décrites par la figure 2.1.

```
Erlang → UE → Scheme → C → Code natif
```

Figure 2.1: Le processus global de compilation

Nous procéderons donc à la compilation de Erlang vers le code natif en passant par plusieurs langages différents. Le programme Erlang sera d'abord compilé vers le langage intermédiaire UE. Cette phase de compilation, ainsi que la description du langage UE, feront l'objet du troisième chapitre. Par la suite, le programme UE sera compilé vers le langage Scheme. Cette phase sera décrite au quatrième chapitre. Ces deux phases, compilant Erlang vers Scheme, constituent donc le compilateur Etos.

La compilation du programme Scheme vers le langage C est réalisée grâce au compilateur Gambit-C de Marc Feeley [GC98]. Finalement, la compilation du programme C vers le code natif est assurée par un compilateur C tel que `gcc`.

2.2 Implications des propriétés particulières de Erlang

Nous présentons ici les implications des principales propriétés du langage dans un contexte de compilation vers Scheme.

2.2.1 Modularité

Scheme n'offre pas de gestion des espaces de noms via les modules comme le fait Erlang. Afin de palier à ce manque, chaque module qui est compilé avec Etos définit l'ensemble des fonctions qu'il exporte en préfixant leurs noms par celui du module. Ainsi, si le module `lists` exporte la fonction `map`, le nom de la fonction au niveau global sera `lists:map`.

2.2.2 Concurrence

Une bonne implantation de Erlang doit produire un système efficace pour basculer le contexte d'exécution afin de permettre au programmeur de créer des centaines, voire des milliers de processus pouvant s'exécuter de façon concurrente.

Bien que le langage Scheme ne soit pas reconnu comme un langage concurrent, sa forme spéciale `call-with-current-continuation`, souvent notée `call/cc`, permet de capturer le contexte d'exécution courant et peut ainsi être utilisée pour implanter un système de partage de tâches. Il devient donc impératif que l'implantation Scheme sous-jacente au compilateur Etos soit munie d'une implantation très efficace du mécanisme de capture de contexte.

2.2.3 Distribution

Les données de Erlang peuvent être transmises d'un système à un autre et ceci impose des considérations spéciales quant à la représentation de certains types de données. Scheme n'a pas été conçu comme un système permettant la communication sur un réseau. En fait, Scheme possède une interface très limitée vers l'extérieur ; son interface vers le système d'exploitation sous-jacent se limite aux fichiers et à la console. Bien que cette interface soit suffisante pour implanter un système distribué, elle en demeure très peu attrayante vue son évidente inefficacité. Il devient alors important que l'implantation

Scheme sous Etos permette un accès moins limité vers l'extérieur. Idéalement, ce système posséderait une interface d'appel de fonctions étrangères (écrites en C par exemple), comme Gambit-C le permet.

2.2.4 Tolérance à la faute

Scheme n'est pas un système tolérant à la faute. Une erreur à l'exécution ne peut être capturée puis traitée. Nous devons donc implanter les mécanismes nécessaires à la vérification de toutes les possibilités d'erreur à même les applications. Le soulèvement d'exception peut se faire grâce à la forme spéciale `call/cc`, comme nous le verrons dans un prochain chapitre.

Le fait que les mécanismes de tolérance soient inclus dans le code de l'application compilée peut provoquer un dédoublement d'effort puisqu'il devient alors impossible que le code généré produise des erreurs au niveau Scheme. L'implantation Scheme utilisée avec Etos a donc grand avantage à permettre une compilation non sécurisée, c'est-à-dire sans vérification d'erreur à l'exécution. Le compilateur Gambit-C possède cette option, ce qui permettra par exemple d'évaluer l'application `(car x)` sans vérifier (à nouveau) si `x` est bien une paire.

2.3 Représentation des données

Nous décrivons ici les différents types de données disponibles sous Erlang. Cette description sera faite en montrant les choix que nous avons faits afin de les représenter sous le langage Scheme.

2.3.1 Entiers

Les entiers de Erlang doivent être représentés sur au moins 60 bits. Les entiers Scheme peuvent aussi être limités, mais la plupart des compilateurs offrent des entiers de taille très grande (bignums). Il devient donc naturel de représenter les entiers Erlang avec les entiers exacts de Scheme.

2.3.2 Caractères et chaînes de caractères

Standard Erlang possède un type de donnée pour les caractères qui est compatible avec celui de Scheme. Les procédures `integer->char` et `char->integer` peuvent être utilisées pour créer et manipuler les caractères Unicode. Nous utilisons donc les caractères de Scheme pour représenter les caractères de Erlang.

Les chaînes de caractères en Erlang sont représentées sous forme de listes de caractères. Il n'est donc pas nécessaire de les traiter spécialement. Ainsi, le type de chaînes de caractères en Scheme reste inutilisé.

2.3.3 Nombres à virgule flottante

Les nombres à virgule flottante de Erlang correspondent naturellement aux nombres réels inexacts de Scheme.

2.3.4 Atomes

Les atomes de Erlang sont des données qui ne peuvent être distinguées que par leur nom. Par exemple, `foo` et `bar` sont des atomes. Le type de données correspondant en Scheme est le symbole. De plus, Erlang exige qu'une comparaison de deux atomes se fasse en temps constant, ce qui est respecté en Scheme avec la procédure Scheme `eq?`.

Un problème potentiel avec l'utilisation des symboles Scheme est le fait que la casse ne doit pas être retenue dans le nom du symbole. Par exemple, une implantation de Scheme peut considérer les symboles `foo`, `fOO` et `FoO` équivalents. Une solution simple peut être utilisée pour les implantations de Scheme qui n'offrent pas de sensibilité à la casse: le préfixage des lettres majuscules par un symbole d'échappement, tel le caractère `'^'`. Ainsi, les trois symboles précédents sont représentés par les symboles distincts `foo`, `f^o^o` et `^fo^o`.

2.3.5 Listes

Les listes de Erlang sont en tout point semblables à celles de Scheme. Elles sont composées de la liste vide dénotée `[]` et de paires formées avec la construction `[X|Y]`, où

`Y` peut être une liste ou une donnée quelconque. Dans ce dernier cas, la liste est dite impropre. Seule la syntaxe diffère dans le cas de Scheme, où la liste vide est dénotée `'()` et les paires sont formées avec un appel à une primitive: `(cons X Y)`.

2.3.6 Tuples

Les tuples de Erlang sont utilisés pour représenter des données composées à accès indexé. L'équivalent naturel en Scheme est bien entendu le vecteur. Bien que la seule différence entre les tuples de Erlang et les vecteurs de Scheme soit l'origine de l'indexage qui est de un pour Erlang et zéro pour Scheme, un problème plus important survient. En effet, les listes et les vecteurs étant les seules constructions utiles pour représenter des données composées, elles doivent aussi être utilisées pour représenter d'autres types tels les fonctions et les binaires. Le problème peut être réglé en sacrifiant le premier élément pour y inscrire le type de donnée concerné. Par exemple, le tuple Erlang `{1,2,3}` sera représenté en Scheme par le vecteur `#(tuple 1 2 3)`. Bien que cette représentation étiquetée ralentisse les tests de type sur les tuples, elle permet de régler le problème de l'origine de l'indexation.

2.3.7 Fonctions

Les objets représentant des fonctions en Erlang permettent la création de fermetures lors de l'exécution. Lorsqu'une fonction est appelée, l'arité de la fonction doit être vérifiée pour éventuellement signaler une erreur. Puisqu'en Scheme il n'y a pas de moyen standard d'obtenir l'arité d'une procédure ni de capturer les erreurs Scheme à l'exécution, la procédure Scheme n'est pas suffisante pour représenter les fonctions Erlang. Ces dernières sont donc représentées à l'aide d'un vecteur étiqueté comprenant une fonction Scheme ainsi qu'un entier correspondant à l'arité de la fonction.

2.3.8 Références

Erlang possède un type de donnée pour représenter un identificateur unique. Cet identificateur n'est utile que pour effectuer des comparaisons. Puisque Erlang est un système conçu pour être utilisé de manière distribuée, les informations à conserver dans la référence doivent tenir compte de plusieurs facteurs. Premièrement, puisqu'un système

peut créer plusieurs références, on doit tenir à jour un compteur qui sera en quelque sorte l'identificateur local au système. Deuxièmement, puisqu'il est possible d'avoir plus d'un système Erlang en marche sur une même machine la référence doit aussi contenir l'identificateur local à la machine (cet identificateur est attribué par un daemon dès la mise en marche du système). Finalement, puisqu'il est possible d'avoir plusieurs systèmes Erlang distribués sur un réseau, la référence doit aussi contenir un identificateur global qui est le nom réseau de la machine. Donc, pour représenter une référence en Scheme, ces trois informations sont stockées dans un vecteur étiqueté.

2.3.9 Identificateurs de processus (PIDs)

Erlang étant un langage concurrent, il possède un type de donnée spécial pour représenter un processus. Le descripteur de processus possède plusieurs propriétés telles que sa file de messages, son dictionnaire de données, sa priorité, etc. L'identificateur permet de localiser le descripteur. La représentation des identificateurs de processus en Scheme ressemble beaucoup à celle des références. Nous le représentons donc par la valeur d'un identificateur entier local au système, un identificateur local à la machine et le nom réseau de cette dernière dans un vecteur étiqueté.

Cependant, puisqu'il n'y a aucune limite sur le nombre de processus pouvant être créés simultanément, la recherche du descripteur de processus à partir de la valeur du compteur ne peut être atomique. Nous avons donc décidé d'ajouter un pointeur sur le descripteur dans la représentation de l'identificateur. Le problème avec cet ajout vient du fait qu'il est possible d'envoyer et de recevoir des identificateurs de processus. En effet, le système de collection peut relocaliser le descripteur et si un identificateur y faisant référence se retrouve sur un autre système Erlang, ce dernier devient alors invalide et pourrait être renvoyé au système initial un peu plus tard. Ce problème peut par contre être réglé en remplaçant le pointeur par la donnée `#f` (faux) de Scheme lorsqu'un identificateur de processus est créé lors du décodage du format externe de Erlang, par lequel doivent passer toutes les données reçues de l'extérieur. Par la suite, lors de l'utilisation d'un identificateur pour l'envoi d'un message ou d'un signal, un simple test permet de savoir s'il faut rechercher le processus à l'aide de la valeur du compteur.

2.3.10 Ports

Afin de communiquer avec l'environnement d'exécution du système et ainsi pouvoir permettre l'utilisation de pilotes écrits dans un autre langage comme le C, le langage Erlang possède un type de donnée permettant de représenter un port de communication vers cet environnement. Comme les descripteurs de processus, les descripteurs de ports possèdent plusieurs propriétés. Ces propriétés définissent entre autres le mode de communication et le processus propriétaire du port. Puisque le nombre de ports ouverts est limité dans un système Erlang, il est possible d'accéder au descripteur de port directement à l'aide de la valeur d'un identificateur entier qui est alors un index dans une table de taille fixe, ce qui assure une localisation atomique. Les ports peuvent donc être représentés en Scheme par un vecteur étiqueté contenant ces informations.

2.3.11 Binaires

Erlang permet l'utilisation de tuples spécialisés afin de représenter efficacement les tableaux de bytes fréquemment utilisés dans un contexte de communication. Scheme ne possède aucune construction permettant de représenter efficacement ce type de donnée. Cependant, certaines implantations de Scheme telle Gambit-C possèdent un tel type spécialisé. Avec Gambit-C, il est possible d'utiliser la famille de procédures suivantes pour manipuler de tels vecteurs:

```
(make-u8vector n)        ; Création d'un vecteur d'octets
(u8vector-ref v i)       ; Accès à un octet
(u8vector-set! v i b)    ; Modification d'un octet
(u8vector-length v)      ; Obtenir la taille du vecteur
```

Ces procédures ont donc la même structure que celles se rapportant aux vecteurs réguliers. Une implantation de Scheme ne possédant pas de vecteurs spécialisés pourrait implanter Etos en invoquant plutôt les procédures sur les vecteurs réguliers correspondants et ainsi pouvoir demeurer fonctionnel quoique moins efficace.

Il serait tentant d'utiliser ce type de donnée directement afin de représenter les binaires Erlang. Cependant, Erlang demande que les binaires puissent être scindés en deux en temps constant puisqu'ils sont grandement utilisés dans les protocoles de

communication, ce qui nous empêche d'utiliser directement le vecteur spécial de Gambit-C. En effet, la seule façon de scinder un vecteur spécial de Gambit-C en deux est d'en créer deux nouveaux et d'y copier les contenus appropriés, ce qui n'est pas O(1).

Notre solution utilise un vecteur étiqueté contenant un pointeur sur un vecteur spécial de Gambit-C, la taille du binaire ainsi que la position de son premier octet dans le vecteur spécial. Ceci permet d'effectuer la césure de façon atomique puisque les deux binaires ainsi créés ont le même pointeur que le binaire d'origine et seulement leurs tailles et leurs positions sont changées. Cette représentation pose cependant un problème de *garbage collection* lorsqu'une partie seulement d'un vecteur spécial est référencée. En effet, le vecteur spécial sera conservé en entier jusqu'à ce que l'ensemble de ses parties ne soient plus référencées. Ce problème, qui est aussi présent dans la version OTP de Ericsson, reste encore ouvert à ce jour.

2.4 Formes syntaxiques

Nous décrivons ici les diverses formes syntaxiques du langage Erlang. Il ne s'agit que d'une brève introduction au langage afin de donner une idée générale de ses possibilités. Nous procéderons à une analyse plus poussée de ces formes syntaxiques dans un chapitre suivant.

2.4.1 Constantes

Les constantes de Erlang permettent de spécifier des données statiques immuables. Les valeurs des entiers, des caractères, des nombres à virgule flottante, des atomes ainsi que les paires et les tuples composés de constantes peuvent être représentés sous forme de constante Scheme. La syntaxe de ces constantes dépend du type de donnée de la valeur. Voici quelques exemples :

Type de donnée	Erlang	Scheme
Entier	`-1234`	`-1234`
Caractère	`$a`	`#\a`
Nombre flottant	`-1.4e3`	`-1.4e3`
Atome	`t12` `'+'`	`'t12` `'+`
Liste vide	`[]`	`'()`
Paire	`[1|2]`	`'(1 . 2)`
Chaînes de caractères	`"abc"` `[$a,$b,$c]`	`'(#\a #\b #\c)`
Tuple	`{1,2,3}`	`#(1 2 3)`

Figure 2.2: Constantes Erlang vs constantes Scheme

2.4.2 Variables

Les variables de Erlang sont liées à des valeurs à l'exécution. Il est important de noter que la liaison est permanente puisque Erlang ne permet pas l'affectation. Les variables sont dénotées dans les programmes par des identificateurs dont la première lettre est une majuscule. Par exemple, `Var` dénote une variable, mais `var` dénote un atome.

2.4.3 Les patrons

Les patrons dénotent la structure de donnée à laquelle doit se conformer une valeur afin d'être filtrée. Le processus de filtrage permet essentiellement de déterminer si une valeur est conforme à un patron ou non. Par exemple, un patron composé d'une constante ou d'une variable liée n'accepte que les valeurs égales à celle dénotée par la constante ou la variable.

Les patrons constituent par ailleurs le site de liaison des variables libres. En effet, un patron composé d'une variable libre accepte n'importe quelle valeur et la valeur ainsi acceptée est alors liée à la dite variable. Il est toutefois possible d'accepter toutes les valeurs sans effectuer de liaison avec le patron universel, dénoté par le caractère de soulignement `'_'`.

Les patrons permettent aussi un filtrage sur les données composées. Par exemple, il est possible de spécifier un patron qui n'accepte que les paires ou les tuples de longueur donnée. Le processus de filtrage s'effectue donc aussi dans les éléments de la structure afin

de s'assurer qu'ils sont aussi tous acceptés. Finalement il est possible d'effectuer des liaisons et du filtrage de données composées à l'aide du patron de match, dont la syntaxe est `<patron1> = <patron2>`. Voici quelques exemples de patrons :

Syntaxe du patron	Sémantique du patron
`12`	accepte l'entier `12`.
`[a|X]`	accepte une paire dont le premier élément est l'atome `a` et lie la variable `X` à son deuxième élément.
`X = {_,_}`	accepte un 2-tuple et le lie à la variable `X`.

Figure 2.3: Patrons

2.4.4 Les gardes

Les gardes sont composés d'une liste d'expressions s'évaluant aux atomes `true` ou `false`. Il s'agit de contraintes supplémentaires pouvant être associées aux patrons afin de raffiner le filtrage. Les expressions pouvant se retrouver dans les gardes sont très limitées et il est garanti qu'aucune de ces expressions ne produit d'effet de bord et que toutes s'exécutent en temps constant. Les formes les plus utilisées d'expressions de gardes comprennent la détection de type, comme par exemple le filtrage des entiers ; et les bornes numériques, comme par exemple le filtrage des nombres plus grands que 100. Voici quelques exemples de patrons augmentés de gardes:

Syntaxe du patron	Sémantique du patron
`X when is_char(X)`	accepte un caractère.
`X when X>0 , X<10`	accepte les nombres entre 0 et 10 exclusivement.
`{X,Y} when X<Y`	accepte un 2-tuple dont le premier élément est inférieur au deuxième.

Figure 2.4: Patrons augmentés

2.4.5 La forme `fun`

Erlang permet la création de fonctions anonymes grâce à la forme `fun`. Une fonction est alors définie par un ensemble de clauses qui serviront à sélectionner le corps à évaluer lors de l'application. La syntaxe de cette forme est la suivante:

```
fun
  (<P11>,…,<P1a>) [when <G1>] -> <B1>;
  …
  (<Pn1>,…,<Pna>) [when <Gn>] -> <Bn>
end
```

Cette forme définit une fonction acceptant a arguments. L'évaluation de cette forme retourne donc une fonction. Lorsqu'appliquée, cette fonction sélectionnera la première clause dont les arguments seront acceptés à la fois par les patrons et le garde et retournera l'évaluation du corps correspondant. Si aucune clause n'accepte les arguments ou si le nombre d'arguments fournis n'est pas égal à `a`, une exception est lancée.

2.4.6 Les déclarations de fonctions nommées

Un programme Erlang est constitué d'un ensemble de définitions de fonctions nommées, qui peuvent par la suite être appliquées. La déclaration de ces fonctions se fait via la syntaxe suivante qui décompose la fonction en clauses:

```
f(<P11>,…,<P1a>) [when <G1>] -> <B1>;
…
f(<Pn1>,…,<Pna>) [when <Gn>] -> <Bn>.
```

La syntaxe est très semblable à celle de la forme `fun`, à l'exception que le nom de la fonction est explicitement donné. Une telle déclaration rencontrée dans un module `m` permet de spécifier que le nom `m:f/a` est associé à la fonction définie par les clauses. Une exception est lancée lors de l'appel si aucune clause n'accepte les arguments. La vérification du nombre d'arguments se fait toutefois avant l'exécution, car le lien entre le nom de fonction et la fonction est connu au chargement du module. La syntaxe des déclarations de fonctions nommées implique que toutes les clauses d'une même fonction doivent être contiguës dans un programme Erlang. Elles sont séparées par des point virgules et terminées par un point.

2.4.7 Les applications de fonction et les opérateurs

L'appel de fonction a une syntaxe préfixe assez standard:

```
<F>(<E1>,…,<En>)
```

Si <*F*> est un nom de fonction déclaré, la fonction correspondante est alors appelée, sinon, <*F*> est évaluée et ensuite appelée. Les arguments `<E1>,...,<En>` sont alors évalués dans un ordre indéfini et passés à la fonction appelée.

Quelques fonctions sont aussi disponibles avec la syntaxe infixe, sous forme d'opérateurs. Outres les opérateurs arithmétiques courants permettant d'effectuer des opérations de base sur les nombres, notons le l'opérateur binaire `'!'` qui permet d'envoyer un message vers un processus. Par exemple, l'expression `A ! B` permet d'envoyer le message `A` (une donnée Erlang quelconque) au processus `B`.

2.4.8 La forme `case`

La forme syntaxique `case` est une forme conditionnelle permettant d'évaluer une expression Erlang (un corps) parmi plusieurs en fonction d'une autre (le sélecteur). Ce choix est effectué par filtrage de patrons. La syntaxe du `case` est la suivante:

```
case <E> of
  <P1> [when <G1>] -> <B1>;
  ...
  <Pn> [when <Gn>] -> <Bn>
end
```

L'évaluation d'une expression `case` nécessite d'abord l'évaluation du sélecteur <*E*>, puis le filtrage de cette valeur parmi les patrons `<P1>,...,<Pn>` augmentés des gardes `<G1>,...,<Gn>`. Une fois qu'un patron `<Pi>` ainsi que son garde `<Gi>` associé acceptent la valeur, l'expression `<Bi>` est évaluée et sa valeur constitue celle de la forme `case`. L'ensemble constitué du patron augmenté du garde et du corps est dénommé « clause ». Si aucune clause n'accepte la valeur du sélecteur, une exception est lancée. Il est à noter que c'est la première clause qui accepte qui est choisie. Ceci n'implique pas pour autant qu'un ordre d'évaluation de haut en bas ne soit obligatoirement respecté puisque l'évaluation effectuée lors du filtrage ne peut engendrer d'effet de bord et que les gardes sont limités à des expressions qui s'évaluent en temps constant (sauf pour la BIFs `length/1`). Il est toutefois essentiel que les expressions faisant références à des variables libres soient

évaluées après la liaison de celles-ci. Voici un exemple d'utilisation de la forme case (tiré de [AVW96]):

```
case allocate(Resource) of
  {yes,Address} when Address > 0, Address =< Max ->
      <Sequence1>;
  no ->
      <Sequence2>
end
```

L'évaluation de cette forme se fait en évaluant d'abord l'application de la fonction allocate afin d'obtenir le sélecteur. La séquence <*Sequence1*> sera évaluée si le sélecteur est un 2-tuple dont le premier élément est l'atome yes et dont le deuxième élément est positif et inférieur à la valeur de la variable Max. Dans ce cas, la variable Address doit être liée à la valeur du deuxième élément du 2-tuple. Sinon, si le sélecteur est l'atome no, la séquence <*Sequence2*> sera évaluée. Finalement, dans les autres cas, une exception est lancée.

2.4.9 L'opérateur de match

L'opérateur de match permet de simplifier les liaisons. Sa syntaxe est la suivante:

```
<Patron> = <Expression>
```

L'expression est ici évaluée et sa valeur est filtrée par le patron. Bien que plus souvent utilisée à des fins de liaison de variable plutôt que pour des vérifications structurelles, l'évaluation de cette expression doit soulever une exception si le patron n'accepte pas la valeur.

2.4.10 La forme begin

La forme begin permet d'enchaîner des opérations. Sa syntaxe est la suivante:

```
begin <E1>,…,<En> end
```

Les expressions <*E1*>,…,<*En*> sont évaluées les unes après les autres dans l'ordre spécifié et la valeur de la dernière expression constitue la valeur de l'expression en entier.

2.4.11 La forme `if`

Erlang permet la sélection basée uniquement sur l'évaluation de gardes via la forme `if`. Sa syntaxe est la suivante:

```
if
  <G1> -> <B1>;
  ...
  <Gn> -> <Bn>
end
```

Les clauses du `if` sont donc formées d'un garde et d'un corps. C'est la première clause dont le garde s'évalue à `true` qui sera choisie. Encore ici, une exception est lancée si aucune clause ne peut être choisie.

2.4.12 La forme `cond`

La forme `cond` permet la sélection basée sur l'évaluation d'expressions booléennes quelconques. Sa syntaxe est pratiquement la même que celle de la forme `if`:

```
cond
  <E1> -> <B1>;
  ...
  <En> -> <Bn>
end
```

La première clause du `cond` dont l'expression de gauche s'évalue à `true` sera alors choisie. Une exception est lancée si la valeur l'évaluation d'une expression de gauche n'est pas booléenne ou encore si aucune clause ne peut être choisie. La forme `cond` est très utile lorsqu'une des conditions ne peut être exprimée par un garde, étant donné que l'étendue des conditions pouvant être représentées avec les gardes est très limitée.

2.4.13 Les formes `some_true` et `all_true`

Les formes syntaxiques `some_true` et `all_true`, permettent l'évaluation court-circuitée d'une série d'expressions booléennes. Leurs syntaxes vont comme suit:

```
some_true <E1>,...,<En> end
all_true <E1>,...,<En> end
```

La forme some_true s'évalue à false si toutes ses expressions constituantes s'évaluent à false et s'évalue à true dès qu'une de ces dernières s'évalue à true. Pour sa part, la forme all_true s'évalue à true si toutes ses expressions constituantes s'évaluent à true et s'évalue à false dès qu'une de ces dernière s'évalue à false. Dans les deux cas, une exception est lancée si la valeur d'une des expressions évaluées n'est pas booléenne.

2.4.14 La forme catch

La forme catch permet d'attraper une exception lancée au cours de l'évaluation d'une expression. La syntaxe est la suivante:

```
catch <E>
```

L'expression <*E*> est alors évaluée et si cette évaluation ne lance aucune exception, la valeur de la forme catch est celle <*E*>. Si une exception est lancée au cours de l'évaluation de <*E*>, l'exception est alors inspectée. Si cette exception résulte d'un appel à la fonction throw/1, l'argument de cet appel constitue la valeur de la forme catch. Sinon il s'agit d'une exception d'erreur et la raison de celle-ci est alors retournée dans un tuple de la forme {'EXIT', *x*} où *x* est une valeur représentant le contexte de l'erreur.

2.4.15 La forme try

La forme try, introduite dans Standard Erlang, est en fait une version améliorée de la forme catch. Cette forme permet une sélection plus précise quant au type d'exception à attraper. Voici sa syntaxe:

```
try <E> catch
  <P1> [when <G1>] -> <B1>;
  ...
  <Pn> [when <Gn>] -> <Bn>
end
```

Comme dans le cas de la forme catch, l'évaluation de <*E*> est d'abord effectuée et si cette évaluation ne lance aucune exception, la valeur de la forme try est celle de <*E*>. Toutefois, lorsqu'une exception est lancée, le 2-tuple représentant cette dernière est filtré par les clauses. Si aucune clause ne traite l'exception, cette dernière est lancée à nouveau.

Les exceptions lancées avec la fonction `throw/1` sont représentées par un tuple de la forme `{'THROW', x}` où *x* est l'argument de `throw/1`.

Cette discrimination basée sur l'information contenue dans l'exception permet de créer un système de gestion des exceptions à plusieurs niveaux. Les erreurs graves comme des arguments invalides peuvent alors être traités par un gestionnaire de bas niveau alors que les anomalies telles que les connexions perdues peuvent être traitées à un niveau plus élevé comme dans une application de l'usager.

2.4.16 La forme `receive`

La forme `receive` est une forme syntaxique dédiée à la réception de messages dans un contexte de programmation concurrente. Elle permet la mise en attente, pour un laps de temps donné, d'un processus en vue de la réception d'un message spécifique. Comme plusieurs autres formes, la syntaxe du `receive` utilise le filtrage:

```
receive
  <P1> [when <G1>] -> <B1>;
  ...
  <Pn> [when <Gn>] -> <Bn>
  [after <T> -> <Bt>]
end
```

Cette syntaxe permet donc de spécifier un laps de temps maximal <*T*> qui, lorsqu'expiré, indique que trop de temps s'est écoulé depuis le début de l'évaluation de la forme sans recevoir de message convenable. L'évaluation de la forme `receive` commence donc par l'évaluation de l'expression <*T*>. Lorsqu'omise, l'expression <*T*> est considérée comme étant l'atome `infinity`, qui signifie d'attendre indéfiniment le message. Une fois l'expression <*T*> évaluée, la file de message du processus courant est inspectée et chaque message passe par le filtre constitué des clauses du `receive`. Si le message est accepté, le corps correspondant est alors évalué et son résultat est celui de la forme en entier. Si aucun message de la file n'est accepté, le processus est mis en attente jusqu'à ce qu'un autre message arrive ou que le délai maximum soit expiré. Dans ce dernier cas, l'expression <*Bt*> est évaluée et sa valeur est celle de la forme `receive`.

2.5 Simplification du langage

Ce chapitre a expliqué sommairement le langage Erlang. Nous avons vu qu'il s'agit d'un langage assez riche en constructions syntaxiques et où le traitement d'erreur est implicite. Nous pouvons observer que la compilation de toutes ces formes syntaxiques vers le langage Scheme peut être grandement simplifiée par la réduction du langage Erlang.

Le prochain chapitre propose donc un langage intermédiaire qui permet d'une part de simplifier largement la syntaxe de Erlang par l'utilisation de transformations simples sur les formes syntaxiques de base du langage. Par ailleurs, nous verrons que l'explicitation des mécanismes de traitement d'erreur et la centralisation du mécanisme de filtrage contribueront aussi à cette simplification.

Chapitre III

Réduction du langage Erlang

Le processus de compilation de ETOS comprend plusieurs phases. Ce chapitre traite de la réduction du langage Erlang vers un langage de plus bas niveau. Nous avons baptisé ce langage "Unsafe Erlang" (UE). Ce sous-langage est donc plus simple que le langage Erlang en ce sens qu'il comporte beaucoup moins de constructions et ne possède aucune forme de traitement d'exception implicite. Il est alors possible d'écrire des programmes en UE qui ont un comportement imprévisible. Cependant, ce langage est utilisé à l'interne dans ETOS dans un esprit d'efficacité, le code UE généré par ETOS est entièrement sécuritaire puisque le traitement d'erreur y est introduit de façon tout à fait explicite.

Le langage *Core Erlang* [CJL99] a été développé parallèlement à l'Université d'Uppsala, dans le cadre du projet HiPE [JNP99]. *Core Erlang* réduit considérablement Erlang mais possède deux formes syntaxiques distinctes pour effectuer le filtrage. On y fait abstraction du traitement des messages, ce qui empêche la réduction de la forme syntaxique `receive`. Bref, ce langage n'est pas d'assez bas niveau pour nos besoins spécifiques dans notre contexte de compilation vers Scheme.

Nous avons conçu UE dans un esprit minimaliste, c'est-à-dire que notre objectif est l'obtention d'un langage qui contient le moins de formes syntaxiques possibles. Ceci est en accord avec la philosophie Scheme [AAB91], langage vers lequel sera éventuellement compilé UE. Nous verrons donc que UE ne possède qu'une forme syntaxique utilisant le filtrage de patrons, ce qui permet de simplifier grandement la prochaine phase de compilation dont il est question dans le quatrième chapitre.

Nous effectuons ici la réduction du langage Erlang vers un langage de plus bas niveau en se basant sur une technique classique de transformation de programmes fonctionnels largement utilisée[PJ86].

3.1 Situation dans le processus global de compilation

Comme le montre la figure 3.1, la phase de réduction peut être vue comme la première transformation effectuée sur le code source.

```
Erlang → UE → Scheme → C → Code natif
```

Figure 3.1: Phase de réduction

Cette première phase de la compilation simplifie grandement le langage. Nous décrivons d'abord UE, puis nous montrons comment les différentes formes syntaxiques de Erlang peuvent être compilées vers ce langage.

3.2 Données

L'ensemble des données de UE est exactement le même que celui d'Erlang. Aucune transformation n'est requise quant aux formes syntaxiques gérant la création des données, comme les constantes par exemple.

3.3 Formes syntaxiques

Le langage UE est composé d'un ensemble très restreint de formes syntaxiques. Souvent, quelques règles de réécriture suffisent pour convertir une forme syntaxique Erlang en forme syntaxique UE équivalente. Dans cette section, nous décrivons l'ensemble de ces formes. Nous joignons aussi la grammaire du langage UE à l'annexe 1. Nous pouvons y observer une grande réduction par rapport au langage Erlang, notamment sur le plan du nombre de formes syntaxiques.

3.3.1 Constantes et variables

Les formes syntaxiques décrivant les constantes et les variables sont les mêmes que celles de Erlang à l'exception qu'il est permis en UE de définir une variable débutant par le caractère `'%'`, ce qui permet à UE de définir de nouvelles variables sans se préoccuper des possibles collisions au niveau des noms avec les variables du programme Erlang source.

3.3.2 Patrons

Les formes syntaxiques décrivant les patrons restent aussi inchangées:

```
<patron> ::= <constante>                  |
             <variable>                   |
             <patron_universel>           |
             [<patron>|<patron>]          |
             {<patron>,...,<patron>}      |
             <patron> = <patron>
```

3.3.3 Filtrage de patrons: ecase

Nous présentons ici l'unique forme syntaxique traitant le filtrage de patrons dans le langage UE. UE ne contient aucune forme de traitement d'exception implicite. Le ecase est essentiellement le case de Erlang, à l'exception qu'il est considéré exhaustif, c'est-à-dire que le compilateur peut supposer que le sélecteur est filtré par au moins une des clauses. Nous dirons qu'une donnée est filtrée par le patron d'une clause si elle concorde avec ce dernier. La syntaxe du ecase est très semblable à celle du case de Erlang:

```
ecase <Expr> of
  <Patron> when <Gardes> -> <Expr> ;
  ...
  <Patron> when <Gardes> -> <Expr>
end
```

Grâce à cette forme syntaxique, nous pourrons transformer l'ensemble des formes syntaxiques de Erlang faisant appel au filtrage de patrons.

3.3.4 Application de fonction

Le langage UE ne possède qu'une forme syntaxique permettant d'appliquer une fonction. La syntaxe est la suivante:

```
<atome>(<Expr>,...,<Expr>)
```

3.3.5 Capture d'exception: ucatch

Erlang met deux formes syntaxiques à la disposition du programmeur pour la capture d'exception. Le langage UE possède une seule forme qui permet de généraliser ce mécanisme:

```
ucatch <Expr>
```

La forme ucatch fonctionne en capturant d'abord la continuation de l'expression et en la plaçant sur une pile. L'expression est ensuite évaluée, mais son résultat est placé dans le 2-tuple suivant:

```
{'NORMAL',<Resultat>}
```

Les fonctions prédéfinies exit et throw, qui génèrent les exceptions, appellent la continuation située sur le dessus de la pile avec leur argument aussi placé dans un 2-tuple ayant comme premier élément l'atome 'EXIT' ou 'THROW' respectivement. L'implantation des formes syntaxiques catch et try de Erlang est alors facile à réaliser comme nous le verrons dans la section 3.4.

3.3.6 Définition de fonctions anonymes: afun

La définition de fonctions anonymes a dû être modifiée afin de ne pas utiliser directement le filtrage de patrons. La syntaxe de définition de fonction anonyme est la suivante:

```
afun(<Arité>,<Id>) -> <Expr> end
```

Nous spécifions à l'aide d'un entier l'arité le la fonction, nous spécifions aussi, à l'aide d'un atome, un identificateur unique. L'expression représentant le corps de la fonction peut alors accéder aux argument via les variables %<*Id*>1,…,%<*Id*><*Arité*>. C'est parce que UE est un langage intermédiaire qui se veut minimal que nous avons décidé de ne conserver que l'arité de la fonction ainsi qu'un seul identificateur pour ses arguments plutôt que d'utiliser une syntaxe plus classique qui permettrait l'utilisation de noms de paramètres quelconques. Nous montrons maintenant un exemple qui permet de mieux voir le lien entre la définition de la fonction anonyme et ses arguments. Voici d'abord une définition de fonction anonyme en Erlang:

```
fun(1,X,a) -> X end
```

Voici maintenant la définition équivalente en UE:

```
afun(3,g) -> ecase {%g1,%g2,%g3} of
                 {1,X,a} -> X;
                 {_,_,_} -> exit(lambda_clause)
             end
end
```

En clair, la forme `afun(A,id) -> E` permet de dire au compilateur qu'il doit créer une fonction prenant `A` arguments qui, lors de l'appel, seront liés aux variables nommés `%id1,%id2,...,%idA` afin d'évaluer l'expression `E`.

3.3.7 Déclaration de fonctions nommées

La déclaration de fonction nommée n'est en fait qu'une liaison d'une fonction anonyme à un atome qui représente son nom, nous avons donc opté pour la syntaxe suivante:

```
<Atome> <- <Expr>
```

Il est à noter que cette forme n'est pas une expression et doit se trouver au niveau le plus haut d'imbrication dans un programme.

3.4 Compilation de Erlang vers UE

Nous avons développé une fonction de transformation ß, qui permet de passer d'une forme syntaxique Erlang vers une forme UE équivalente. La fonction de transformation ß prend donc en argument une forme syntaxique Erlang retourne une expression UE équivalente. L'implantation de cette fonction de transformation constitue donc le compilateur de Erlang vers UE. Nous illustrons la fonction de transformation par plusieurs règles de réécriture pour chacune des formes syntaxiques de Erlang.

Dans le compilateur Etos, ces règles sont appliquées au niveau des arbres de syntaxe. La forme textuelle de UE est utile à titre explicatif mais elle n'est pas utilisée sur le plan pratique par Etos. Cependant, le compilateur peut générer le code textuel pour fin de débuggage via la commande de compilation `-E`.

3.4.1 Les déclarations de fonctions nommées

La compilation des fonctions nommées se fait via une transformation vers une définition de fonction anonyme avec la forme `fun`. Il est alors important d'ajouter la clause d'exception dont la raison diffère de celle des fonctions anonymes. Nous utilisons ensuite l'opérateur de définition de fonctions nommées précédemment présenté.

```
ß(f(<P11>,…,<P1n>) when <G1> ->
    <B1>;
  …
  f(<Pm1>,…,<Pmn>) when <Gm> ->
    <Bm>)
```

≡

```
f <-
  ß(fun(<P11>,…,<P1n>) when <G1> ->
        <B1>;
    …
       (<Pm1>,…,<Pmn>) when <Gm> ->
        <Bm>;
        (_,…,_) when true ->
        exit(function_clause)
    end)
```

3.4.2 La forme case

La forme syntaxique `case` est évidemment transformée en forme `ecase`, en prenant soin d'expliciter la condition de lancement d'exception:

```
ß(case <Expr> of
      <P1> when <G1> -> <B1>;
      …
      <Pn> when <Gn> -> <Bn>
  end)
```

≡

```
ecase ß(<Expr>) of
  <P1> when <G1> -> ß(begin <B1> end)
  …
  <Pn> when <Gn> -> ß(begin <Bn> end)
  _ when true -> exit(case_clause)
end
```

Notons aussi que les corps sont encapsulés dans une forme `begin` avant d'être transformés. Ceci est requis car contrairement au `case` d'Erlang, le `ecase` sélectionne une expression à évaluer et non un corps. La spécification explicite d'un corps ne fait pas partie du langage UE.

3.4.3 L'opérateur de match

L'opérateur de match effectuant un filtrage de patron, nous utilisons donc notre forme `ecase` pour en faire la transformation. L'opérateur de match levant implicitement une exception si le patron n'est pas filtré, nous devons expliciter ce cas à l'aide d'une clause supplémentaire. À première vue, la transformation suivante est excellente:

```
ß(<P> = <E>)
```

≡

```
ecase ß(<E>) of
  <P> when true -> ß(<P>);
  _ -> exit(badmatch)
end
```

En effet, cette transformation préserve la sémantique de la forme syntaxique, mais le problème ici, c'est la duplication du code du patron. En effet, un patron peut aisément être transformé en expression, mais il peut alors se traduire en construction de tuples ou de paires, ce qui génère du code inutilement puisque le résultat a déjà été calculé lors de l'évaluation de <*E*>. Nous pouvons corriger la transformation en utilisant le patron de match pour placer le résultat dans une variable temporaire:

```
                          ecase ß(<E>) of
ß(<P> = <E>)      ≡         %v = <P> when true -> %v;
                            _ -> exit(badmatch)
                          end
```

Nous obtenons donc ici un code UE à la fois efficace et tout à fait équivalent au code Erlang. Bien entendu, Etos possède d'autres mécanismes d'optimisation, telle la propagation de copie, qui permettront d'éliminer la liaison de variable si ce n'est pas requis.

3.4.4 La forme `begin`

La forme syntaxique de séquencement `begin` peut être compilée récursivement à l'aide du `ecase` puisque ce dernier contient une séquence d'évaluation implicite. En effet, le sélecteur est toujours évalué avant les expressions des clauses. La transformation de cette forme peut donc se faire selon la règle récursive suivante:

```
ß(begin <E> end)                  ≡  ß(<E>)

                                     ecase ß(<E1>) of
                                       _ when true ->
ß(begin <E1>,<E2>,…,<En> end)     ≡      ß(begin <E2>,…,<En> end);
                                     end
```

3.4.5 La forme `if`

La forme syntaxique `if` est un enrobage syntaxique qui permet de faire un filtrage basé exclusivement sur des gardes. Contrairement aux autres formes de filtrage, elle ne permet pas d'effectuer de liaisons puisqu'elle ne permet pas l'utilisation de patrons. La transformation est donc très directe, il suffit de réécrire la forme et d'expliciter la condition de lancement d'exception:

```
ß(if                                 ecase 0 of
    <G1> -> <B1>;                      _ when <G1> -> ß(begin <B1> end)
    ...                         ≡      ...
    <Gn> -> <Bn>                       _ when <Gn> -> ß(begin <Bn> end)
  end)                                 _ when true -> exit(if_clause)
                                     end
```

Il s'agit donc de faire une sélection sur une expression bidon (nous utilisons ici la constante 0) afin d'évaluer les expressions de garde et faire le choix d'une clause basé sur l'évaluation de ceux-ci.

3.4.6 La forme cond

Nous compilons la forme conditionnelle cond de façon récursive. En effet, il s'agit de faire une sélection booléenne sur chacune des expressions à vérifier jusqu'à ce qu'il y en ait une qui évalue à true:

```
                            ecase ß(<E>) of
ß(cond                        true when true -> ß(begin <B> end);
   <E> -> <B>         ≡       false when true -> exit(cond_clause);
  end)                        _ -> exit(badbool)
                            end

                            ecase ß(<E1>) of
                              true when true -> ß(begin <B1> end);
ß(cond                        false when true -> ß(cond
    <E1> -> <B1>;                                   <E2> -> begin <B2> end;
    <E2> -> <B2>;     ≡                             ...
    ...                                             <En> -> begin <Bn> end
    <En> -> <Bn>                                  end);
  end)                        _ when true -> exit(badbool)
                            end
```

3.4.7 La forme some_true

La forme some_true peut aisément être transformée en forme cond de la façon décrite ci-dessous:

```
                                        ß(cond
                                            <E1> -> true;
                                            ...
ß(some_true <E1>,...,<En> end)    ≡         <En> -> true;
                                            true -> false
                                          end)
```

En effet, dès qu'une expression évalue à `true`, le résultat des deux constructions est `true` et si aucune expression s'évalue à `true`, le résultat est `false` dans les deux cas. Nous avons donc bien l'équivalence entre les deux constructions.

3.4.8 La forme `all_true`

La forme `all_true` doit se faire récursivement à l'aide du `ecase`. Nous devons évaluer chacune des sous-expressions tant qu'elles s'évaluent à `true`. Voici la transformation proposée:

```
                                ecase ß(<E>) of
                                  true when true -> true;
ß(all_true <E> end)     ≡         false when true -> false;
                                  _ when true -> exit(badbool)
                                end

ß(all_true
     <E1>,                      ecase ß(<E1>) of
     <E2>,                        true when true -> ß(all_true <E2>,…,<En> end);
     …                  ≡         false -> false;
     <En>                         _ -> exit(badbool)
  end)                          end
```

3.4.9 La forme `catch`

Nous compilons la forme `catch` de Erlang en utilisant la forme `ucatch` de UE. Il s'agit simplement de vérifier l'évaluation de l'expression sous `ucatch` et de retirer l'information appropriée du tuple retourné. La transformation qui vient d'abord à l'esprit va comme suit:

```
                           ecase ucatch ß(<E>) of
                             {'NORMAL',%v} when true -> %v;
ß(catch <E>)       ≡         {'THROW',%v} when true -> %v;
                             %v when true -> %v
                           end
```

Cette transformation est tout à fait valide, cependant il est possible de rendre le code généré beaucoup plus efficace puisque nous avons des informations quant à la structure du patron de la dernière clause. En effet, celui-ci devra nécessairement être un tuple avec `'EXIT'` comme premier élément. Le fait de spécifier cette information permettra au compilateur de n'effectuer aucun test sur la structure de 2-tuple présente alors dans tous les

patrons, puisque le `ecase` garantit qu'au moins une clause sera respectée. Nous procédons alors comme suit pour spécifier la structure de 2-tuple:

```
                          ecase ucatch ß(<E>) of
                            {'NORMAL',%v} when true -> %v;
ß(catch <E>)     ≡          {'THROW',%v} when true -> %v;
                            {'EXIT',_}=%v when true -> %v
                          end
```

Avec ce code, un seul test est requis si l'évaluation se termine normalement, et dans le cas où l'évaluation lance une exception, un seul test supplémentaire sera effectué pour déterminer laquelle des deux clauses restantes sélectionner.

3.4.10 La forme `try`

La compilation de la forme `try` ressemble beaucoup à celle du `catch`. En effet, il s'agit d'y ajouter les différentes clauses fournies dans la forme try, ainsi que de relancer les exceptions si elles ne sont pas capturées dans les clauses du `try`. Nous traitons aussi le cas du `try` sans clause. Notre transformation est donc la suivante:

```
                                     ß(try <E> catch
ß(try <E> end)                  ≡        %v -> %v
                                       end)

                                     ecase ucatch ß(<E>) of
                                       {'NORMAL',%v} when true -> %v;
ß(try <E> catch                        <P1> when <G1> -> ß(begin <B1> end);
    <P1> when <G1> -> <B1>;            ...
    ...                         ≡      <Pn> when <Gn> -> ß(begin <Bn> end);
    <Pn> when <Gn> -> <Bn>             {'EXIT',%v} when true -> exit(%v);
  end)                                 {'THROW',%v} when true -> throw(%v)
                                     end
```

3.4.11 La forme `receive`

La compilation de la forme `receive` est particulière puisqu'une boucle implicite doit être réalisée. En effet, afin de choisir le bon message dans la file, il faut parfois en filtrer plusieurs. Nous utilisons des fonctions de notre librairie d'exécution pour effectuer la boucle implicite:

```
ß(receive                                ecase receive_first(ß(<T>)) of
    <P1> when <G1> -> <B1>;                $timeout$ -> ß(begin <Bt> end)
    ...                                     <P1> when <G1> ->
    <Pn> when <Gn> -> <Bn>        ≡           ß(begin receive_accept(),<B1> end);
    after <T> -> <Bt>                       ...
  end)                                      <Pn> when <Gn> ->
                                              ß(begin receive_accept(),<Bn> end);
                                            _ when true -> receive_next()
                                          end
```

La fonction `receive_first` va d'abord capturer la continuation courante et la sauvegarder dans le descripteur du processus ainsi que la valeur du délai d'expiration. Ensuite, la file de message du processus est inspectée et si un message est en attente d'être lu, la continuation est appelée et le filtrage a lieu. La fonction `receive_next` inspecte à son tour la file de message et rappelle la continuation pour chacun des messages y étant présents. Si aucun message n'est présent, la continuation déjà capturée est utilisée pour effectuer un changement de contexte afin de laisser le contrôle aux autres processus. La fonction `receive_accept` s'occupe de retirer le message choisi de la file.

3.4.12 La forme `fun`

La compilation des fonctions anonymes se fait via la forme `afun` précédemment décrite. On s'occupe alors de vérifier l'arité de la fonction, ainsi que de définir un préfixe qui servira pour nommer les arguments. Le choix du préfixe doit être fait de façon à ce qu'aucune autre fonction imbriquée ne provoque de collision de nom. Ceci peut être réalisé avec une fonction comme `gensym` de Scheme qui permet de générer un symbole toujours différent au fil des appels. Nous devons aussi expliciter la condition d'exception, lorsqu'aucune clause de la fonction n'est sélectionnée.

```
                                            afun(n,g) ->
ß(fun                                         ecase {%g1,...,%gn} of
    (<P11>,...,<P1n>) when <G1> ->              {<P11>,...,<P1n>} when <G1> ->
      <B1>;                                       ß(<B1>);
    ...                               ≡         ...
    (<Pm1>,...,<Pmn>) when <Gm> ->              {<Pm1>,...,<Pmn>} when <Gm> ->
      <Bm>                                        ß(<Bm>);
  end)                                          {_,...,_} when true ->
                                                  exit(lambda_clause)
                                              end
```

3.4.13 Les opérateurs autres que l'opérateur de match

Les expressions Erlang formées à partir d'opérateurs, à l'exception de l'opérateur de match, sont transformées sous forme fonctionnelle. Il s'agit d'une transformation qui permet une grande simplification de la grammaire. En effet, l'utilisation d'opérateur n'est en fait qu'un enrobage syntaxique destiné à faciliter la compréhension du programme par l'humain. Le langage UE n'étant pas conçu pour être lu par le programmeur, nous pouvons alors nous permettre d'alléger la complexité de la grammaire en réécrivant chaque expression infixe par son équivalent préfixe, c'est-à-dire l'application d'une fonction. Cette étape est d'autant plus importante que la prochaine étape de compilation portera le code vers Scheme, qui est un langage utilisant exclusivement la notation préfixe.

Notons aussi que les formes syntaxiques permettant la construction de tuples, de paires ou de listes sont aussi transformées vers des applications de fonctions. Si tous les arguments sont constants et ne peuvent donner lieu à une exception, le compilateur transformera l'application en constante.

3.4.14 Les patrons

Aucune transformation n'est effectuée au niveau des patrons Erlang. Puisque les patrons ne dénotent pas une expression à évaluer mais contiennent plutôt de l'information au sujet de la structure d'un filtre, il devient alors inutile de transformer un patron de tuple ou de paire par une application de fonction.

3.4.15 Les gardes

Les règles de transformation des gardes sont les mêmes que celles des expressions Erlang normales. Il est à noter que les gardes ne sont pas séquencés, un peu comme dans le cas de la forme `begin`, comme le font certaines implantations du langage. Les gardes sont conservés sous forme de liste car c'est la prochaine étape de compilation qui décidera de l'ordre d'évaluation afin de produire un code plus performant.

3.5 Propriétés du langage UE

Le langage UE possède plusieurs propriétés intéressantes. Nous énumérons ici les propriétés qui le démarquent du langage Erlang. L'ensemble de ces propriétés en font un langage intéressant non pas seulement en tant que langage intermédiaire, mais aussi en tant que langage à part entière. Le langage UE pourrait être utilisé pour implanter des parties de code très efficaces, comme des librairies. Nous verrons entre autres comment il est possible de profiter du fait que UE ne soit pas sécuritaire, d'où son nom "Unsafe Erlang".

3.5.1 Centralisation du filtrage de patrons

Dans la section précédente, nous avons vu que toutes les formes syntaxiques de Erlang qui utilisent le filtrage de patrons peuvent être transformées en formes UE équivalentes. Il s'agit de la principale propriété qui est à l'origine de la conception du langage UE. En effet, cette centralisation du filtrage permet de traiter en un seul cas l'ensemble des formes de filtrage, ce qui réduit considérablement le travail de la prochaine étape de compilation qui passe du langage UE au langage Scheme.

L'utilisation de la forme `ecase` de UE a permis d'effectuer cette centralisation de façon simple. Il est à noter que cette centralisation aurait pu être effectuée avec la forme `case`, la nature non sécure du `ecase` n'étant pas nécessaire pour ce faire. La motivation derrière l'utilisation d'une forme non sécuritaire sera expliquée plus loin.

3.5.2 Centralisation des liaisons de variables

Un effet intéressant de la centralisation du filtrage est la centralisation des liaisons de variables. En effet, toute liaison de variable dans Erlang passe par le filtrage. Par exemple, pour lier une valeur V à une variable X, on tente de filtrer avec le patron X une valeur U qui peut être égale ou contenir, dans une construction telle un tuple, la valeur V. La forme la plus simple de liaison peut être réalisée avec l'opérateur de match comme dans l'exemple suivant:

```
{_,X}=U
```

Ici, nous lions à X la valeur du deuxième élément du tuple U. Comme nous avons vu dans la section précédente, cette instruction Erlang sera transformé vers une instruction ecase du langage UE, qui explicitera le traitement d'erreur qui dans ce cas est le lancement de l'exception badmatch si U n'est pas un 2-tuple.

La centralisation des liaisons de variables n'est toutefois pas complète dans le langage UE. En effet, la définition de fonction anonyme avec la forme afun implique la liaisons des arguments actuels aux arguments formels lors de l'appel de la fonction ainsi définie. Nous verrons que ce cas spécial ne pose aucun problème lors de la compilation vers Scheme puisque chaque définition de fonction dans Scheme implique la même liaison d'arguments.

3.5.3 Brèche de sécurité

Comme son nom l'indique, "Unsafe Erlang" n'est pas un langage aussi sécuritaire qu'Erlang. En effet, il existe une brèche volontaire dans la sécurité de UE qui est introduite avec la forme syntaxique ecase. Le fait de supposer qu'au moins une clause filtre l'expression de sélection permet de construire une expression UE dont l'évaluation reste indéterminée. Par exemple:

```
ecase 1 of
  2 when true -> a;
  3 when true -> b
end
```

La valeur de cette expression ne peut être déterminée puisqu'aucune clause ne filtre l'expression de sélection. Le compilateur suppose donc une contradiction.

Cependant, il est à noter qu'aucune transformation vue dans la section précédente ne résulte en expressions UE dont l'évaluation ne peut être déterminée. Ceci est dû au fait que le traitement d'erreur qui est implicite dans Erlang doit être explicité dans UE. Donc, une expression Erlang telle que la suivante:

```
case 1 of
  2 when true -> a;
  3 when true -> b
end
```

sera augmentée d'une clause de traitement d'exception au sein de sa forme équivalente UE:

```
ecase 1 of
  2 when true -> a;
  3 when true -> b;
  _ when true -> exit(case_clause)
end
```

Nous permettons donc à UE de supposer qu'au moins une clause filtre l'expression de sélection et l'évaluation de l'expression `ecase` est alors déterminée et soulèvera l'exception `case_clause`.

Il est à noter que nous n'avons pas encore justifié la raison de l'existence de la brèche de sécurité. En effet, toutes les transformations peuvent se faire vers la forme `case` et non la forme `ecase`. La brèche de sécurité a plutôt été introduite afin d'augmenter l'efficacité du code généré comme nous en discutons dans la prochaine section.

3.5.4 Aucun traitement d'exception implicite

Nous montrons ici comment la brèche de sécurité introduite dans la forme syntaxique `ecase` permet des constructions beaucoup plus efficaces que celles de Erlang. Essentiellement, le fait d'expliciter le traitement d'erreur permet au compilateur de l'éliminer dans les cas où il peut s'assurer, grâce à des informations de haut niveau difficilement détectables de façon automatisée, qu'au moins une clause filtre l'expression de sélection.

Il y a un cas dans les transformations précédemment décrites qui utilise cette possibilité d'optimisation, il s'agit du traitement d'exception des formes `try` et `catch`. Rappelons-nous la transformation du `catch` par exemple:

```
                          ecase ucatch ß(<E>) of
                            {'NORMAL',%v} when true -> %v;
ß(catch <E>)      ≡         {'THROW',%v} when true -> %v;
                            {'EXIT',_}=%v when true -> %v
                          end
```

Nous savons que `ucatch` *<expr>* retourne un 2-tuple permettant de déterminer si l'évaluation de l'expression a été terminée normalement, par un lancement d'exception avec `throw` ou avec `exit`. Cette information structurelle à propos de la valeur de l'expression de sélection nous permet de ne générer aucun cas d'exception. De plus, comme les patrons de toutes les clauses sont des 2-tuples, il devient inutile de générer un test pour s'en assurer. En effet, le cas où ce test serait négatif est en contradiction avec la supposition du compilateur à l'effet qu'au moins une clause filtre l'expression de sélection.

Dans l'exemple présenté, l'optimisation va encore plus loin. En effet, non seulement nous devons considérer la valeur de l'expression de sélection comme étant un 2-tuple, nous devons aussi nous assurer que son premier élément est `'NORMAL'`, `'THROW'` ou `'EXIT'`. Il devient alors intéressant de voir qu'une fois que le compilateur aura déterminé par deux tests que le premier élément n'est pas `'NORMAL'` ni `'THROW'`, il pourra déduire automatiquement que c'est `'EXIT'` et ce sans générer de test. Nous verrons plus en détail dans le quatrième chapitre comment cette optimisation peut générer un code plus efficace.

3.6 Mode de compilation non sécuritaire

Le gain d'efficacité obtenu par l'explicitation des conditions d'erreur peut être utilisé pour implanter des parties de code Erlang très critiques au niveau du temps d'exécution. En effet, une fois que la validité d'un module a été établie, il pourrait devenir intéressant de retirer les mécanismes de traitement d'exception implicites et profiter par le fait même d'optimisations très intéressantes.

La nature exhaustive de la forme `ecase` permet par exemple de supposer que les arguments des fonctions sont toujours valides. Par exemple, voici une implantation de la BIF (*Built-In Function*) `length/1` en Erlang:

```
length([_|T]) -> 1+length(T);
length([]) -> 0.
```

Le code compilé de cette fonction doit d'abord tester si son argument est une paire. Si ce n'est pas le cas, un deuxième test doit être effectué pour déterminer si l'argument est la liste vide ou si une exception doit être lancée. La suppression du traitement d'erreur implicite pourrait donc éliminer ce dernier test, rendant du même coup la fonction plus efficace.

Notre implantation définit donc l'option de compilation `unsafe` qui permet effectivement d'enlever les mécanismes de traitement d'erreur implicites. L'utilisation de l'option se fait avec la syntaxe suivante:

```
-compile(unsafe).
```

La fonction `length/1` est alors traduite en UE par:

```
length/1 <-
  afun(1,a) ->
    ecase {%a1} of
      {[_|T]} when true -> 'erl-+'(1,'length/1'(T));
      {[]} when true -> 0;
    end
  end.
```

Aucun test n'est alors requis pour que la deuxième clause soit choisie après l'échec de la première puisque c'est la dernière d'une forme `ecase`. Les gains peuvent être encore plus grands quand le filtrage n'est utilisé que pour l'extraction de donnée à l'intérieur de données structurées. Il doit être clair cependant que ces gains de performance se font au détriment de la sécurité. En effet, un module compilé avec l'option `unsafe` peut faire planter le système lors de l'exécution s'il n'est pas utilisé comme il se doit ou s'il contient des erreurs. Cette option permet par contre d'écrire des programmes performants en Erlang qui ne reposent pas sur la sécurité offerte par le langage.

3.7 UE comme langage de transition de Erlang vers Scheme

Nous avons donc vu dans ce chapitre comment Erlang peut être grandement simplifié par le retrait des enrobages syntaxiques en faisant partie, ainsi qu'en réduisant son niveau d'abstraction par l'explicitation du traitement d'erreur géré par les exceptions. Avec le langage UE, nous croyons avoir réduit Erlang vers une de ses plus simples expressions.

Comme déjà spécifié, UE est un langage de transition de Erlang vers Scheme. En effet, UE isole une des principales difficultés de compilation vers Scheme: le filtrage de patrons. Une implantation efficace de Erlang doit donc passer par une implantation efficace de système de filtrage de patrons, c'est-à-dire la compilation de la forme syntaxique `ecase`.

Nous consacrons donc le prochain chapitre à la compilation du langage UE vers Scheme. Nous y verrons que l'ensemble des formes syntaxiques de UE, à l'exception du `ecase`, ont un équivalent très proche en Scheme. L'essentiel du travail sera donc la compilation du filtrage de patrons en Scheme.

Chapitre IV

UE vers Scheme: implantation efficace du filtrage

La compilation des patrons de filtrage est effectuée lors de la compilation du langage UE vers Scheme. En effet, Scheme ne possède pas de forme syntaxique de base permettant le filtrage. Il s'agit en fait de la principale difficulté à surmonter lors de cette phase. Nous montrerons donc dans ce chapitre comment sont compilées les formes syntaxiques du langage UE. Ensuite, nous présentons une analyse du problème du filtrage qui est inhérent à l'implantation de la forme `ecase`.

4.1 Situation dans le processus global de compilation

Comme le montre la figure 4.1, la compilation du langage UE vers Scheme est la deuxième phase de compilation. Il s'agit aussi de la dernière phase de compilation de Etos, puisque les autres phases sont prises en charge par le compilateur Gambit-C.

```
Erlang → UE → Scheme → C → Code natif
```

Figure 4.1: Compilation vers Scheme

4.2 Passage de UE à Scheme : la transformation μ

Afin de compiler efficacement les liaisons de variables vers Scheme, nous avons procédé à une compilation par conversion partielle en style CPS (*Continuation Passing Style*). La fonction de compilation, que nous avons nommée μ, prend donc comme arguments une expression UE ainsi qu'une continuation (une expression Scheme dénotant une fonction). L'ensemble des déclarations de fonctions nommées qui constituent un programme UE est alors compilé avec la fonction d'identité comme continuation, ce qui permet d'éviter de passer explicitement une continuation à chaque appel de fonction. C'est pour cette raison que nous parlons d'une conversion partielle.

4.2.1 Système de gestion des exceptions

Lors de l'exécution, notre système fournit un service de gestion des exceptions au sein de la librairie de Etos. Ce système peut facilement être implanté à l'aide des continuations et de la fonction `call/cc`. L'idée est de conserver une pile de continuation qui sera alimentée de la continuation courante dès qu'un calcul est mis sous la surveillance

d'une expression `ucatch`. Les fonctions suivantes constituent l'interface du système de gestion des exceptions:

```
(erl-push-abrupt! k) ; empile une continuation
(erl-pop-abrupt!)    ; dépile une continuation
(erl-abrupt-top)     ; la continuation sur la pile
```

La fonction `erl-abrupt-top` permettra aux fonctions `throw/1` et `exit/1` de lancer des exceptions en invoquant simplement la continuation au dessus de la pile. Au chargement du système, une fonction avertissant l'usager à la console est empilée et constitue le gestionnaire de traitement d'erreur par défaut disponible pour les applications.

4.3 Compilation des formes syntaxiques UE

Nous présentons ici les règles de compilation que nous avons élaborées afin de traduire le code UE vers un code Scheme équivalent. Comme dans le chapitre précédent, nous traiterons chaque forme séparément. Les détails de la forme `ecase` seront toutefois réservés à des sections suivantes, puisqu'ils constituent une grande part du travail présenté dans ce mémoire.

4.3.1 Constantes et variables

Les détails de la conversion des constantes ont été présentés dans les deux premiers chapitres d'introduction au langage Erlang. Puisque les données de Erlang et de UE sont identiques, il serait inutile de les décrire à nouveau. Pour les variables, seuls les noms doivent être changés si l'implantation de Scheme utilisée n'est pas sensible à la casse, comme expliqué au deuxième chapitre.

4.3.2 Application de fonction

UE ne permet l'application directe que des fonctions nommées. La compilation d'une application UE vers une application équivalente Scheme n'est donc qu'une simple conversion de continuation:

```
                              µ(<E1>,(lambda ($A1)
                                       µ(…
µ(f(<E1>,…,<En>),k)    ≡                 µ(<En>,(lambda ($An)
                                                  (k (f $A1 … $An))))…)))
```

Afin de garantir l'intégrité du système, chaque appel de fonction non locale au module implique la définition d'une fonction de traitement d'erreur du même nom, permettant de soulever une exception. Ainsi, un appel à une fonction d'un module non chargé ne provoquera pas une erreur Scheme mais bien une erreur Erlang. Par exemple, si un module `A` contient un appel à la fonction externe `B:f/0`, une fonction ayant pour nom `B:f/0` sera définie dans le module A pour éviter que l'appel ne fasse planter le système Scheme dans le cas où le module A ne serait pas chargé.

4.3.3 La forme `ucatch`

La compilation vers Scheme de la forme `ucatch` est réalisée à l'aide de la fonction `call/cc`:

```
                                  (let ((rname
                                         (call/cc (lambda (abrupt)
                                                    (begin
                                                      (erl-push-abrupt! abrupt)
                                                      μ(<E>,
μ(ucatch <E>,k)  ≡                                      (lambda (e)
                                                          (erl-tuple '^n^o^r^m^a^l
                                                                      e)))))))
                                    (erl-pop-abrupt!)
                                    (k rname))
```

Il s'agit donc d'abord de capturer la continuation courante et de l'empiler à l'aide de la fonction `erl-push-abrupt!` du système de gestion des exceptions. On évalue ensuite l'expression <*E*> avec une continuation qui retourne un 2-tuple composé de l'atome `'NORMAL'` et du résultat de l'évaluation. Ceci est requis pour permettre de distinguer les erreurs des données normales. Le résultat de l'appel à `call/cc`, qui peut alors provenir de l'évaluation complète ou terminée de façon abrupte de <*E*>, est alors mis dans une variable temporaire. Ensuite, la continuation est dépilée à l'aide de la fonction `erl-pop-abrupt!`. Finalement, la continuation de départ `k` est appelée avec le résultat du `call/cc` précédemment conservé.

4.3.4 La forme `afun`

La compilation des définitions de fonctions anonymes passe par la fonction `erl-function` de la librairie d'exécution qui permet de construire une fonction Erlang à l'aide d'une fermeture Scheme et de son arité:

```
                                      (k (erl-function
                                           <n>
µ(afun(<n>,<Id>) -> <E>,k)  ≡              (lambda
                                             (%<id>1 … %<id><n>)
                                             µ(<E>,(lambda (x) x)))))
```

La fonction `erl-function` est donc appelée avec l'arité de la fonction anonyme. La fermeture est créée avec la forme spéciale `lambda` qui définit les noms des arguments en fonction de l'identificateur unique et dont le corps est le résultat de la compilation de l'expression <*E*> avec comme continuation la fonction identité `(lambda (x) x)`.

4.3.5 Déclaration de fonctions nommées

La compilation de fonctions nommées se fait à l'aide de la fonction `erl-function-lambda` de la librairie d'exécution qui permet d'accéder à la fermeture Scheme contenue dans une fonction Erlang. Nous définissons la compilation d'une fonction nommée comme une fonction þ retournant une liste composée du nom local de la fonction nommée ainsi que sa fermeture associée. Donc, la compilation d'une fonction nommée s'illustre comme suit:

```
                                           (<Nom>/<n>
þ(<Nom> <- afun(<n>,<Id>) -> <E>)  ≡        (µ(afun(<n>,<Id>) -> <E>),
                                             (lambda (x) x)))
```

Puisqu'un module UE est une suite de déclarations de fonctions nommées, un module UE compilé est représenté par une liste associative composée des noms locaux et des fermetures qui leurs sont associés. Il reste bien peu à faire pour avoir un module Scheme utilisable. En fait, cette liste constitue le premier argument qui sera passé à la forme Scheme `letrec`. Pour compiler le module `M`, le corps du `letrec` ne consiste qu'en l'exportation des fonctions appropriées, ce qui peut être réalisé grâce à la transformation `T` suivante:

```
                                         (letrec ((a/1 (lambda (%a1) …))
                                                  (b/2 (lambda (%b1 %b2) …))
T((a/1 (lambda (%a1) …))                          …)
  (b/2 (lambda (%b1 %b2) …))  ≡           (set! M:a/1 a/1)
  …))                                     (set! M:b/2 b/2)
                                          …)
```

Il devient alors possible de créer des modules Scheme définissant l'ensemble des fonctions du module Erlang et permettant à certaines fonctions exportées d'être accessibles à partir des autres modules. Dans notre implantation, l'ensemble des fonctions exportées est plutôt inclus dans un descripteur de module (un tuple) qui permet de vérifier, lors de l'exécution, quelles fonctions sont définies.

4.3.6 La forme ecase

Jusqu'à présent, il a été démontré que l'ensemble du langage UE pouvait se traduire vers le langage Scheme de façon assez directe. Cependant, puisque Scheme ne possède aucun mécanisme approprié de filtrage de patrons, la compilation de la forme syntaxique `ecase` exige plus de travail. Nous consacrerons donc le reste de ce travail à l'élaboration de solutions pour produire du code Scheme efficace à partir des expressions `ecase` du langage UE. Pour le moment contentons-nous de définir une fonction de transformation `S` englobant plus précisément le problème du filtrage:

```
μ(ecase <E> of                     μ(<E>,
    <P1> when <G1> -> <B1>;          (lambda (sel)
    …                          ≡       S(<P1> when <G1> -> μ(<B1>,k),
    <Pn> when <Gn> -> <Bn>               …,
  end,k)                                 <Pn> when <Gn> -> μ(<Bn>,k))))
```

La fonction de transformation `S` travaille donc exclusivement sur la sélection des clauses en faisant abstraction de l'évaluation du sélecteur. Chaque clause fournie à `S` est formée d'un patron augmenté d'un garde, puis d'un corps compilé en Scheme. Notons que la duplication de code présente dans la forme actuelle de la transformation sera traitée ultérieurement. Le problème du filtrage de patrons se traduit donc maintenant par l'implantation de la fonction `S`.

4.4 Traitement du filtrage

Sous les langages Erlang et UE, le filtrage de patrons englobe deux mécanismes permettant d'en faire des langages utiles: la liaison des données aux variables ainsi que la gestion du flot de contrôle. Voici un exemple d'expression `ecase` qui démontre ces deux aspects:

```
ecase L of
  [_|T] -> 1 + f(T) ;
  [] -> 0 ;
end
```

Cette expression permet de diriger le flot de contrôle afin d'évaluer l'expression `1 + f(t)` si `L` est une paire et l'expression `0` si `L` est la liste vide. D'autre part, si `L` est une paire, la variable `T` sera liée à la queue de `L`, en supposant que `T` soit libre lors du choix de la clause. Les deux prochaines sections s'attaquent donc respectivement à la liaison des variables ainsi qu'à la gestion du flot de contrôle au sein du mécanisme de filtrage de patrons de UE.

4.4.1 Le filtrage de patrons Erlang versus l'unification de Prolog

Bien que la syntaxe de Erlang soit grandement inspirée de celle de Prolog, il existe des différences entre le filtrage de patrons de Erlang et le processus d'unification de Prolog.

Erlang permet de contraindre la sélection de clauses avec les gardes. À ce chapitre, quelques variantes de Prolog dont Concurrent Prolog [SHA88] permettent aussi de le faire. La différence majeure tient au fait que Erlang n'impose pas le *backtracking*. En effet, puisque les variables ne peuvent être contenues dans les données Erlang, aucun mécanisme de *backtracking* n'est nécessaire dans une implantation de Erlang puisque le processus de filtrage revient à une unification dans un seul sens. Le processus de filtrage ne simule donc pas le non déterminisme comme Prolog et ceci implique que l'implantation du filtrage sous Erlang est un problème différent de l'implantation de l'unification sous Prolog. Le filtrage de Erlang est destiné à gérer le contrôle de flot et à extraire les données des structures, tandis que l'unification de Prolog est surtout utile à la recherche de solutions. Des travaux

effectués sur Prolog [DW86] tentent d'ailleurs de minimiser l'impact des mécanismes de *backtracking* lorsque les prédicats s'apparentent à des fonctions.

À notre avis, il devient alors moins intéressant de baser nos travaux sur ceux effectués dans le cadre d'une implantation Prolog [BOI88]. Le problème du filtrage sous Erlang revient essentiellement au choix des tests à effectuer afin de décider quelle clause choisir et en ce sens, les travaux de Baudinet et MacQueen [BM85] effectués dans un contexte de compilation du langage ML nous ont semblé plus utiles.

4.5 Liaison des variables

Nous présentons ici deux exemples d'expressions `ecase` afin de définir plus clairement le problème des liaisons de variables.

4.5.1 Le statut des variables

Le premier exemple illustre l'importance du statut des variables dans l'interprétation de la sémantique des expressions `ecase`:

```
ecase <E> of
  X -> <B1> ;
  _ -> <B2>
end
```

L'évaluation de cette expression implique d'abord l'évaluation du sélecteur, l'expression <*E*>. Une fois la valeur de cette expression déterminée, elle doit être filtrée par l'ensemble des clauses. Afin de déterminer comment évaluer le reste de l'expression, il est ici primordial de savoir si la variable `X` est liée ou libre (non liée), puisque la première clause est formée d'un patron contenant cette variable.

Si la variable `X` est liée après l'évaluation du sélecteur, le patron de la première clause accepte la valeur de ce dernier si et seulement si elle est égale à la valeur de `X`. Dans ce cas, le filtrage se traduit par un test d'équivalence, qui est implanté dans la librairie d'exécution par la fonction `erl-=:=`. Le patron de la deuxième clause étant le patron universel, cela revient à dire que si le test échoue, c'est la deuxième clause qui accepte le

sélecteur. Avec une continuation d'exécution k, un code Scheme équivalent à l'expression ecase pourrait ressembler à ceci:

```
µ(<E>,(lambda (sel)
        (if (erl-=:= sel ^x)
            µ(<B1>,k)
            µ(<B2>,k))))
```

Par contre, si la variable X n'est pas liée après l'évaluation du sélecteur, le patron de la première clause acceptera alors toute valeur en la liant à la variable X. Dans ce cas, la deuxième clause reste inaccessible et aucun test n'est à effectuer. Toujours avec une continuation d'exécution k, un code Scheme équivalent pourrait alors être:

```
µ(<E>,(lambda (sel)
        (let ((^x sel))
          µ(<B1>,k))))
```

Ou encore, une fois simplifié:

```
µ(<E>,(lambda (^x) µ(<B1>,k)))
```

Cet exemple démontre bien que la sémantique d'une expression ecase dépend du statut des variables contenues dans ses patrons, à savoir si elles sont liées ou libres. Cette information peut être obtenue par une analyse effectuée sur l'arbre de syntaxe abstrait de chacune des déclarations de fonctions. Chaque expression est alors scrutée et un registre des variables libres vient alors augmenter l'arbre de syntaxe. Bref, le problème des liaisons de variables au sein du mécanisme de filtrage de patrons implique d'abord une analyse du contexte dans lequel il apparaît.

4.5.2 Le moment de liaison et les patrons non linéaires

On dira qu'un patron est non linéaire lorsqu'il contient plus d'une référence à une même variable. Le deuxième exemple illustre pour sa part le problème du moment de liaison des variables et celui des patrons non linéaires.

```
ucase <E> of
   {X,<P1>} -> <B1> ;
   {<P2>,X} -> <B2> ;
   _ -> <B3>
end
```

Cet exemple illustre le fait que les liaisons effectuées lors du filtrage ne sont pas nécessairement permanentes. En effet, en supposant la variable X libre après l'évaluation du sélecteur et une fois qu'il a été confirmé que ce dernier est un 2-tuple, la liaison de X au premier élément de ce tuple devient caduque s'il s'avère impossible que le deuxième élément soit accepté par le patron <*P1*>. Dans ce cas, le patron de la deuxième clause pourrait plutôt lier X au deuxième élément du tuple.

Pourtant, la liaison de X au premier élément du tuple pourrait être requise si jamais le patron <*P1*> y faisait référence. Par exemple, si nous supposons <*P1*> étant la variable X, nous nous retrouvons avec l'expression suivante:

```
ucase <E> of
  {X,X} -> <B1> ;
  {<P2>,X} -> <B2> ;
  _ -> <B3>
end
```

Ici, le patron de la première clause est non linéaire puisqu'il contient plus d'une référence à la variable X. Dans cette nouvelle expression, il semble important de lier la variable X au premier élément du tuple afin que le test sur le deuxième élément puisse être fait. Cependant, si jamais ce test échouait, il faudrait alors délier la variable afin que la deuxième clause puisse être vérifiée.

Afin d'éviter ce problème, nous définissons le moment de liaison dynamique des variables lorsque le patron accepte le sélecteur. Durant le processus de filtrage, des environnements statiques différents seront conservés pour chacune des clauses. La liaison des variables s'effectuera donc une fois le patron accepté par la vérification des variables liées dans l'environnement statique de la clause choisie.

Ce mécanisme est rendu possible grâce à un *lambda-lifting* partiel des corps des clauses. Nous y vérifions l'ensemble des liaisons requises par l'analyse des variables libres. En effet, les variables requises sont constituées de la différence entre l'ensemble des variables libres juste après l'évaluation du sélecteur et l'ensemble des variables libres avant

l'évaluation du corps. La compilation de l'expression ecase générale ressemble donc maintenant à ceci:

```
µ(<E>,(let ((B1f (lambda (B1Vars…) µ(<B1>,k)))
            …
            (Bnf (lambda (BnVars…) µ(<Bn>,k))))
        (lambda (sel)
          S(<P1> when <G1> -> (B1f B1Vars…),
            …,
            <Pn> when <Gn> -> (Bnf BnVars…)))))
```

Cette transformation permet donc la liaison des variables requises par les corps. De plus, les liaisons introduites à des fins très locales peuvent être éliminées. Il n'est pas rare qu'un patron effectue une liaison qui n'est utile que pour des fins de filtrage. Par exemple, la fonction member/2, qui vérifie si un élément fait partie d'une liste pourrait être implantée comme suit:

```
member(X, [X|_]) -> true ;
member(X, [_|T]) -> member(X,T) ;
member(_, []) -> false.
```

Dans la première clause, la liaison de X au premier argument n'est utile que pour vérifier la tête de la liste formée par le deuxième élément. Cette liaison n'est donc pas utile pour l'évaluation du corps et il n'est donc pas nécessaire de la générer dynamiquement.

La dernière transformation possède aussi le problème de duplication de code engendré par la copie de la continuation k. Ce problème peut aussi être réglé par un mécanisme de *lambda-lifting* partiel. En effet, l'ensemble des variables requises par la continuation k peut aussi être calculé à l'aide de l'analyse des variables libres, il s'agit alors de la différence entre l'ensemble des variables libres après l'évaluation du sélecteur et l'ensemble des variables libres après l'expression ecase. La compilation de l'expression ecase générale devient donc:

```
μ(<E>,(let ((kf (lambda (r kVars…) (k r))))
       (let ((B1f (lambda (B1Vars…)
                    μ(<B1>,(lambda (r)
                             (kf r kVars…)))))
             …
             (Bnf (lambda (BnVars…)
                    μ(<Bn>,(lambda (r)
                             (kf r kVars…))))))
         (lambda (sel)
           S(<P1> when <G1> -> (B1f B1Vars…),
             …,
             <Pn> when <Gn> -> (Bnf BnVars…))))))
```

Le moment de liaison des variables pouvant être difficile à déterminer dans le cas des patrons non linéaires, nous procédons à l'explicitation des tests impliqués par les apparitions multiples de mêmes variables au sein d'un patron. Cette transformation est effectuée grâce aux expressions de garde qui permettent d'augmenter les patrons. Les occurrences multiples des variables sont donc remplacées par des tests d'équivalence dans les expressions de garde. Par exemple, la déclaration de la fonction `member/2` présentée plus haut est transformée par la suivante avant d'être traitée:

```
member(X, [Y|_]) when X =:= Y -> true ;
member(X, [_|T]) -> member(X,T) ;
member(_, []) -> false.
```

Cette transformation permet donc de s'assurer que le statut d'une variable présente dans un patron peut être déterminé avant même que la compilation du choix de la clause ne soit amorcée. Par ailleurs, cette transformation n'implique pas que les tests d'égalité soient effectués après les autres tests de conformité comme nous le verrons dans un prochain chapitre.

4.6 Gestion du flot de contrôle

Comme vu précédemment, la gestion du flot de contrôle s'effectue à l'aide de tests déterminés par la structure des patrons inclus dans les clauses des expressions `ecase`. Nous illustrerons ici quatre aspects importants qui entrent en jeu lors des prises de décisions entourant cette gestion, soient la duplication de code, la fusion des expressions communes, la prédiction des tests et enfin l'ordre d'évaluation de ces tests.

4.6.1 La duplication de code

L'exemple suivant tente de démontrer le problème de la duplication de code qui apparaît quand vient le moment du choix du corps à évaluer. Il s'agit d'une transformation `S` où l'on supposera les variables `X` et `Y` libres:

```
S([X] -> (B1f X),
  [] -> (B2f),
  Y -> (B3f Y))
```

Un code Scheme équivalent pourrait alors être:

```
(if (erl-cons? sel)
    (if (erl-nil? (erl-unsafe-tl/1 sel))
        (B1f (erl-unsafe-hd/1 sel))
        (if (erl-nil? sel)
            (B2f)
            (B3f sel)))
    (if (erl-nil? sel)
        (B2f)
        (B3f sel)))
```

Il est à noter ici que la fonction `erl-unsafe-tl/1` est une implantation de la BIF Erlang `tl/1` qui suppose que son argument est une paire. Cette fonction est équivalente à la fonction `cdr` de Scheme. En général, une fonction `erl-unsafe-X` est l'implantation de la BIF Erlang `X` dont on suppose que les arguments sont tels qu'il ne puisse y avoir de lancement d'exception. Dans le cas présent, nous évitons donc une duplication des tests puisque nous savons que le sélecteur est une paire.

Par ailleurs, cet exemple illustre bien le problème de la duplication de code. En effet, le code testant si le sélecteur est bien la liste vide est répété. Trois solutions permettent de régler ce problème de duplication. D'abord, si nous décidions de tester en premier si le sélecteur est bien la liste vide, nous aurions le code suivant, sans duplication de test:

```
(if (erl-nil? sel)
    (B2f)
    (if (erl-cons? sel)
        (if (erl-nil? (erl-unsafe-tl/1 sel))
            (B1f (erl-unsafe-hd/1 sel))
            (B3f sel))
        (B3f sel)))
```

Mais l'ordre des tests ne peut résoudre tous les cas. En effet, si la deuxième clause requérait plus d'un test différent de ceux de la première clause avant d'accepter le sélecteur, il deviendrait impossible de trouver un ordre dans lequel aucune duplication n'aurait lieu. Par exemple, si le patron de la deuxième clause est `[a|b]`, la duplication de code est inévitable. Dans ces cas, une autre solution est d'utiliser la création de fonctions sans arguments (*thunks*) qui permettent la réutilisation de code. Cette solution permet d'éviter la duplication de code dans notre exemple, même sans changer l'ordre des tests:

```
(let ((thunk (lambda ()
               (if (erl-nil? sel)
                   (B2f)
                   (B3f sel)))))
  (if (erl-cons? sel)
      (if (erl-nil? (erl-unsafe-tl/1 sel))
          (B1f (erl-unsafe-hd/1 sel))
          (thunk))
      (thunk)))
```

Une autre solution réside dans la prédiction des tests. Bien que la prédiction de tests permette de résoudre le problème en inférant que le test `(erl-nil? sel)` ne peut être vrai si le test `(erl-cons? sel)` l'est aussi, il demeure cependant qu'il existe des cas où la prédiction des tests ne permet pas de retirer totalement la duplication, notamment si le patron de la deuxième clause est `[a|b]`. Nous reparlerons dans un autre chapitre de la prédiction des tests et de l'approche que nous avons choisi pour éliminer la duplication de code.

4.6.2 La fusion des expressions communes

Voici un nouvel exemple de transformation `S` présentant la nécessité de la fusion des expressions communes afin d'obtenir une compilation efficace du filtrage:

```
S([$a|T] -> (B1f T),
  [$b|T] -> (B2f T),
  [$c|T] -> (B3f T),
  [_|_]  -> (B4f))
```

Ici, la discrimination des clauses s'effectue clairement sur le premier élément de la paire qu'est le sélecteur. Un code Scheme équivalent peut utiliser quelques tests en cascade afin de déterminer quelle clause doit être choisie:

```
(if (erl-chr= (erl-unsafe-hd/1 sel) #\a)
    (B1f (erl-unsafe-tl/1 sel))
    (if (erl-chr= (erl-unsafe-hd/1 sel) #\b)
        (B2f (erl-unsafe-tl/1 sel))
        (if (erl-chr= (erl-unsafe-hd/1 sel) #\c)
            (B3f (erl-unsafe-tl/1 sel))
            (B4f))))
```

Ce bout de code peut évidemment être optimisé en fusionnant les expressions communes. Dans le cas présent, les expressions `(erl-unsafe-hd/1 sel)` et `(erl-unsafe-tl/1 sel)` apparaissent trois fois chacune, il convient alors de ne les évaluer qu'une seule fois. Ceci peut être effectué par la conservation des résultats dans des variables temporaires comme dans la solution suivante:

```
(let ((exp1 (erl-unsafe-hd/1 sel))
      (exp2 (erl-unsafe-tl/1 sel)))
  (if (erl-chr= exp1 #\a)
      (B1f exp2)
      (if (erl-chr= exp1 #\b)
          (B2f exp2)
          (if (erl-chr= exp1 #\c)
              (B3f exp2)
              (B4f)))))
```

Nous traiterons des moyens utilisés afin de réaliser la détection des expressions communes dans le chapitre portant sur la prédiction des tests.

4.6.3 Évaluation efficace des conditions : la prédiction des tests

L'exemple suivant illustre le problème de la prédiction des tests. Soit la transformation `S` suivante vue précédemment:

```
S([X] -> (B1f X),
  [] -> (B2f),
  Y -> (B3f Y))
```

Le code Scheme naïf équivalent est:

```
(if (erl-cons? sel)
    (if (erl-nil? (erl-unsafe-tl/1 sel))
        (B1f (erl-unsafe-hd/1 sel))
        (if (erl-nil? sel)
            (B2f)
            (B3f sel)))
    (if (erl-nil? sel)
        (B2f)
        (B3f sel)))
```

Ce bout de code est inefficace puisqu'il contient un test inutile dont une des branches pointe vers du code mort. En effet, une fois confirmé que la variable `sel` est une paire avec le test `erl-cons?`, le test `erl-nil?` sur cette même valeur doit invariablement être faux. Le code pourrait donc être simplifié par le suivant:

```
(if (erl-cons? sel)
    (if (erl-nil? (erl-unsafe-tl/1 sel))
        (B1f (erl-unsafe-hd/1 sel))
        (B3f sel))
    (if (erl-nil? sel)
        (B2f)
        (B3f sel)))
```

Cet exemple démontre bien que la prédiction des tests peut améliorer les performances du code généré, tout en réduisant sa taille. La prédiction des tests n'est pas un problème trivial, il exige une connaissance approfondie de la structure des types de données ainsi que certains liens logiques unissant les valeurs de ces dernières. Pour cette raison, nous traiterons plus en détails de nos solutions dans un prochain chapitre.

4.6.4 L'ordre des tests

Bien qu'il faille effectuer plusieurs tests dans certains cas afin de vérifier si une valeur est conforme à un patron, l'ordre dans lequel procéder n'est toutefois pas défini. L'exemple suivant démontre l'impact que l'ordre des tests peut avoir sur l'efficacité d'un algorithme de filtrage:

```
S({a,b} -> (B1f),
  {_,c} -> (B2f))
```

Ici, nous pouvons décider d'effectuer en premier l'un des trois tests sur les éléments du tuple. Il est à noter ici qu'aucun test n'est requis pour déterminer si le sélecteur est bien un 2-tuple puisque la forme syntaxique `ecase` est exhaustive. Voici le code Scheme équivalent construit en effectuant d'abord un test déterminant si le premier élément du 2-tuple est l'atome `a`:

```
(if (erl-ato= (erl-unsafe-element/2 1 sel) 'a)
    (if (erl-ato= (erl-unsafe-element/2 2 sel) 'b)
        (B1f)
        (B2f))
    (B2f))
```

Voici maintenant un autre code Scheme équivalent, construit cette fois en effectuant d'abord un test déterminant si le deuxième élément du 2-tuple est l'atome `b`:

```
(if (erl-ato= (erl-unsafe-element 2 sel) 'b)
    (B1f)
    (B2f))
```

Les gains tant aux plans de l'espace que du temps d'exécution peuvent ici être facilement observés. En effet, le test sur le deuxième élément du tuple permet de choisir immédiatement quelle clause peut accepter le sélecteur. Le problème entourant le choix de l'ordre d'évaluation des tests est complexe et le résultat de nos recherches le concernant est présenté dans le prochain chapitre.

4.7 Vers des solutions

Nous avons donc vu dans ce chapitre que la compilation du langage UE vers le langage Scheme est très directe sauf en ce qui concerne le filtrage. Il a aussi été démontré que le problème du filtrage renvoyait aux problèmes de la liaison des variables et de la gestion du flot de contrôle.

Bien que la résolution du problème de la liaison des variables s'effectue aisément à la suite d'une analyse des variables libres, nous n'avons pas présenté de solution au

problème de la gestion du flot de contrôle. Les prochains chapitres présentent les solutions que nous avons élaborées pour le résoudre.

Chapitre V
La prédiction des tests

Ce chapitre traite du problème de la prédiction des tests qui est sous-jacent au problème du filtrage. Comme démontré précédemment, la prédiction des tests permet l'élimination du code mort et des tests inutiles. Un algorithme de filtrage ayant la faculté de prédire l'issue de certains tests devrait donc être plus performant tant aux plans du temps d'exécution que de l'espace mémoire consommé.

Nous établirons d'abord le cadre dans lequel sera implanté et utilisé la prédiction des tests, soit par la compilation de la transformation `S` vers une structure d'arbre de décisions après la transformation des clauses vers une forme plus adéquate: les expressions de test. Nous verrons par la suite que les problèmes de duplication de code ainsi que de fusion des expressions communes seront plus aisément résolus avec la structure d'arbre. Ensuite, il sera démontré comment l'analyse de la structure des types de données ainsi que des relations logiques concernant leurs valeurs permettent, avec la retenue des informations provenant de tests antérieurs, l'inférence de la valeur de certains tests.

5.1 Les expressions de test

Les tests à effectuer lors du filtrage sont déterminés par les patrons augmentés contenus dans les clauses. La syntaxe actuelle de ces patrons n'est pas très commode puisqu'elle n'explicite pas les tests Scheme correspondants et n'est pas très concise. Par exemple, les paires de patrons augmentés suivantes ont des représentations syntaxiques différentes mais sont tout à fait équivalentes:

```
[_|_]   ≡   X when is_cons(X)
1       ≡   X when X=:=1
[_]     ≡   X when is_list(X),length(X)=:=1
```

La nouvelle syntaxe ici proposée explicitera les tests Scheme et les liaisons de variables. Cette syntaxe est aussi augmentée de formes spéciales permettant la prédiction de certains tests. Nous présenterons d'abord un diagramme montrant l'éventail des tests disponibles sur les différents types de donnée Scheme utilisés. Nous parlerons plus spécifiquement de la représentation du compilateur Gambit, qui a été utilisé dans notre implantation de Etos.

5.1.1 Tests sur les types de données Scheme de Gambit

L'ensemble des données Erlang est représenté en Scheme à l'aide des entiers, des réels, des caractères, des symboles, des paires, des vecteurs et la liste vide. Sous Gambit, les données sont représentées en mémoire à l'aide de références de 32 ou 64 bits dont les deux derniers bits permettent de discriminer entre des classes de types. Ces quatre grandes classes incluent les entiers à taille fixe (les fixnums), les quantités immédiates (liste vide, caractères), les paires et les données sous-typées (nombres à virgule flottante, grands entiers, symboles et vecteurs). Ces classes forment une hiérarchie de tests qui est illustrée à la figure 5.1.

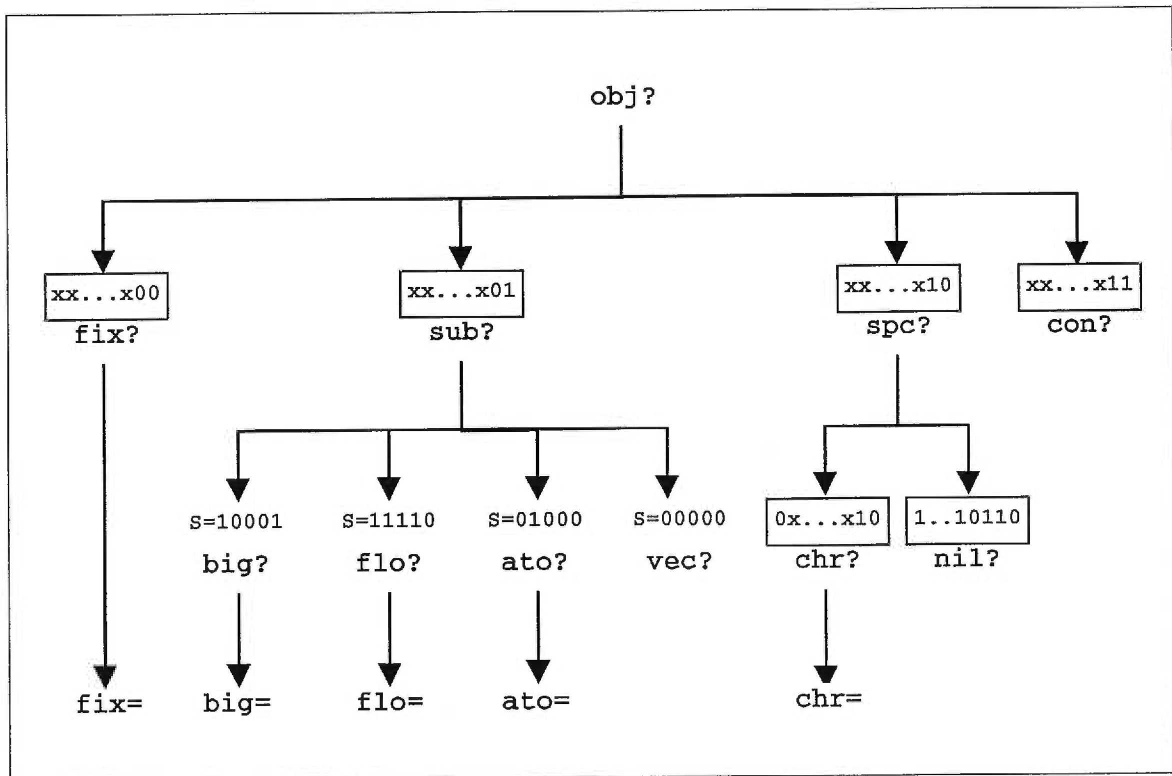

Figure 5.1: Hiérarchie des tests sur les données

Cette hiérarchie couvre donc l'ensemble des données Erlang et UE et démontre la structure de ces dernières. Comme nous le verrons plus tard, plusieurs règles de prédiction pourront être déduites à partir de ce diagramme.

5.1.2 Sémantique des expressions de test

Les expressions de test représentent l'information contenue dans les expressions `ecase` de filtrage sous une forme adaptée à la prédiction des tests. Ces expressions sont évaluées à la compilation et leurs résultats sont des expressions de tests simplifiées qui peuvent ensuite être compilées de façon directe vers du code Scheme.

5.1.3 Syntaxe

Les expressions de test sont construites à partir de constantes, de tests Scheme, de liaisons de variables, d'opérateurs de jonction et de constructions dépendant de la prédiction. Nous présentons donc ici ces éléments afin de bien définir la syntaxe de ces expressions. Il est à noter que ces expressions sont dénotées en langage Scheme.

5.1.3.1 Constantes

Les patrons contradictoires et les patrons tautologiques ou universels sont représentés par les constantes `#f` et `#t` respectivement. Ces expressions de tests indiquent d'entrée de jeu que la clause refuse ou accepte le sélecteur.

5.1.3.2 Tests

Les tests sont représentés par leurs expressions Scheme correspondantes dans lesquelles les variables liées sont toujours accessibles au moment de leur évaluation. Ces tests peuvent être unaires ou binaires.

5.1.3.3 L'opérateur `bind`

Les expressions formées avec l'opérateur `bind` acceptent le sélecteur et indiquent que le nom de variable donné comme premier argument est lié à la valeur de l'expression Scheme donnée comme deuxième argument. Par exemple, si la variable `X` est libre après l'évaluation du sélecteur, l'expression de test correspondant au patron `X` est `(bind ^x sel)`. Il est à noter que la liaison est effectuée à la compilation puisque les expressions de test sont évaluées à cette étape. Seules les liaisons utiles au corps choisi seront effectuées à l'exécution et auront une portée comprenant le corps choisi ainsi que la continuation d'exécution (le reste de la fonction).

5.1.3.4 L'opérateur `bnd?`

L'opérateur `bnd?` prend une variable en argument et détermine si la variable est présentement accessible. Une variable est accessible si elle fait partie des variables liées après l'évaluation du sélecteur ou bien si une expression `bind` avec la même variable a été évaluée précédemment.

5.1.3.5 Opérateurs `and` et `all`

Les expressions formées avec les opérateurs `and` et `all` acceptent le sélecteur si les expressions de tests qu'elles contiennent acceptent ce dernier. L'opérateur `and` diffère de l'opérateur `all` en ce sens qu'il impose un ordre d'évaluation de gauche à droite court-circuité.

5.1.3.6 Opérateurs `or` et `one`

Par opposition aux opérateurs `and` et `all`, les expressions formées avec les opérateurs `or` et `one` acceptent le sélecteur si au moins une des expressions de tests qu'elles contiennent acceptent ce dernier. Encore ici, l'opérateur `or` diffère de l'opérateur `one` en ce sens qu'il impose un ordre d'évaluation de gauche à droite court-circuité.

5.1.3.7 Opérateur `not`

Les expression formée avec l'opérateur unaire `not` acceptent le sélecteur si ce dernier n'est pas accepté par l'expression de test qu'elles contiennent.

5.1.3.8 Forme `if3`

Cette forme spéciale de prédiction prend quatre expressions de test comme arguments:

```
(if3 <E> <T> <F> <U>)
```

L'évaluation (partielle) de cette expression de test est effectuée en tentant d'abord de prédire le résultat de <*E*>. S'il est prédit que <*E*> accepte, on évalue <*T*>, s'il est prédit que <*E*> n'accepte pas, on évalue <*F*> et sinon, on évalue <*U*>. Cette forme permet donc

de créer des règles de dépendance entre les expressions de test. Par exemple, soit l'expression de test suivante qui vérifie si le sélecteur est une liste:

```
(erl-list? sel)
```

Il est possible de remplacer cette expression par une expression équivalente plus complexe qui peut exploiter l'information contextuelle déjà accumulée:

```
(if3 (erl-cons? sel)
     (erl-list? (erl-unsafe-tl/1 sel))
     (erl-nil? sel)
     (erl-list? sel))
```

Cette expression permet donc à notre système d'optimiser l'évaluation de l'expression d'origine en fonction de la prédiction d'un autre test. En effet, si nous voulons savoir si le sélecteur est une liste et que nous savons qu'il est une paire, il nous suffit de tester si la queue de cette paire est une liste. Par contre, si nous savons que le sélecteur n'est pas une paire, nous n'avons qu'à tester s'il est la liste vide. Finalement, si nous ne savons pas si le sélecteur est une paire ou non, nous devons alors effectuer le test initial.

5.1.3.9 Opérateurs `kt` et `kf` *(known true, known false)*

Ces opérateurs unaires ne sont qu'un enrobage syntaxique permettant d'exprimer certains idiomes `if3`. Leur sens est défini comme suit:

```
(kt <E>)    ≡    (if3 <E> #t #f #f)
(kf <E>)    ≡    (if3 <E> #f #t #f)
```

L'opérateur `kt` accepte donc s'il est prouvé que son argument accepte et l'opérateur `kf` accepte s'il est prouvé que son argument n'accepte pas.

5.1.3.10 La forme `any`

Cette forme spéciale permet la prédiction basée des tests binaires déjà effectués. On peut l'utiliser par exemple pour savoir s'il a déjà été montré que le terme `T` était égal à un atome quelconque. On s'intéressera alors à vérifier s'il existe un test de la forme `(erl-ato= T <x>)` qui pourrait être vrai. La forme `any` permet donc de faire de telles prédictions en spécifiant trois arguments :

```
(any <Test> <TestArg1> <Expr>)
```

L'argument `<Test>` spécifie le nom du test (dans notre exemple il s'agirait de `erl-ato=`), `<TestArg1>` spécifie le premier argument du test (`T` dans notre exemple) et le dernier est une expression de test pouvant contenir la variable spéciale `$any` (ce pourrait être `(erl-ato= T $any)`). Cette forme est évaluée en vérifiant si le test a déjà été vérifié. Si c'est le cas, l'expression `<Expr>` est évaluée en remplaçant les occurrences de `$any` qu'elle contient par le terme deuxième argument du test déjà effectué, sinon la forme n'accepte pas. Grâce à cette forme, nous pouvons par exemple prédire que le sélecteur est un entier s'il a été démontré qu'il était égal à un quelconque entier fixe en remplaçant l'expression de test `(erl-int? sel)` par l'expression de test suivante:

```
(one (any erl-fix= sel (erl-fix= sel $any))
     (erl-int? sel))
```

Ainsi par exemple, si le test `(erl-fix= sel 1)` a déjà été vérifié, la forme `any` associera $any à 1 et son évaluation sera remplacée par celle de l'expression `(erl-fix= sel 1)`, qui peut alors être trivialement prédite.

5.1.3.11 L'opérateur `exec`

Cet opérateur se comporte exactement comme la fonction `apply` de Scheme. Il sera alors supposé que son évaluation retourne une expression de test valide. De telles expressions de test sont principalement utiles pour implanter des mécanismes de prédiction récursifs. En se rappelant que l'évaluation partielle des expressions de tests est effectuée à la compilation, une construction comme la suivante devient utile pour prédire une expression de garde `G` comme `(erl-fix= (erl-length/1 sel) 2)`:

```
(erl-fix= (or (and (kt (erl-nil? A))
                   0)
              (and (kt (erl-con? A))
                   (erl-fix+/2 1
                            (exec <F> (erl-tl/1 A)))
              (erl-length/1 sel))
          2)
```

Où `<F>` est une fermeture qui représente la fonction à un paramètre qui génère l'expression de test mentionnée en exemple. Ceci permet d'éviter l'expansion infinie de l'expression de test.

5.1.4 Règles de construction

Nous présentons ici les règles de construction permettant de transformer la représentation des clauses UE vers une structure de donnée plus adéquate pour le traitement. Cette structure contient deux expressions de test, soient une expression L pour les liaisons de variables et une expression A pour l'acceptation du sélecteur. L'expression L sert principalement à déterminer le moment de liaison des variables tandis que l'expression A contient toute l'information requise pour effectuer le choix de la clause.

Puisque certaines conditions doivent être remplies avant de pouvoir accéder aux valeurs des liaisons (par exemple, on ne peut lier une variable au premier élément d'un tuple sans savoir si l'on a vraiment un tuple), il est important de savoir à tout moment quelles sont les variables accessibles. L'évaluation de l'expression de test correspondant aux liaisons est spéciale et retourne deux valeurs, soient les liaisons accessibles et une nouvelle expression de test (qui contient l'information requise pour les autres liaisons). Si cette dernière accepte, cela signifie qu'il ne reste plus de liaison inaccessible. Par exemple, soit la clause suivante, en supposant la variable X libre après l'évaluation du sélecteur:

```
[X] when is_integer(X) -> (B1f)
```

L'expression L de liaison correspondante est la suivante:

```
(and (erl-cons? sel)
     (bind ^x (erl-unsafe-hd/1 sel)))
```

Par sa part, l'expression A d'acceptation est:

```
(and (erl-cons? sel)
     (all (erl-nil? (erl-unsafe-tl/1 sel))
          (erl-int? ^x)))
```

Dès le départ, l'évaluation de l'expression L retourne un environnement vide et la nouvelle expression L reste la même. Ceci signifie qu'aucune liaison n'est encore accessible. Une fois prouvé que le sélecteur est une paire, l'évaluation de L retourne un environnement composé de la liaison `(^x (erl-unsafe-hd/1 sel))` ainsi que la constante `#t`, indiquant qu'il ne reste plus de liaison à effectuer.

Voici un résumé des différentes fonctions de transformation qui seront utilisées dans cette section:

LT	:	Transformation des patrons en expressions L.
AP	:	Transformation des patrons en expressions A.
AG	:	Transformation des gardes en expressions A.
GT	:	Transformation des expressions de garde en expression de test.
GC	:	Transformation des expressions de garde en expression Scheme équivalente.
AT	:	Transformation des clauses en expressions A (à partir du patron, du garde et du sélecteur).

Figure 5.2: Résumé des fonctions de transformation

5.1.4.1 L'expression de test des liaisons

La construction des expressions L se fait aisément à partir du patron P non augmenté d'un garde, du sélecteur S et de l'ensemble BV des variables liées, obtenu par une analyse statique de l'arbre de syntaxe. Les règles de construction des expressions L sont présentées sous la forme d'une fonction de transformation LT à la figure 5.3.

```
LT(k,S)             →   #t
LT(_,S)             →   #t
LT(X,S)  | X∈BV     →   #t
LT(X,S)  | X∉BV     →   (bind X S)
LT([P1|P2],S)       →   (and (erl-cons? S)
                              (all LT(P1,(erl-unsafe-hd/1 S))
                                   LT(P2,(erl-unsafe-tl/1 S))))
LT({P1,…,Pn},S)     →   (and (erl-tuple-size? S n)
                              (all LT(P1,(erl-unsafe-element/1 1 S))
                                   …
                                   LT(Pn,(erl-unsafe-element/1 n S))))
LT(P1=P2,S)         →   (all LT(P1,S) LT(P2,S))
```

Figure 5.3: Règles de construction des expressions L

Il est à noter que la transformation V produit des expressions de test qui n'utilisent qu'un sous-ensemble très limité de la syntaxe disponible. L'évaluation spéciale de ces expressions est donc assez simple à implanter et leur traitement est aussi simplifié.

5.1.4.2 L'expression de test de l'acceptation du sélecteur

La construction des expressions A est réalisée à l'aide du patron augmenté P, du garde G, du sélecteur S et de l'ensemble BV. L'expression de garde rend donc la construction un peu plus complexe que celle des expressions L. Nous présentons les règles de construction des expressions A sous la forme de cinq fonctions de transformation. D'abord, la figure 5.4 présente la transformation AP qui prend le patron et le sélecteur.

```
AP(<k:fixnum>,S)     →  (erl-fix= S k)
AP(<k:bignum>,S)     →  (and (erl-sub? S) (erl-big? S) (erl-big= S k))
AP(<k:flonum>,S)     →  (and (erl-sub? S) (erl-flo? S) (erl-flo= S k)
AP(<k:atom>,S)       →  (erl-ato= S k)
AP(<k:char>,S)       →  (erl-chr= S k)
AP([],S)             →  (erl-nil? S)
AP(_,S)              →  #t
AP(X,S) | X∈BV       →  (erl-=:= X S)
AP(X,S) | X∉BV       →  #t
AP([P1|P2],S)        →  (and (erl-cons? S)
                             (all A(P1,(erl-unsafe-hd/1 S))
                                  A(P2,(erl-unsafe-tl/1 S))))
AP({P1,…,Pn},S)      →  (and (erl-tuple-size? S n)
                             (all AP(P1,(erl-unsafe-element/1 1 S))
                                  …
                                  AP(Pn,(erl-unsafe-element/1 n S))))
AP(P1=P2,S)          →  (all AP(P1,S) AP(P2,S))
```

Figure 5.4: Règles de construction des expressions A à partir des patrons

De son coté, la transformation AG présentée à la figure 5.5 prend le garde associé au patron comme argument. Il est à noter que les règles présentées ici ne sont pas exhaustives puisqu'Erlang possède un ensemble plus élaboré d'opérateurs et de fonctions prédéfinies à utiliser dans les gardes.

```
AG(true)          →  #t
AG(E1 =:= E2)     →  (and (all GT(E1) GT(E2))
                          (erl-=:= GC(E1) GC(E2)))
AG(E1 < E2)       →  (and (all GT(E1) GT(E2))
                          (erl-< GC(E1) GC(E2)))
AG(is_char(E))    →  (and GT(E) (erl-chr? GC(E)))
AG(is_list(E))    →  (and GT(E) (erl-list? GC(E)))
```

Figure 5.5: Règles de construction des expressions A à partir des gardes

La transformation GT prend une expression de garde et la transforme en expression de test qui doit s'évaluer à #t s'il est possible de passer à l'évaluation de l'expression de

garde sans erreur. De son coté, la transformation GC prend aussi une expression de garde, mais la transforme en expression de test spéciale qui peut s'évaluer à autre chose que #t ou #f. Cependant, son évaluation doit toujours pouvoir être faite sans effectuer de test. Cette expression ne contiendra donc que des formes prédictives et des expressions Scheme. Quelques règles des fonction GT et GC sont présentées à la figure 5.6 afin de clarifier les choses.

```
GT(k)          →  #t
GC(k)          →  k
GT(X)          →  (bnd? ^x)
GC(X)          →  ^x
GT(E1+E2)      →  (and (all GT(E1) GT(E2))
                       (or (kt (all (erl-num? GC(E1))
                                    (erl-num? GC(E2)))
                           (erl-tst-+/2 GC(E1) GC(E2))))
GC(E1+E2)      →  (erl-unsafe-+/2 GC(E1) GC(E2))
```

Figure 5.6: Fonctions de construction des expressions de garde

Finalement, la transformation AT prend le patron P, le garde G et le sélecteur S pour produire l'expression de test finale. Cette unique règle, présentée à la figure 5.7, définit la construction des expressions AT.

```
AT(P,G,S)   →   (all AP(P,S) AG(G))
```

Figure 5.7: Règle de construction des expressions A

Voici maintenant un exemple. Soit le patron augmenté suivant:

```
[X|Y] when X=:=Y+1
```

L'expression de test L associée est alors:

```
(and (erl-cons? sel)
     (all (bind ^x (erl-unsafe-hd/1 sel))
          (bind ^y (erl-unsafe-tl/1 sel))))
```

Pour sa part, l'expression de test A est:

```
(all (erl-cons? sel)
     (and (all (bnd? ^x)
               (and (all (bnd? ^y) #t)
                    (or (kt (all (erl-num? ^y)
                                 (erl-num? 1)))
                        (erl-tst-+/2 ^y 1))))
          (erl-=:= ^x (erl-unsafe-+/2 ^y 1))))
```

5.1.5 Extraction des liaisons par réduction des expressions L

Les expressions L renferment l'ensemble des liaisons pouvant être effectuées au moment de leur évaluation. La nature simple des expressions L permet une extraction des liaisons avec très peu d'effort. Il s'agit d'abord de réduire l'expression en tentant de prédire les tests. Une fois ce travail fait, l'obtention de la liste des nouvelles liaisons peut se faire à l'aide des quelques règles présentées à la figure 5.8.

```
XL(<test>)             → ()
XL((bind X E))         → ((X E))
XL((all E1 … En))      → (XL(E1) … XL(En))
XL((and E1 … En))      → XL(E1)
```

Figure 5.8: Règles d'extraction des tests

5.1.6 Extraction des tests par évaluation partielle des expressions A

Afin d'évaluer des expressions de test lors de l'exécution, il faut d'abord être en mesure d'extraire les tests présents dans les expressions A. Les expressions de test présentées ici comportent deux types distincts de constructions. En effet, certaines constructions peuvent être évaluées à la compilation tandis que d'autres ne peuvent être évaluées qu'à l'exécution. Dans l'exemple précédent, l'expression (erl-cons? sel) ne peut être déterminée qu'à l'exécution si le sélecteur n'est pas une constante, tandis que l'expression (bnd? ^x) peut être déterminée à la compilation, durant le processus de filtrage.

L'ensemble des expressions de test pouvant être évaluées à la compilation est vaste. D'abord, l'ensemble des tests pouvant être prédits peuvent être évalués lors de la compilation. Par exemple, si nous savons que le sélecteur est une liste mais pas la liste vide, nous pouvons inférer qu'il est une paire. Ensuite, l'ensemble des constructions prédictives, c'est-à-dire les expressions formées par les formes if3, kt, kf, any et exec,

peuvent être évaluées. Finalement une exception: les expressions formées avec l'opérateur `bnd?` ne peuvent pas nécessairement être évaluées. En effet, si la variable n'est pas encore liée, il est possible qu'elle le devienne un peu plus tard dans le processus. Donc, tant que la variable n'est pas liée, cette expression reste après l'évaluation partielle.

Une fois qu'une expression de test à été évaluée partiellement, l'expression résultante ne comporte donc que des tests, des constantes et des expressions `all`, `and`, `one`, `or`, `not` ou `bnd?`. L'extraction de la liste des tests pouvant être effectués à l'exécution devient alors très simple et se résume par les règles présentées à la figure 5.9.

```
XA(<test>)              →  (<test>)
XA(k)                   →  ()
XA((bnd? X))            →  ()
XA((all E1 … En))       →  (XA(E1) … XA(En))
XA((and E1 … En))       →  XA(E1)
XA((one E1 … En))       →  (XA(E1) … XA(En))
XA((or E1 … En))        →  XA(E1)
XA((not E))             →  XA(E)
```

Figure 5.9: Règles d'extraction des tests

Grâce aux tests maintenant extraits des expressions, il devient possible de compiler les clauses vers Scheme. La prochaine section traite des arbres de tests qui permettent de représenter le code Scheme sous une forme pratique pour les optimisations.

5.2 L'arbre de décisions

Nous avons vu précédemment que la transformation `S` est en fait une série de tests effectués via la construction `if` de Scheme. La syntaxe même de cette forme conditionnelle produit donc un arbre binaire avec le test comme nœud, les alternatives comme branches et l'appel des fonctions des corps comme feuilles. Il sera donc présenté dans cette section notre méthode de génération de ces arbres à partir des expressions `L` et `A` obtenues à partir des clauses de filtrage.

C'est à cette étape que sont effectués la fusion des expressions communes ainsi que le retrait de la duplication de code. D'une part, nous prendrons soin de maintenir une table

des expressions déjà calculées afin d'effectuer la fusion de ces dernières. D'autre part, une table des nœuds déjà construits sera conservée afin de construire un DAG (*directed acyclic graph*) plutôt qu'un arbre de test, ce qui permettra d'éliminer la duplication de code.

5.2.1 Génération

La génération de l'arbre commence par l'extraction de l'ensemble des tests qu'il est possible d'effectuer à partir des expressions `A` et de l'ensemble des liaisons disponibles à partir des expressions `L`. Une fois l'ensemble des tests disponibles, le choix du test à effectuer en premier dépendra de l'algorithme d'ordre des tests qui sera choisi. Le prochain chapitre traite de ce problème en détail. Dès que le choix du test à effectuer est arrêté, le processus recommence pour les deux cas possibles, qui seront les branches du nœud formé par ce test. Le procédé est donc récursif et l'information au sujet des résultats des tests est conservée au fil des itérations sous la forme de deux listes contenant d'une part les vérités (l'ensemble des tests avec un résultat positif) et d'autre part les contradictions (l'ensemble des tests avec un résultat négatif). Comme nous le verrons plus loin, ces informations seront utiles pour la prédiction des tests.

Essentiellement, l'arbre est construit à partir des tests qu'il est possible d'effectuer. Dès que la première clause accepte le sélecteur, ce qui peut être détecté lorsque son expression de test s'évalue à `#t`, cela signifie qu'une feuille est atteinte. Lorsqu'une clause refuse le sélecteur, cette clause est alors retirée et l'évaluation continue avec le reste des clauses à prouver. Finalement, si toutes les clauses refusent le sélecteur dans une branche sous un test, cela signifie que ce dernier est inutile et le nœud est alors remplacé par celui au bout de l'autre branche. Ceci est possible puisque nous avons posé plus tôt que le filtrage sous UE est invariablement exhaustif. De la même manière, dès que l'ensemble des clauses restantes est un singleton, on peut supposer que la clause accepte. Voici un exemple pour clarifier le procédé. Soit la transformation `S` suivante :

```
S({a,b} -> (B1f),
  {_,c} -> (B2f))
```

Voici l'expression `A` de la première clause :

```
(and (erl-tuple? sel)
     (all (erl-ato= (erl-unsafe-element/2 1 sel) 'a)
          (erl-ato= (erl-unsafe-element/2 2 sel) 'b)))
```

Voici l'expression `A` de la deuxième clause :

```
(and (erl-tuple? sel)
     (erl-ato= (erl-unsafe-element/2 2 sel) 'c))
```

Maintenant, pour la génération de l'arbre, considérons l'ensemble des tests qu'il est possible d'effectuer au départ. Les deux expressions `A` ne permette d'effectuer que le test `(erl-tuple? sel)`, il n'y a donc pas de choix à faire et le premier nœud de l'arbre est le test `(erl-tuple? sel)`. Maintenant, l'ensemble des clauses restantes si le test est vrai reste le même qu'au départ mais l'ensemble des clauses restantes si le test est faux est vide. Puisque le filtrage doit être exhaustif, ceci implique que le test `(erl-tuple? sel)` est inutile et l'on peut continuer l'évaluation dans l'autre branche.

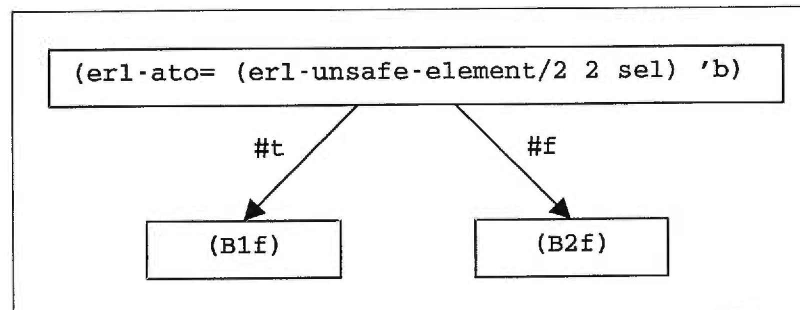

Figure 5.10: Arbre de décision généré

L'ensemble des tests pouvant être effectués est alors constitué des trois tests qui inspectent les éléments du 2-tuple. Supposons que nous choisissions le test `(erl-ato= (erl-unsafe-element/2 2 sel) 'b)` (le sixième chapitre est consacré à l'ordre des tests). Dans le cas où le test est vrai, seule la première clause reste, ce qui signifie que nous avons atteint une feuille qui consiste en l'évaluation du corps `(b1f)`. Dans le cas où le test est faux, seule la deuxième clause reste, ce qui signifie encore une fois que nous avons atteint une feuille qui consiste en l'évaluation du corps `(b2f)`. L'arbre de décision généré est montré à la figure 5.10.

5.2.2 Retrait de la duplication de code

La duplication de code est évitée par le maintien d'une table de nœuds. Chaque nœud créé est mis dans un tableau et est ensuite référé par son indice. Il est alors possible, lors de la création d'un nœud dupliqué, de référer à la première instance créée. L'élimination du code pourra alors se produire à la génération de code Scheme, où l'on pourra décider de placer le nœud dans une fonction sans argument (*thunk*) dont l'appel remplacera le code du test. Par exemple, soit la transformation S suivante:

```
S({a,b} -> (B1f),
  {c,_} -> (B2f),
  {_,_} -> (B3f))
```

La génération de l'arbre de décision construira alors une table de nœuds dont le point d'entrée est son premier élément, comme on peut le voir dans la figure 5.11.

Indice	Nœud
0	`(erl-ato= (erl-unsafe-element 1 sel) 'a), 1, 2`
1	`(erl-ato= (erl-unsafe-element 2 sel) 'b), 3, 2`
2	`(erl-ato= (erl-unsafe-element 1 sel) 'c), 4, 5`
3	`(B1f)`
4	`(B2f)`
5	`(B3f)`

Figure 5.11: Exemple de table de nœuds

La table ainsi générée peut alors se comparer à un DAG et à une machine à état fini dont les états finaux sont les appels des corps. Dans l'exemple, nous voyons que le nœud numéro 2 est référencé à deux reprises, ce qui permettra le retrait de la duplication de code lors du passage à Scheme.

5.2.3 Fusion des expressions communes

Une autre table est utilisée lors de la construction de l'arbre de décisions et contient l'ensemble des expressions déjà évaluées. Chaque test est composé de plusieurs sous-expressions qui sont alors toutes mises dans la table et référencées encore une fois par leur indice. L'utilisation de cette table permet donc la fusion des expressions communes lors de la génération du code Scheme. À cette étape, s'il est admis qu'une expression est référencée

dans chaque branche d'un nœud, il devient alors possible de générer une liaison avec une variable temporaire pour contenir le résultat.

Cette stratégie peut cependant causer un problème si l'on tente d'évaluer une expression qui comporte des éléments qui ne sont pas encore accessibles au niveau du nœud où l'on veut faire la liaison. Par exemple, il est impossible d'accéder aux éléments d'une paire sans que le test de type n'ait été effectué pour s'assurer qu'on a réellement affaire à une paire comme le montre cet exemple :

```
(if (erl-ato= <expr1> 'a)
    (if (erl-con? <expr2>)
        (erl-hd <expr2>)
        <...>)
    (if (erl-con? <expr2>)
        (erl-+/2 (erl-hd <expr2>) 1)
        <...>))
```

Dans cet exemple, bien que l'expression `(erl-hd <expr2>)` soit commune aux deux branches du premier nœud, il n'est pas possible de l'évaluer à cet endroit puisqu'un test est requis.

Notre implantation utilise une règle conservatrice qui permet en outre de s'assurer que les liaisons ne sont effectuées que si leur valeur est utilisée. Nous distinguons deux cas qui nous permettent de placer l'évaluation d'une expression juste avant un test. D'abord, lorsqu'une expression est référencée à la fois dans le test d'un nœud et dans un de ses descendants, nous sommes assurés que son évaluation peut être faite. Deuxièmement, si une expression est référencée dans les deux tests à la tête de chaque branches d'un nœud, c'est que le test de ce nœud n'est pas requis pour l'évaluation de l'expression.

5.3 Prédictions

Pendant la construction de l'arbre de décision, de l'information utile concernant la structure du sélecteur est accumulée. Comme nous l'avons déjà mentionné, nous conservons l'ensemble des tests déjà effectués dans deux ensembles comprenant les vérités et les mensonges. À partir de ces deux ensembles et d'un certain nombre de règles, nous

pouvons prédire l'issue de certains tests, ce qui nous permet d'éviter de les évaluer à l'exécution. Cette section présente donc l'algorithme permettant la prédiction d'un test en fonction des tests déjà effectués sous la forme d'une fonction `P`. La fonction `P` prend comme paramètres une expression de test, l'ensemble des vérités `T`, l'ensemble des mensonges `L`, un ensemble de tests `U` dont ont sait qu'on ignore le résultat et finalement un environnement d'évaluation `E` qui renferme l'ensemble des liaisons disponibles. Cette fonction retourne une nouvelle expression de test modifiée qui élimine ou simplifie les tests dont l'issue peut être prédite complètement ou partiellement.

5.3.1 La prédiction des expressions de test composées

La prédiction des expressions de test composées se fait à l'aide de règles dépendantes de l'opérateur utilisé. Nous utilisons ici une syntaxe simplifiée pour la fonction `P` en faisant abstraction des paramètres `T`,`F` et `U` puisqu'ils ne sont pas utilisés directement dans cette section. La version de la fonction P prend donc ici comme paramètres l'expression de test à prédire ainsi que son environnement.

5.3.1.1 L'opérateur `bnd?`

La prédiction d'une expression `bnd?` se fait en vérifiant si la variable est liée dans l'environnement `E`. Si c'est le cas, la fonction `P` retourne `#t`, sinon, l'expression `bnd?` reste inchangée. En effet, retourner `#f` signifierait que la variable ne pourrait jamais être liée:

```
P((bnd? <Var>),E) si <Var> ∈ E      ≡  #t
P((bnd? <Var>),E) si <Var> ∉ E      ≡  (bnd? <Var>)
```

5.3.1.2 Opérateurs `all` et `one`

La prédiction des expressions formées avec les opérateurs `all` et `one` est faite en tentant de prédire l'ensemble de leurs sous-expressions. Dans le cas de l'opérateur `all`, les sous-expressions dont la valeur est prédite à `#t` seront éliminées, tandis que celles dont la valeur est prédite à `#f` permettront la prédiction de toute l'expression à `#f`. Les sous expressions restantes seront alors remises dans une expression `all` et s'il ne reste aucune expression, la prédiction de toute l'expression sera `#t` :

```
P((all <E1> ... <Ei> ... <En>),E) si P(<Ei>,E) = #f    ≡  #f

P((all <E1>),E) si P(<E1>,E) = #t                      ≡  #t

                                                           P((all <E1>
                                                                  ...
                                                                  <Ei-1>
P((all <E1> ... <Ei> ... <En>),E) si P(<Ei>,E) = #t    ≡          <Ei+1>
                                                                  ...
                                                                  <En>),
                                                             E)

                                                           (all P(<E1>,E)
Sinon, P((all <E1> ... <En>),E)                        ≡        ...
                                                                P(<En>,E))
```

Le cas de l'opérateur one est tout à fait symétrique.

5.3.1.3 Opérateurs and et or

La prédiction des expressions formées avec les opérateurs and et or est faite en tentant d'évaluer leur première sous-expression. Dans le cas de l'opérateur and, si la première sous-expression est prédite à #t, on retourne la prédiction de l'expression all formé du reste des sous-expressions. Si la première est prédite à #f, toute l'expression est prédite à #f. Dans les autres cas, on reforme une expression and avec la prédiction de la première sous-expression et le reste des sous-expressions non prédites. Ces dernières ne sont pas prédites comme dans l'expression all afin de permettre des constructions conditionnelles aux prédictions:

```
P((and <E1> <E2> ... <En>),E) si P(<E1>,E) = #f       ≡  #f

P((and <E1>),E) si P(<E1>,E) = #t                     ≡  #t

                                                          P((and <E2>
                                                                 ...
P((and <E1> ... <Ei> ... <En>),E) si P(<E1>,E) = #t   ≡          <En>),
                                                            E)

                                                          (and P(<E1>,E)
                                                               <E2>
Sinon, P((and <E1> <E2> ... <En>),E)                  ≡        ...
                                                               <En>)
```

Le cas de l'opérateur or est encore ici parfaitement symétrique.

5.3.1.4 Opérateur `not`

La prédiction des expressions formées avec l'opérateur `not` utilise le résultat de la prédiction de sa sous-expression. Si le résultat est booléen, `P` inverse et retourne cette valeur, sinon une nouvelle expression `not` est construite avec cette valeur :

`P((not <T>),E)` si `P(<T>,E) = #f`	≡	`#t`
`P((not <T>),E)` si `P(<T>,E) = #t`	≡	`#f`
Sinon, `P((not <T>),E)`	≡	`(not P(<T>,E))`

5.3.1.5 Forme `if3`

La forme `if3` est très simple à prédire. Il s'agit d'abord de tenter de prédire de façon récursive son premier argument. Si le résultat est `#t`, on applique `P` au deuxième argument. Si le résultat est `#f`, on applique `P` au troisième argument. Dans les autres cas, on applique `P` au dernier argument :

`P((if3 <T1> <T2> <T3> <T4>),E)` si `P(<T1>,E) = #t`	≡	`P(<E2>,E)`
`P((if3 <T1> <T2> <T3> <T4>),E)` si `P(<T1>,E) = #f`	≡	`P(<E3>,E)`
Sinon, `P((if3 <T1> <T2> <T3> <T4>),E)`	≡	`P(<E4>,E)`

5.3.1.6 La forme `any`

Pour prédire une expression utilisant la forme spéciale `any`, il faut d'abord parcourir l'ensemble `T` des vérités à la recherche d'un test constant sur la bonne expression à tester. Pour chacun de ces tests trouvés, nous tentons de prédire le troisième argument de la forme `any` en prenant soin de lier la variable `$any` au deuxième argument du test. Si une de ces prédiction donne `#t`, la prédiction de toute l'expression donne `#t` sinon `#f` :

`P((any <N> <A> <T>),E)`	≡	`(one P(<T>,E1) ... P(<T>,En))` où *Ei* est l'environnement `E` augmenté de la variable $any telle que `P((<T> <A> $any),E) = #t`, n étant le nombre de valeurs différentes que `$any` puisse prendre.

5.3.2 La prédiction des tests

Les expressions de test formées par un seul test sont prédites d'abord en vérifiant si le test a déjà été effectué (dans ce cas, le résultat de cette évaluation représente la prédiction

du test). Si ce n'est pas le cas, nous déclarons ce test impossible à prédire en le plaçant dans l'ensemble des inconnus. On tente alors une expansion de ce test vers une expression de test équivalente qui pourra profiter de l'information à propos d'autres tests pouvant permettre de prédire ou de simplifier le test. La section suivante traite du problème de l'expansion des tests.

Il y a trois cas où un test est prédit sans utiliser de règle spéciale. Si le test fait partie de l'ensemble des vérités, sa prédiction est `#t`. Par contre, si le test fait partie de l'ensemble des mensonges, sa prédiction est `#f`. Finalement, si le test fait partie de l'ensemble des inconnus, la prédiction de ce test est impossible et la fonction `P` retourne alors ce même test.

5.4 L'expansion des tests

L'expansion des tests vers des expressions de test explicitant les liens entre les différentes structures de données permet de compléter notre algorithme de prédiction de tests. Trois catégories d'expansion sont présentées dans cette section. D'abord, l'expansion des tests de base, qui comprennent les tests de type de base et les tests de comparaison avec des constantes. Ensuite, l'expansion des tests de comparaison entre objets et sur les structures plus élaborées comme les listes. Finalement, la génération des expressions de test spéciales permettant la simplification des expressions incluses dans les gardes.

5.4.1 Tests de base

La hiérarchie présentée à la figure 4.1 permet de voir des liens de dépendance entre les différents tests de types et de constantes. En effet, dans la hiérarchie présentée, tous les objets sont représentés dans les plus jeunes fils. De plus, chaque objet ne possédant qu'un seul emplacement bien précis, nous pouvons établir trois règles importantes. La première concerne le lien de fraternité, tandis que les deux autres concernent le lien de parenté.

Le lien de fraternité entre deux tests permet d'emblée de déterminer qu'ils ne peuvent s'évaluer positivement pour un seul objet. En effet, un objet ne peut être son propre frère. Ceci implique donc premièrement que si l'on peut prédire que le frère d'un

test `T` est vrai, on peut prédire que `T` est faux. De plus, puisque tous les objets sont représentés dans la hiérarchie, si tous les frères de `T` sont faux et que le parent est vrai, nous pouvons en déduire que `T` est vrai. La règle d'expansion générale associée est donc:

```
(if3 (one <frères de T>) #f (kt <parent de T>) T)
```

Le lien de parenté entre deux tests permet de constater qu'un test n'a qu'un seul parent et conséquemment, la valeur de vérité de ce dernier est nécessairement celle du test en question. La règle d'expansion générale associée pour un test `T` est donc:

```
(if3 <parent de T> T #f T)
```

Ce même lien implique aussi que le parent d'un test positif ne peut être que positif à son tour. La règle d'expansion générale associée pour un test `T` est alors:

```
(one (kt (one <fils de T>)) T)
```

L'ensemble de ces trois règles peut être combiné pour obtenir une seule expression de test très utile:

```
(one (kt (one <fils de T>))
     (if3 <parent de T>
          (if3 (one <frères de T>) #f #t T)
          #f
          T))
```

La prédiction de cette nouvelle expression tient donc compte des relations de dépendance entre les éléments de la hiérarchie des types de données. Il est à noter qu'avant d'être étendu, un test `T` est toujours ajouté à l'ensemble des tests dont la valeur est inconnue, ce qui permet à l'algorithme de prédiction de ne pas étendre infiniment les tests. En effet, la prédiction d'un test dont le résultat est supposé inconnu n'est jamais effectuée.

5.4.2 Tests de comparaison et sur structures complexes

La non linéarité des patrons et les gardes permettent d'ajouter des comparaisons d'objets ainsi que des tests sur des structures plus complexes telles les listes. Les règles d'expansion pour ces tests sont différentes de celles des tests de base puisque la hiérarchie ne s'adapte pas à leur cas. Par exemple, les tests de comparaisons d'égalité effectuent des

tests de valeur différents en fonction du type. De leur coté, les listes sont des objets qui n'ont pas d'emplacement unique dans la hiérarchie.

5.4.2.1 Comparaisons

Sur le plan de la comparaison d'inégalité `(erl-< A B)`, nous pouvons encapsuler sa propriété transitive dans l'expansion suivante:

```
(one (any erl-< A (erl-< $any B))
     (and (not (kt (erl-=:= A B))) (erl-< A B)))
```

En effet, si nous avons déjà testé avec succès `(erl-< A C)` et que nous sommes en mesure de prédire que `(erl-< C B)` est vrai, il s'ensuit que `(erl-< A B)` est vrai aussi. De plus, s'il est possible de prouver que `(erl-=:= A B)`, on peut supposer que `(erl-< A B)` est faux.

Il est aussi possible d'encapsuler les propriétés transitives et associatives de la comparaison d'égalité absolue grâce à l'expansion du test `(erl-=:= A B)` vers:

```
(one (kt (erl-=:= B A))
     (any erl-=:= A (erl-=:= $any B))
     (and (not (kt (erl-< A B))) (erl-=:= A B)))
```

Nous voyons donc que grâce à la puissance expressive des expressions de test, il devient facile d'intégrer le savoir relatif aux liens unissant les différents tests à même celles-ci.

5.4.2.2 Structures complexes

Sur le plan des structures complexes, l'expansion des tests est définie différemment selon le cas. Pour les listes, nous avons déjà vu dans le chapitre précédent que le test `(erl-list? A)` peut être étendu vers une expression explicitant sa structure particulière:

```
(if3 (erl-cons? A)
     (erl-list? (erl-unsafe-tl/1 A))
     (erl-nil? A)
     (or (kt (erl-nil? A)) (erl-list? A)))
```

De la même façon, on peut aussi expliciter la structure des chaînes de caractères dans l'expansion du test `(erl-str? A)`. Rappelons que les chaînes caractères de Erlang sont des listes de caractères. L'expansion du test est donc:

```
(if3 (erl-cons? A)
     (if3 (erl-chr? (erl-unsafe-hd/1 A))
          (erl-str? (erl-unsafe-tl/1 A))
          #f
          (erl-str? A))
     (erl-nil? A)
     (or (kt (erl-nil? A)) (erl-str? A)))
```

Les entiers sont aussi au nombre des structures complexes. L'expansion du test `(erl-int? A)` est toutefois très simple:

```
(one (erl-fix? A)
     (and (erl-sub? A) (erl-big? A)))
```

Il est à noter qu'il n'y a aucune forme prédictive dans cette expression, ce qui signifie que le test sur les entiers est tout simplement remplacé par un test plus explicite dans le code, ce qui impose l'application en ligne de ce test. C'est la même chose pour la dernière structure complexe que sont les nombres, dont l'expansion du test `(erl-num? A)` se traduit par:

```
(one (erl-fix? A)
     (and (erl-sub? A)
          (one (erl-big? A) (erl-flo? A))))
```

La puissance expressive des expressions de test permet donc, ici aussi, l'intégration du savoir relatif aux structures complexes de Erlang afin d'améliorer la prédiction ou la simplification des tests.

5.4.3 Génération des expression de test spéciales des gardes

L'élément manquant dans la génération des expressions de test reste la génération des expressions spéciales représentant les expressions contenues dans les gardes. En effet, Erlang permet l'évaluation de certaines expressions dans les gardes. Outre les constantes et les variables, un certain nombre de fonction intégrées (BIFs) sont autorisées. Cependant, leur comportement est différent que dans les applications normales, puisqu'aucune

exception ne doit être lancée. Nous avons donc augmenté notre librairie de BIF pour inclure des versions test de ces dernières. Par exemple, la fonction `erl-tst-length/1` fonctionne comme `erl-length/1`, sauf que si une exception devait avoir lieu, la première retournerait `#f` au lieu de générer une exception. De plus, afin de profiter de notre système de prédiction de tests, nous avons aussi créé des fonctions équivalentes qui supposent leurs arguments corrects. La fonction `erl-unsafe-length/1` calcule donc la longueur sans vérifier si son argument est bel et bien une liste. C'est avec ces fonctions que peuvent être implantées les fonctions `GT` et `GC` décrites plus haut qui permettent respectivement de déterminer l'expression de test et l'expression d'évaluation des expressions de garde. Il est à noter que le mécanisme de fusion des expressions communes est outillé pour reconnaître que les fonctions `unsafe` peuvent toujours être remplacées par les expressions `tst`, le cas échéant.

5.5 Résumé

En bref, nous avons vu que la transformation des clauses Erlang vers les expressions de test permettent la génération d'un arbre de décision efficace puisque les tests effectués peuvent parfois être prédits ou simplifiés. Ainsi, le seul aspect concernant la compilation du filtrage qui reste ouvert à la lumière de ce chapitre est l'ordre dans lequel effectuer les tests. En effet, nous avons montré comment déduire l'ensemble des tests possibles à partir des expressions de test, mais n'avons pas encore parlé du mécanisme de choix du test dans cet ensemble. Ce choix fait l'objet du chapitre suivant.

Chapitre VI

L'ordre des tests

L'ordre dans lequel les tests sont effectués peut influencer l'efficacité de l'arbre de décision. Nous verrons dans ce chapitre les différentes solutions possible à ce problème, ainsi que les résultats en terme de temps d'exécution et de taille d'exécutable qui s'y rattachent afin de comparer. La première stratégie présentée est la plus simple et les résultats obtenus s'apparentent à ceux des algorithmes conçus par Sestoft [SES96], Pettersson [MP99] et Queinnec [QG92]. Nous présenterons par la suite notre solution, inspirée du travail de Baudinet et MacQueen [BM85]. Nous verrons que la détermination du meilleur test à faire doit passer par l'évaluation de certaines heuristiques.

6.1 Étalons utilisés

Afin de quantifier l'efficacité des algorithmes de choix présentés ici, nous avons utilisés quatre étalons *(benchmarks)* qui utilisent le filtrage de façon intensive. Ces programmes Erlang sont présentés à l'annexe 2. Le premier, *epoptest* teste de façon intensive le point d'entrée de la fonction `trans/2` du programme `epop.erl` (une application qui est présenté à l'annexe 3). Le second, *litescm* est un interprète Scheme rudimentaire qui évalue une expression plusieurs fois. Le troisième, *poker* est un programme qui génère et évalue plusieurs mains de poker. Finalement, le dernier, *type* est un petit programme qui détermine le type de plusieurs termes Erlang présents dans une liste.

Nous aurions aimé pouvoir faire des tests sur de vraies applications qui sont présentement utilisées dans la communauté Erlang. Cependant, étant donné que Standard Erlang est encore peu utilisé par rapport à Erlang 4.7, la librairie standard d'exécution de Erlang 4.7 n'a pas encore été portée vers Standard Erlang. La quasi totalité des applications Erlang utilisent cette librairie et c'est pour cette raison que nous ne pouvons encore tester pleinement ces applications et que nos tests peuvent paraître minimaux. L'idée ici étant de démontrer les gains de performance sur le plan du filtrage, nous avons synthétisé des exemples qui l'utilisent intensivement. Toutefois, l'étalon *epoptest* montre les gains pouvant être effectués sur l'évaluation d'une fonction dont la définition se trouve dans une réelle application qui est présentée à l'annexe 3.

6.2 Première stratégie

La première stratégie est excessivement simple et consiste à toujours choisir le premier test accessible provenant de la première clause. Avec notre implantation, cela revient à un ordre d'évaluation *top-down, left-right*, c'est-à-dire une clause à la fois, évaluée de gauche à droite. Bien que cette stratégie soit simple, notre prédiction des tests la rend toutefois très efficace.

Notre solution rejoint donc l'algorithme proposé par Sestoft qui accumule aussi l'information positive et négative sur les tests déjà effectués et s'en sert pour évaluer partiellement ses expressions de test. Nous rejoignons aussi l'algorithme proposé par Pettersson, qui modélise son arbre de décision à l'aide d'une machine à état fini qui est ensuite optimisée par la fusion des états équivalents. Cependant, à la différence de ces algorithmes, notre solution intègre les contraintes des expressions de garde.

6.3 Seconde stratégie

La seconde stratégie est basée sur les travaux de Baudinet et MacQueen [BM85]. Dans leur article, les auteurs présentent un algorithme de choix de test pour le langage ML. Le mécanisme de filtrage du langage ML est différent de celui de Erlang. D'une part, ML est un langage statiquement typé, ce qui permet plusieurs optimisations impossibles au sein d'un langage dynamiquement typé comme Erlang. Par ailleurs, ML ne possède pas de syntaxe permettant la déclaration de contraintes, ce qui en simplifie beaucoup la compilation. L'élaboration d'un algorithme de choix de test pour Erlang à partir de l'algorithme proposé doit donc tenir compte de ces différences.

6.3.1 Objectif

L'algorithme du choix de test proposé se base sur la création d'heuristiques de recherche dans l'arbre de solution que consiste l'ensemble des choix possibles. Cet algorithme tente de réduire la taille de l'arbre de décision. Le but visé ici est la réduction du nombre de nœuds dans l'arbre de décision. Les auteurs ont découvert que ce problème est

NP-complet et ne peut donc être résolu en temps raisonnable avec une recherche exhaustive.

L'objectif de réduction du nombre de nœuds dans l'arbre de décision est en fait la première heuristique présentée par Baudinet et MacQueen. En effet, le lien entre l'efficacité du filtrage et le nombre de nœuds dans l'arbre de décision est existant, mais l'arbre de décision optimal dans un contexte d'utilisation donné n'est pas nécessairement celui ayant le moins de nœuds et il existe plusieurs arbres de décision plus ou moins efficaces ayant le même nombre de nœuds. Nous adhérons toutefois à l'idée derrière cette heuristique, puisque d'une part le contexte d'utilisation est une donnée extrêmement difficile à capturer au moment de la compilation et par ailleurs, la réduction du nombre de nœuds de l'arbre de décision permet d'optimiser tant sur le plan de la taille de l'exécutable que sur celui du temps d'exécution, ce qui rejoint nos objectifs.

La réduction du nombre de nœuds se fait grâce aux choix des tests. Ces choix seront basés sur les différentes heuristiques proposées par Baudinet et MacQueen qui tiennent compte de l'objectif de réduction du nombre de nœuds. Nous présenterons donc ici ces heuristiques, ainsi que leur adaptation à notre contexte de filtrage Erlang.

6.3.2 Heuristique de convenance (*relevance*)

Baudinet et MacQueen définissent le concept de convenance d'un test par rapport à une clause. Un test est considéré convenable à une clause si cette dernière peut être éliminée par le test. Par exemple, le test `(erl-cons? sel)` est convenable à la clause `[X|Y] -> X` puisqu'elle est éliminée si le test est faux. Ce même test est aussi convenable à la clause `[] -> X` puisqu'elle est éliminée si le test est vrai. D'un autre coté, le test `(erl-fix? sel)` n'est pas convenable à la clause `X when is_integer(X) -> X` puisque cette dernière demeure, peu importe le résultat du test. En effet, si le test est faux, le sélecteur pourrait encore être un *bignum*. Cette règle implique par ailleurs qu'aucun test n'est convenable au patron universel.

L'idée ici est de constater que le choix de la clause acceptante nécessite invariablement l'acceptation ou le rejet de la première clause. En effet, tous les chemins dans l'arbre de décision doivent passer par un test qui permet de décider si la première clause est acceptée. Ceci est impliqué directement par la sémantique du filtrage: on ne peut choisir une clause à une position i avant d'avoir prouvé que l'ensemble des clauses $\{j \mid j<i\}$ ne peuvent être choisies. Il devient alors intéressant d'effectuer ce test le plus tôt possible, puisque la façon la plus économique d'insérer un nœud commun à tous les chemins est bien de le placer le plus près possible de la racine.

Pour Baudinet et MacQueen, la valeur de l'heuristique de convenance pour un test donné est la position de la première clause pour laquelle le test est convenable. L'ensemble des tests pour lesquelles la valeur de l'heuristique est la plus petite constitue alors le nouvel ensemble de choix. Si cet ensemble est un singleton, le choix du test est arrêté et sinon la prochaine heuristique est appliquée. Si aucun test ne convient à aucune clause, c'est qu'une feuille de l'arbre de décision a été atteinte.

Notre cas est un peu plus complexe. Puisque Erlang est dynamiquement typé et que parfois deux tests sont requis pour discriminer une clause, il se peut qu'un test ne convienne à aucune clause bien que le choix ne puisse être déterminé. Par exemple, soit la transformation `S` suivante:

```
S(X when is_integer(X) -> (B1f),
  _ -> (B2f))
```

Nous avons l'expression de test suivante pour la première clause:

```
(one (erl-fix? sel)
     (and (erl-sub? sel) (erl-big? sel)))
```

Les tests pouvant être effectués sont donc `(erl-fix? sel)` et `(erl-sub? sel)`. Cependant, aucun de ces tests ne peut discriminer à lui seul la première clause. En effet, qu'un d'eux soit vrai ou faux, la première clause demeure. Ceci est dû au fait que si le sélecteur n'est pas un entier, au moins deux tests doivent être effectués pour le détecter. Ceci découle évidemment du fait que les entiers Erlang ne sont

pas des types de base dans leur représentation sous Gambit. Les entiers font partie d'un type complexe formé par l'union des entiers à taille fixe et des grands entiers.

Cet état de fait implique que nous devons modifier la façon dont est calculée l'heuristique de convenance. En effet, bien que parfois plusieurs tests doivent être effectués pour discriminer la première clause, il n'en demeure pas moins qu'elle doive être discriminée en priorité. Il devient par conséquent moins utile d'accorder des points en faveur de la position de la clause, puisque les tests "à demi convenables" seraient alors retardés bien qu'ils doivent faire partie de tous les chemins.

Nous proposons donc de calculer l'heuristique de convenance en fonction du seul critère de convenance avec la première clause. De plus, nous modifierons ce critère pour inclure les tests retrouvés dans l'expression de test de la clause. La valeur de l'heuristique de convenance pour un test donnée sera donc de un si ce test convient à la première clause ou est présent dans son expression de test et de zéro sinon. De cette façon, seuls les tests pour lesquels la convenance avec la première clause est certaine seront avantagés. Puisque tous les tests inclus dans l'expression de test de la première clause sont maintenant considérés convenables, il devient clair que s'il n'existe aucun test convenable pour la première clause, c'est que celle-ci doit être retirée ou choisie. Dans les autres cas, l'ensemble des tests ayant eu la valeur un constitue le nouvel ensemble des candidats. Encore ici et pour toutes les heuristiques, si le nouvel ensemble de candidats est un singleton, le choix s'arrête sur ce dernier et sinon la prochaine heuristique est appliquée.

6.3.3 Heuristique du facteur de branchement (*branching factor*)

Baudinet et MacQueen définissent le facteur de branchement d'un test en fonction du sélecteur testé ou d'une de ses sous-expressions et fait appel à la notion de constructeur. Cette dernière est surtout utile dans un contexte impliquant un langage statiquement typé comme ML. L'équivalent Erlang des constructeurs est l'ensemble des littéraux, augmentés des constructeurs *cons* et *n-tuple*. qui représentent respectivement les constructeurs de paires et de tuples. On dira donc, par exemple, que les listes comportent deux constructeurs, soient *cons* et la liste vide.

Le facteur de branchement est donc présenté comme étant le nombre de constructeurs différents qui sont présents dans le patron à la position de la sous-expression testée. On choisira alors de tester la sous-expression qui a le facteur de branchement le moins élevé en espérant éliminer le plus de clauses avec un seul test. Les auteurs précisent que leur implantation du critère inclut de l'information qui tend a favoriser les tests pour lesquels il n'existe pas de patron général comme la variable ou le patron universel. Cependant, l'article référé n'a jamais été publié, ce qui laisse la définition de ce critère plutôt nébuleuse.

Nous avons tenté d'interpréter ce critère en observant l'exemple proposé par les auteurs. Il s'agit de la transformation `S` suivante:

```
S({true,green} -> (B1f),
  {false,green} -> (B2f),
  {_,_} -> (B3f))
```

Le facteur de branchement des tests effectuées sur le premier élément du 2-tuple est deux puisqu'il y a deux constructeurs présents. D'un autre coté, le facteur de branchement du deuxième élément du 2-tuple est un puisque seul le constructeur `green` y est présent. On décidera donc avec raison de tester le deuxième élément du 2-tuple en premier, ce qui permettra d'obtenir l'arbre de décision optimal de la figure 6.1.

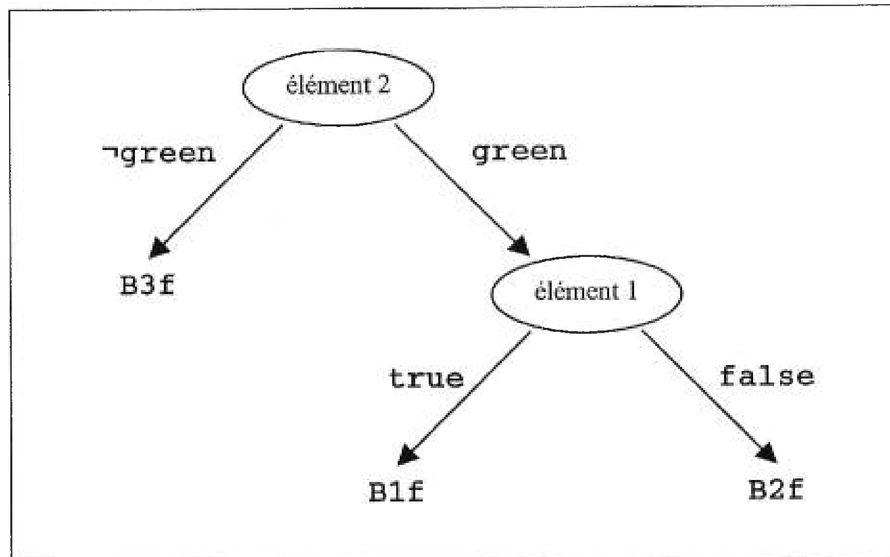

Figure 6.1: Arbre de décision optimal, langage statiquement typé

Ce problème de filtrage ML a une sémantique légèrement différente en Erlang. Le fait que ML soit statiquement typé implique que le premier élément du 2-tuple soit nécessairement un booléen, ce qui est différent de Erlang, où il est impossible de faire cette prédiction. On ne peut supposer que le premier élément du tuple soit `false` s'il n'est pas `true`. Une transformation `S` adaptée pour le langage Erlang et qui illustre exactement le même problème de filtrage est:

```
S({true,green} -> (B1f),
  {_,green} -> (B2f),
  {_,_} -> (B3f))
```

Le critère du facteur de branchement devient alors moins évident à évaluer. On ne peut se contenter ici de compter le nombre de constructeurs différents. Nous devons plutôt tenter de trouver un autre moyen d'estimer le facteur de branchement. La figure 6.2 montre les ensembles de clauses vivantes, c'est-à-dire celles qui peuvent encore être choisies, en fonction du choix du test et du résultat de celui-ci.

Test choisi	Clauses candidates en cas de succès	Clauses candidates en cas d'échec
`(erl-ato= <elt1> 'true)`	`B1f,B2f,B3f`	`B2f,B3f`
`(erl-ato= <elt2> 'green)`	`B1f,B2f,B3f`	`B3f`

Figure 6.2: Clauses candidates après le choix du test et son résultat

Il devient alors plus clair ici que le test sur le deuxième élément possède le moins de successeurs et constitue donc un choix avantageux. Il semble donc à ce point-ci que nous devions définir notre critère comme étant la somme du nombre de clauses candidates restantes en cas de succès et en cas d'échec. Bien que cette stratégie colle bien à la définition du facteur de branchement, il y a moyen de l'améliorer en accordant plus de valeur aux tests ayant du succès qu'à ceux qui n'en ont pas. Nous démontrerons notre point par l'exemple de la transformation S suivante:

```
S({a,a} -> (B1f),
  {b,a} -> (B2f),
  {c,b} -> (B3f),
  {_,_} -> (B4f))
```

La figure 6.3 montre les ensembles des clauses candidates en fonction des différents tests pouvant être choisis et les résultats de ces derniers. Nous pouvons y remarquer que la somme des clauses candidates est constante dans tous les cas: chaque test divise l'ensemble des clauses candidates en sous-ensembles de deux et trois clauses. Cependant, le test d'équivalence avec l'atome a du deuxième élément est le seul à s'attribuer plus de clauses en cas de succès.

Test choisi	Clauses candidates en cas de succès	Clauses candidates en cas d'échec
`(erl-ato= <elt1> 'a)`	B1f,B4f	B2f,B3f,B4f
`(erl-ato= <elt1> 'b)`	B2f,B4f	B1f,B3f,B4f
`(erl-ato= <elt1> 'a)`	B3f,B4f	B1f,B2f,B4f
`(erl-ato= <elt2> 'a)`	B1f,B2f,B4f	B3f,B4f
`(erl-ato= <elt2> 'b)`	B3f,B4f	B1f,B2f,B4f

Figure 6.3: Clauses candidates après le choix du test et son résultat

Puisque nous sommes dans le contexte d'un langage typé dynamiquement, l'information négative accumulée est très peu utile. Par exemple, le fait de savoir qu'un terme n'est pas l'atome `b` est beaucoup moins utile que de savoir qu'il est l'atome `c`. En effet, si le terme n'est pas l'atome `c`, il peut alors être l'atome `b` ou une donnée de tout autre type Erlang. Dans le second cas, le fait de savoir que le terme est l'atome `b` permet de prédire qu'il n'est pas l'atome `c` ni tout autre donnée de type ou de valeur différent. Nous pouvons donc supposer pour le cas de la transformation `S` présentée plus haut qu'il serait plus avantageux d'effectuer le test qui détermine si le deuxième élément est l'atome `a` puisqu'il contient moins de clauses candidates possédant l'information peu utile lorsque le test échoue. La répartition est alors meilleure et l'arbre de décision correspondant est optimal et mieux balancé (Figure 6.4) que celui correspondant au choix d'un test sur le premier élément (Figure 6.5).

Nous augmentons donc l'heuristique de facteur de branchement pour un test par le nombre de clauses candidates lorsque ce test échoue. Le critère se calcule donc maintenant comme étant la somme de la cardinalité de l'ensemble des clauses candidates si le test est un succès et du double de la cardinalité de l'ensemble des clauses candidates si le test est un échec. Les tests ayant la valeur la moins importante seront alors choisis comme faisant partie du nouvel ensemble de candidats pour l'application de la prochaine heuristique de sélection.

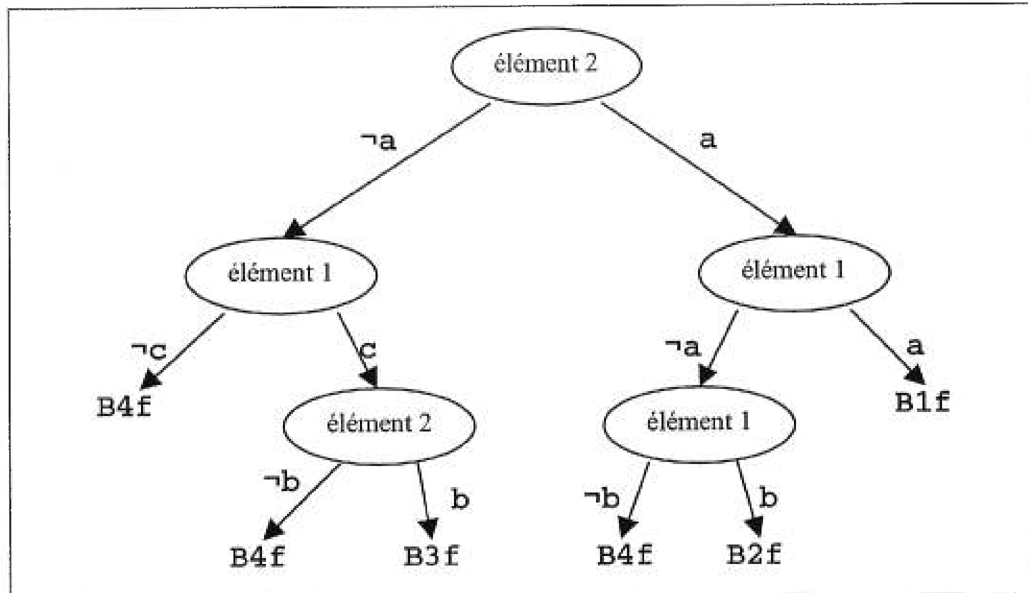

Figure 6.4: Arbre de décision optimal

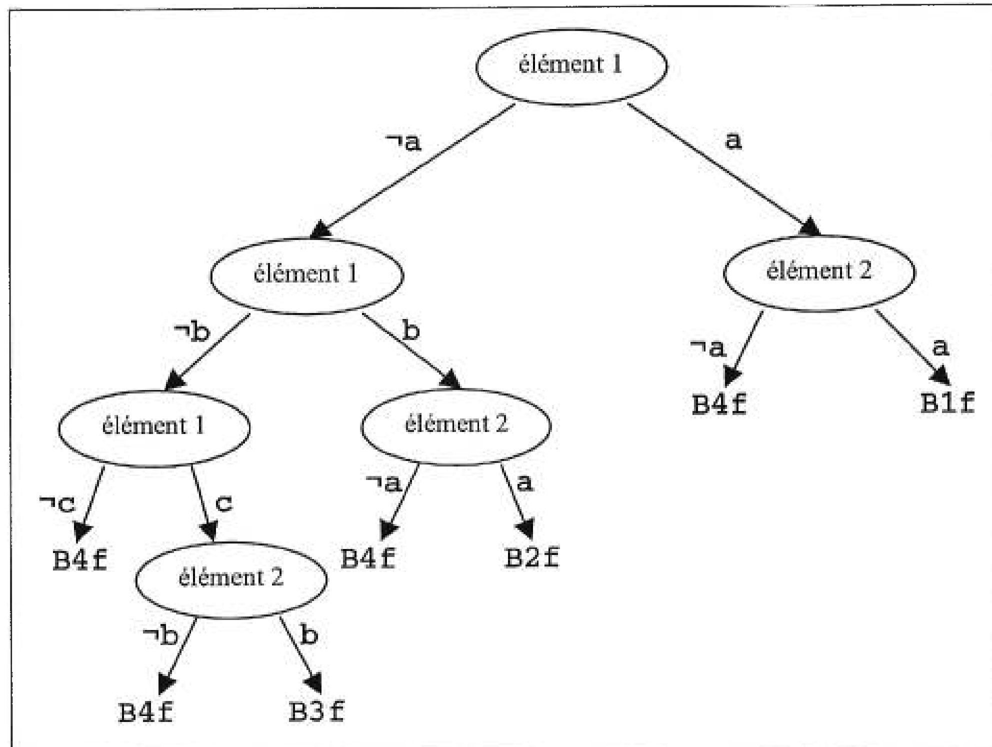

Figure 6.5: Arbre de décision non optimal

6.3.4 Heuristique du facteur d'arité (*arity factor*)

Baudinet et MacQueen définissent le facteur d'arité d'un test sur une sous-expression comme étant la somme des arités des constructeurs apparaissant au moins une fois dans les sous-expressions correspondantes. Sous Erlang, l'arité du constructeur *cons* est 2, celle du *n-tuple* est *n* et le reste des données sont formées avec des constructeurs d'arité 1. La pertinence du facteur d'arité est difficile à saisir et à notre avis, les auteurs ne la justifient pas clairement dans leur article. Bien qu'une donnée composée requière plus de tests afin d'être acceptée par un patron composé, chacun de ces tests est autant discriminatoire qu'un test sur un patron non composé. Par contre, il y a un avantage à choisir d'abord les tests sur les patrons non composés: l'accès aux sous-expressions est coûteuse puisqu'elle se traduit par des accès mémoire supplémentaires. Il y a donc un avantage à effectuer les tests sur les sous-expressions les plus simples d'abord. La complexité du test effectué sera donc tenue en compte dans notre prochaine heuristique. Nous avons donc laissé tomber le facteur d'arité dans notre implantation pour le remplacer par le facteur de complexité.

6.3.5 Heuristique du facteur de complexité

Une fois que la valeur de discrimination des tests a été évaluée par les deux premières heuristique, nous obtenons un ensemble de tests dont la complexité peut être très différente. Par exemple, soit la transformation `S` suivante:

```
S({[_|_],1} -> (B1f),
  {_,_} -> (B2f))
```

Les tests `(erl-cons? <elt1>)` et `(erl-fix= <elt2> 1)` passent tous deux les deux premières heuristiques mais ont des complexité différentes. En effet, la comparaison d'un entier à taille fixe est plus simple à effectuer que de déterminer si le premier élément est une paire, vue la structure des données de Gambit. Il devient alors ici plus avantageux de vérifier le deuxième élément d'abord.

D'un autre coté, examinons la transformation `S` suivante:

```
S({[1|_],1} -> (B1f),
  {[_|_],_} -> (B2f))
```

Le fait que le filtrage est exhaustif nous permet de supposer d'entrée de jeu que le premier élément est une paire. Nous nous retrouvons donc avec les tests (erl-fix= (erl-unsafe-hd/2 <*elt1*>) 1) et (erl-fix= <*elt2*> 1), qui ont aussi des complexités différentes. En effet, le premier test effectue un accès à la mémoire additionnel pour accéder au premier élément de la paire, ce qui le rend beaucoup plus lent que le deuxième. Il devient donc intéressant ici d'effectuer d'abord le deuxième test, qui discriminera plus rapidement la première clause.

Nous définissons donc le facteur de complexité en fonction du type de test effectué et de la complexité des opérandes de ce test. La complexité des types de tests est définie par la fonction CT avec les règles présentées à la figure 6.6. En général, un point est donné pour chaque opération arithmétique requise et cinq points pour chaque accès mémoire. Pour les tests plus complexes, une valeur forfaitaire de 100 points est accordée en raison d'un possible appel inter-module très coûteux. La complexité des opérandes des tests est définie pour sa part par la fonction CE avec les règles de la figure 6.7. Encore ici, les expressions plus complexes ont une valeur forfaitaire de 100 points.

```
CT((erl-fix= <term> k))    →  1+CE(<term>)
CT((erl-ato= <term> k))    →  1+CE(<term>)
CT((erl-chr= <term> k))    →  1+CE(<term>)
CT((erl-nil? <term>))      →  1+CE(<term>)
CT((erl-con? <term>))      →  2+CE(<term>)
CT((erl-fix? <term>))      →  1+CE(<term>)
CT((erl-sub? <term>))      →  2+CE(<term>)
CT((erl-spc? <term>))      →  2+CE(<term>)
CT((erl-vec? <term>))      →  6+CE(<term>)
CT((erl-flo? <term>))      →  7+CE(<term>)
CT((erl-big? <term>))      →  7+CE(<term>)
CT((erl-ato? <term>))      →  7+CE(<term>)
CT((erl-flo= <term>) k)    →  7+CE(<term>)
CT((erl-chr? <term>))      →  2+CE(<term>)
CT((<other> ...))          →  100
```

Figure 6.6: Règles de complexité des tests

```
CE(k)                                  →  0
CE(<Var>)                              →  4
CE((erl-unsafe-element/2 k <term>))    →  5+CE(<term>)
CE((erl-unsafe-hd/1 <term>))           →  5+CE(<term>)
CE((erl-unsafe-tl/1 <term>))           →  5+CE(<term>)
CE((<other> …))                        →  100
```

Figure 6.7: Règles de complexité des termes

Évidemment, l'ensemble des tests ayant le plus petit facteur de complexité est le nouvel ensemble de candidats. Puisqu'il s'agit de la dernière heuristique de sélection appliquée, le premier élément de l'ensemble des candidats est alors choisi arbitrairement comme étant le meilleur test à effectuer.

6.3.6 Le problème des listes

Notre algorithme de choix est maintenant bien défini. Il est à noter que nous avons jusqu'ici tenté de réduire le nombre de nœuds de l'arbre de décision. Cette stratégie semble être la plus adéquate puisque nous ne possédons aucune information statistique sur les chances qu'ont chaque clauses d'être choisies. Cependant, une exception persiste au niveau des listes. En effet, les listes sont très couramment utilisées dans les définitions de fonctions. Prenons par exemple la définition de la fonction `length`, qui calcule la longueur d'une liste. En Erlang, nous pouvons implanter cette fonction avec le code suivant:

```
length([]) -> 0;
length([_|X]) -> 1 + length(X).
```

Tel qu'il est conçu, notre algorithme de choix préconise le test `(erl-nil? sel)` comme premier test. Ceci est tout à fait justifié comme choix si l'on suppose qu'il y ait autant de chance que l'argument de la fonction soit la liste vide qu'une paire. De plus, dans un langage statiquement typé comme ML, ce choix serait d'autant plus approprié puisqu'un seul test ne serait alors requis. Avec Erlang, nous devons tout de même tester si l'argument est une paire après avoir prouvé qu'il n'était pas la liste vide puisqu'une exception peut alors être lancée. Dans les faits, la clause impliquant la liste vide ne sera choisie qu'une fois pour chaque liste de longueur *n*, alors que la clause impliquant la paire sera choisie *n* fois. Ce raisonnement découle d'une analyse du sens du programme qui serait difficile a faire de

façon automatique. Pourtant, le cas des fonctions définies de façon récursive sur les listes est tellement répandu qu'il vaut la peine d'être pris en considération.

Nous réalisons cette optimisation en désavantageant volontairement le test `erl-nil?`, puisqu'il est principalement utilisé dans ce type de construction. L'heuristique du facteur de complexité ajoutera donc 2 points à ce type de test afin que notre algorithme lui préfère l'utilisation du test `erl-con?` s'il est disponible. Cette simple modification rend plus efficace la plupart des fonctions définies de façon récursive sur les listes.

6.4 Résultats comparatifs

Cette section présente des résultats comparatifs des deux stratégies de choix présentées dans ce chapitre. Pour chacun des étalons de test, le temps d'exécution et la taille de l'exécutable (module compilé) sont fournis. Les tests ont été effectués sur un Pentium II 300MHz d'Intel, avec 128Mo de mémoire vive sous Linux. Les résultats sont illustrés sous forme de tableau à la figure 6.8.

	`epoptest.erl`		`litescm.erl`		`poker.erl`		`type.erl`	
	temps	taille	temps	taille	temps	taille	temps	taille
Première stratégie	15.93 s	10.4 Ko	10.44 s	48.1 Ko	13.30 s	51.6 Ko	13.80 s	10.9 Ko
Deuxième stratégie	14.93 s	10.0 Ko	9.59 s	47.3 Ko	12.93 s	43.8 Ko	12.56 s	10.9 Ko
Gains relatifs	**6.3%**	**4.0%**	**8.1%**	**1.7%**	**2.8%**	**15%**	**9.0%**	**0%**

Figure 6.8: Résultats comparatifs des deux stratégies

Les résultats obtenus montrent donc une amélioration significative des performances pour les étalons *epoptest*, *litescm* et *type*. Dans le cas de *epoptest*, les gains sont dus au fait que la deuxième stratégie décide de tester si l'on a une liste d'au moins quatre éléments avant de tester les éléments. Ceci permet de tester le quatrième élément dont le test d'équivalence avec le caractère $T a un facteur de branchement moins élevé en premier.

Dans le cas de *litescm*, les gains sont dus au fait que la compilation du filtrage de la fonction `expr_eval` décèle qu'il serait plus approprié de tester d'abord si l'argument est un tuple plutôt qu'un entier. Cette décision est déclenchée principalement grâce à l'heuristique du facteur de branchement. Pour *type*, c'est aussi cette même heuristique qui permet à notre stratégie d'optimiser la compilation du module. En effet, dans la fonction `type`, il est plus avantageux de vérifier d'abord si l'argument est de type sous-typé avec le test `(erl-sub? sel)`, plutôt que d'effectuer d'entrée de jeu le test `(erl-nil? sel)`, qui donne une information négative peu utile en cas d'échec et ce, pour le reste des clauses de la fonction. Il est aussi à noter l'optimisation du traitement des listes, qui est présente dans tous les étalons et qui permet de tester d'abord pour la paire avant la liste vide.

Des gains substantiels en terme d'espace peuvent être observés pour l'étalon *poker*. En effet, une réduction appréciable de 15% de la taille de l'exécutable est un atout non négligeable pour les applications s'exécutant dans un environnement où l'espace mémoire est limité. C'est la combinaison des heuristiques de convenance et du facteur de branchement qui est responsable de ces gains. La fonction `eval_h` du module *poker* permet de classifier une main de poker selon les règles du jeu. Évidemment, la majorité des cas ont des probabilités d'apparition plutôt basses, ce qui implique que de grandes parties de l'arbre de décision ne sont pratiquement jamais utilisées. Bien que les gains en terme de performance pour cet étalon soient négligeables, nous voyons qu'il est quand même profitable d'optimiser les parties peu utilisées de l'arbre de décision puisqu'il est alors possible de réduire significativement la taille de l'exécutable.

Conclusion

Les objectifs principaux des travaux rapportés dans ce mémoire portaient essentiellement sur la génération de programmes Scheme efficaces tant au plan spatial que temporel, à partir de programmes Erlang. Les résultats obtenus à ce jour démontrent que ces objectifs ont été partiellement atteints. En effet, il serait plutôt imprudent de parler d'accomplissement total dans un contexte d'optimisation de code, puisqu'il est en particulier très difficile et en général impossible d'établir une limite concrète à cette dernière.

Nous avons vu que la réduction du langage Erlang vers le langage intermédiaire UE permet une importante simplification du langage qui met en relief la principale différence syntaxique d'Erlang versus Scheme: le filtrage de patrons. L'explicitation des mécanismes de traitement d'erreur inhérents à Erlang a en outre permis des constructions plus efficaces, à l'aide de la forme syntaxique `ecase`, qui permet le filtrage de patrons sur un ensemble de clauses obligatoirement exhaustives.

Notre analyse du problème du filtrage a démontré qu'une compilation efficace du filtrage dépend d'une part de la capacité du compilateur à prédire l'issue de certains tests et d'autre part de la faculté de choisir l'ordre dans lequel ces derniers doivent être effectués. Nous avons montré comment les règles régissant les liens entre les différentes structures de données peuvent être explicitées au compilateur de façon intelligible avec l'utilisation des expressions de test prédictives. Nous avons aussi montré comment une adaptation de l'algorithme de Baudinet et MacQueen pouvait permettre d'effectuer un choix éclairé en ce qui concerne l'ordre des tests par l'utilisation d'heuristiques efficaces.

Les tests que nous avons effectués avec notre implantation de Etos sur les programmes étalons montrent une amélioration significative des temps d'exécution et de la taille des exécutables. Les gains en terme de temps d'exécution vont de 2.8% à 9.0% tandis

que la taille des exécutables est réduite dans des proportions variant entre 0% et 15% pour les programmes testés.

Plusieurs aspects de la compilation de Erlang vers Scheme présentés dans ce mémoire restent ouverts à la recherche. Par exemple, il serait intéressant de mesurer l'impact qu'aurait la conservation de l'information structurelle recueillie au cours du processus de filtrage afin d'optimiser l'évaluation des corps choisis pouvant éventuellement contenir d'autres expressions faisant appel au filtrage. De plus, la spécialisation des fonctions récursives par l'ajout de contraintes au niveau des types pourrait apporter des gains d'efficacité considérable en évitant les vérifications d'erreur inutiles.

En terminant, soulignons que l'ensemble des mécanismes de compilation présentés dans ce mémoire a été l'objet d'une implantation réelle au sein du projet Etos de l'Université de Montréal. Etos est un compilateur optimisant de Erlang vers Scheme écrit en Scheme pour le compilateur Gambit-C. Les sources du système peuvent être consultées sur le site Internet suivant: `http://www.iro.umontreal.ca/~etos`.

Remerciements

Je tiens à remercier mon directeur de recherche, Monsieur Marc Feeley, professeur au département d'informatique et de recherche opérationnelle de l'Université de Montréal, pour les conseils et critiques éclairés qu'il a su porter sur mes travaux.

Je tiens aussi à exprimer ma gratitude envers la compagnie de télécommunication Ericsson, pour son soutien financier tout au long de mes études sur le langage Erlang. Merci entre autres à Joe Armstrong et Bjarne Däcker pour m'avoir reçu si chaleureusement dans leurs laboratoires de Stockholm, lors de mes deux visites.

Je tiens finalement à remercier l'équipe travaillant au projet HiPE de l'Université d'Uppsala pour l'ensemble de leurs conseils, particulièrement ceux de Mikael Pettersson sur le plan du filtrage et de la réduction du langage Erlang.

Annexe 1

Grammaire du langage UE

```
<NamedDef>           ::=  <symbol> <- <AnonFunExpr>
<AnonFunExpr>        ::=  afun (<integer>,<symbol>) -> <EcaseExpr>
<Patterns>           ::=  <Pattern>
                          <Patterns> , <Pattern>
<Pattern>            ::=  <AtomicLiteral> |
                          <Variable> |
                          _ |
                          <TuplePattern> |
                          <ConsPattern>
<TuplePattern>       ::=  {}
                          {<Patterns>}
<ConsPattern>        ::=  [] |
                          [<Pattern>|<Pattern>]
<Patterns>           ::=  <Pattern> |
                          <Patterns> , <Pattern>
<Expr>               ::=  ecatch <Expr> |
                          <ApplicationExpr>
<Exprs>              ::=  <Expr> |
                          <Exprs> , <Expr>
<ApplicationExpr>    ::=  <AtomLiteral>(<Exprs>opt) |
                          <PrimaryExpr>
<PrimaryExpr>        ::=  <Variable> |
                          <AtomicLiteral> |
                          <EcaseExpr> |
                          <AnonFunExpr>
<AtomicLiteral>      ::=  <IntegerLiteral> |
                          <FloatLiteral> |
                          <CharLiteral> |
                          <StringLiteral> |
                          <AtomLiteral>
<EcaseExpr>          ::=  ecase <Expr> of <Clauses> end
<Clauses>            ::=  <Clause> ; <Clauses> |
                          <Clause>
<Clause>             ::=  <Pattern> when <Guards> -> <Expr>
<Guards>             ::=  <Guard> |
                          <Guards> | <Guard>
<Guard>              ::=  true |
                          <RecognizerBif>(<GuardExprs>opt) |
                          <ComparisonBif>(<GuardExpr>,<GuardExpr>)
<GuardExprs>         ::=  <GuardExpr> |
                          <GuardExprs> , <GuardExpr>
<GuardExpr>          ::=  <Variable> |
                          <AtomicLiteral> |
                          <GuardBif>(<GuardExprs>opt)
```

Annexe 2

Programmes étalons

epoptest.erl

```
% File: epoptest.erl
% Pattern matching benchmark
%
% This program tests the trans/3 function signature taken
% from epop.erl, a POP3 server written in Erlang
%
% Copyright 2000 Patrick Piche
%
-module(epoptest).
-export([start/0]).

trans(S,[$S,$T,$A,$T|_],Db) -> 1;
trans(S,[$L,$I,$S,$T|T],Db) -> 2;
trans(S,[$R,$E,$T,$R|T],Db) -> 3;
trans(S,[$D,$E,$L,$E|T],Db) -> 4;
trans(S,[$N,$O,$O,$P|_],Db) -> 5;
trans(S,[$R,$S,$E,$T|_],Db) -> 6;
trans(S,[$Q,$U,$I,$T|_],Db) -> 7;
trans(S,_,Db)               -> 8.

test_trans() ->
    trans(data,"STAT test",data),
    trans(data,"LIST test",data),
    trans(data,"RETR test",data),
    trans(data,"DELE test",data),
    trans(data,"NOOP test",data),
    trans(data,"QUIT test",data),
    trans(data,"TEST test",data).

call_trans(0) -> done;
call_trans(N) ->
    test_trans(),
    call_trans(N-1).

start() ->
    statistics(runtime),
    R = call_trans(10000000),
    {_,Time} = statistics(runtime),
    io:format("runtime = ~w msecs~nresult = ~w~n", [Time,R]).
```

litescm.erl

```
% File: litescm.erl
% Pattern matching benchmark -
%   Uses intensive pattern-matching for selection among many atoms
%   Also some selection among objects of different types
% Copyright 1999 Patrick Piche
-module(litescm).
-export([start/0]).

-ifdef(ETOS).
-define(IS_INTEGER,is_integer).
-define(STR_TO_ATOM,string_to_atom).
-else.
-define(IS_INTEGER,integer).
-define(STR_TO_ATOM,list_to_atom).
-endif.
% Lexical tokens:
% spaces                  : skip
% "("                     : lparen
% ")"                     : rparen
% [0-9]+                  : Val
% [any other symbol]+ : {symbol,Val}

tokenize([$\s|T]) -> tokenize(T);
tokenize([$\n|T]) -> tokenize(T);
tokenize([$(|T]) -> [lparen|tokenize(T)];
tokenize([$)|T]) -> [rparen|tokenize(T)];
tokenize([X|T]) when X>=$0,X=<$9 -> token_int([X|T],0);
tokenize([X|T]) -> token_sym(T,[X]);
tokenize([]) -> [].

token_int([$0|T],N) -> token_int(T,N*10);
token_int([$1|T],N) -> token_int(T,N*10+1);
token_int([$2|T],N) -> token_int(T,N*10+2);
token_int([$3|T],N) -> token_int(T,N*10+3);
token_int([$4|T],N) -> token_int(T,N*10+4);
token_int([$5|T],N) -> token_int(T,N*10+5);
token_int([$6|T],N) -> token_int(T,N*10+6);
token_int([$7|T],N) -> token_int(T,N*10+7);
token_int([$8|T],N) -> token_int(T,N*10+8);
token_int([$9|T],N) -> token_int(T,N*10+9);
token_int(R,N) -> [N|tokenize(R)].

token_sym([X|T],S) when X=/=$(,X=/=$),X=/=$\s,X=/=$\n -> token_sym(T,[X|S]);
token_sym(R,"tel") -> ['let'|tokenize(R)];
token_sym(R,"adbmal") -> ['lambda'|tokenize(R)];
token_sym(R,"fi") -> ['if'|tokenize(R)];
token_sym(R,"+") -> ['+'|tokenize(R)];
token_sym(R,"-") -> ['-'|tokenize(R)];
token_sym(R,"<") -> ['<'|tokenize(R)];
token_sym(R,"*") -> ['*'|tokenize(R)];
token_sym(R,"tneitouq") -> ['quotient'|tokenize(R)];
token_sym(R,S) -> [{var,?STR_TO_ATOM(reverse(S))}|tokenize(R)].

% Syntactic parser:

parse_expr([X|R]) when ?IS_INTEGER(X) -> {X,R};
parse_expr([{var,X}|R]) -> {{var,X},R};
parse_expr([lparen,'+'|R])  ->
```

```
    {Arg1,Rest1}=parse_expr(R),
    {Arg2,[rparen|Rest2]}=parse_expr(Rest1),
    {{plus,Arg1,Arg2},Rest2};
parse_expr([lparen,'-'|R]) ->
    {Arg1,Rest1}=parse_expr(R),
    {Arg2,[rparen|Rest2]}=parse_expr(Rest1),
    {{minus,Arg1,Arg2},Rest2};
parse_expr([lparen,'<'|R]) ->
    {Arg1,Rest1}=parse_expr(R),
    {Arg2,[rparen|Rest2]}=parse_expr(Rest1),
    {{lt,Arg1,Arg2},Rest2};
parse_expr([lparen,'*'|R]) ->
    {Arg1,Rest1}=parse_expr(R),
    {Arg2,[rparen|Rest2]}=parse_expr(Rest1),
    {{times,Arg1,Arg2},Rest2};
parse_expr([lparen,quotient|R]) ->
    {Arg1,Rest1}=parse_expr(R),
    {Arg2,[rparen|Rest2]}=parse_expr(Rest1),
    {{quotient,Arg1,Arg2},Rest2};
parse_expr([lparen,lambda,lparen|R]) ->
    {Params,Rest1}=parse_params(R),
    {Body,[rparen|Rest2]}=parse_expr(Rest1),
    {{lambda,Params,Body},Rest2};
parse_expr([lparen,'let',lparen|R]) ->
    {Formals,Actuals,Rest1}=parse_bnds(R),
    {Body,[rparen|Rest2]}=parse_expr(Rest1),
    {{apply,{lambda,Formals,Body},Actuals},Rest2};
parse_expr([lparen,'if'|R]) ->
    {Test,Rest1}=parse_expr(R),
    {Alt1,Rest2}=parse_expr(Rest1),
    {Alt2,[rparen|Rest3]}=parse_expr(Rest2),
    {{'if',Test,Alt1,Alt2},Rest3};
parse_expr([lparen|R]) ->
    {Func,Rest1}=parse_expr(R),
    {Args,Rest2}=parse_args(Rest1),
    {{apply,Func,Args},Rest2}.

% parse a parameter list, returns {[<parameters>],<rest>}
parse_params(X) -> parse_params(X,[]).
parse_params([{var,Sym}|R],L) -> parse_params(R,[Sym|L]);
parse_params([rparen|R],L) -> {L,R}.

% parse a parameter list, returns {[<parameters>],<rest>}
parse_args(X) -> parse_args(X,[]).
parse_args([rparen|R],L) -> {L,R};
parse_args(X,L) ->
    {Arg,Rest}=parse_expr(X),
    parse_args(Rest,[Arg|L]).

% parse a bindings list, returns {[<formals>],[<actuals>],<rest>}
parse_bnds(X) -> parse_bnds(X,[],[]).
parse_bnds([lparen,{var,Sym}|R],F,A) ->
    {Expr,[rparen|Rest]}=parse_expr(R),
    parse_bnds(Rest,[Sym|F],[Expr|A]);
parse_bnds([rparen|R],F,A) -> {F,A,R}.

parse(String) ->
    Toks=tokenize(String),
    {Expr,_}=parse_expr(Toks),
    Expr.

% Evaluation of AST
```

```
env_lookup(X,[{X,Y}|_]) -> Y;
env_lookup(X,[_|T]) -> env_lookup(X,T).

env_augment([],[],E) -> E;
env_augment([F|Fs],[A|As],E) -> env_augment(Fs,As,[{F,A}|E]).

eval_expr(K,E) when ?IS_INTEGER(K) -> K;
eval_expr({var,V},E) -> env_lookup(V,E);
eval_expr({plus,X,Y},E) -> eval_expr(X,E)+eval_expr(Y,E);
eval_expr({minus,X,Y},E) -> eval_expr(X,E)-eval_expr(Y,E);
eval_expr({times,X,Y},E) -> eval_expr(X,E)*eval_expr(Y,E);
eval_expr({quotient,X,Y},E) -> eval_expr(X,E) div eval_expr(Y,E);
eval_expr({lt,X,Y},E) ->
    case {eval_expr(X,E),eval_expr(Y,E)} of
      {A,B} when A<B -> 1;
        _ -> 0
    end;
eval_expr({'if',Test,Alt1,Alt2},E) ->
    case eval_expr(Test,E) of
      0 -> eval_expr(Alt2,E);
      _ -> eval_expr(Alt1,E)
    end;
eval_expr({lambda,Params,Body},E) ->
    fun(Args) -> eval_expr(Body,env_augment(Params,Args,E)) end;

eval_expr({apply,Func,Args},E) ->
    apply(eval_expr(Func,E),
        [map(fun(Arg) -> eval_expr(Arg,E) end,Args)]).

eval(String) ->
    Ast=parse(String),
    eval_expr(Ast,[]).

% Utils
reverse(X) -> reverse(X,[]).
reverse([],X) -> X;
reverse([H|T],X) -> reverse(T,[H|X]).

map(F,[]) -> [];
map(F,[H|T]) -> [F(H)|map(F,T)].

test_fib() ->
    eval(
"          (let ((fib"
"                 (let ((g (lambda (f)"
"                            (lambda (n) (if (< n 2)"
"                                            n"
"                                            (+ ((f f) (- n 1))"
"                                               ((f f) (- n 2))))))))"
"                   (g g))))"
"            (fib 6))"
        ).
loop(N) -> loop(N,0).
loop(0,R) -> R;
loop(N,R) -> loop(N-1,test_fib()).

% Entry program
start() ->
  statistics(runtime),
  R=loop(20000),
  {_,Time} = statistics(runtime),
  io:nl(),
```

```
io:format("runtime = "),
io:write(Time),
io:format(" msecs"),
io:nl(),
io:format("result = "),
io:write(R),
io:nl().
```

poker.erl

```
% File: poker.erl
% Pattern matching benchmark - Uses intensive Non-linear patterns
% Copyright 1999 Patrick Piche
-module(poker).
-export([start/0]).

% Evaluates a sorted poker hand
% Most of the execution time is spent here
h_eval({{10,C},{_,C},{_,C},{_,C},{_,C}}) -> royal_flush;
h_eval({{X,C},{_,C},{_,C},{_,C},{Y,C}}) when Y=:=X+4 -> straight_flush;
h_eval({{_,C},{_,C},{_,C},{5,C},{14,C}}) -> straight_flush;
h_eval({{X,_},{_,_},{_,_},{X,_},{_,_}}) -> four_of_a_kind;
h_eval({{_,_},{X,_},{_,_},{_,_},{X,_}}) -> four_of_a_kind;
h_eval({{X,_},{_,_},{X,_},{Y,_},{Y,_}}) -> full_house;
h_eval({{X,_},{X,_},{Y,_},{_,_},{Y,_}}) -> full_house;
h_eval({{_,C},{_,C},{_,C},{_,C},{_,C}}) -> flush;
h_eval({{X,_},{_,_},{X,_},{_,_},{_,_}}) -> three_of_a_kind;
h_eval({{_,_},{X,_},{_,_},{X,_},{_,_}}) -> three_of_a_kind;
h_eval({{_,_},{_,_},{X,_},{_,_},{X,_}}) -> three_of_a_kind;
h_eval({{X,_},{X,_},{Y,_},{Y,_},{_,_}}) -> two_pairs;
h_eval({{X,_},{X,_},{_,_},{Y,_},{Y,_}}) -> two_pairs;
h_eval({{_,_},{X,_},{X,_},{Y,_},{Y,_}}) -> two_pairs;
h_eval({{X,_},{X,_},{_,_},{_,_},{_,_}}) -> pair;
h_eval({{_,_},{X,_},{X,_},{_,_},{_,_}}) -> pair;
h_eval({{_,_},{_,_},{X,_},{X,_},{_,_}}) -> pair;
h_eval({{_,_},{_,_},{_,_},{X,_},{X,_}}) -> pair;
h_eval({{X,_},{_,_},{_,_},{_,_},{Y,_}}) when Y=:=X+4 -> straight;
h_eval({{_,_},{_,_},{_,_},{5,_},{14,_}}) -> straight;
h_eval({{_,_},{_,_},{_,_},{_,_},{_,_}}) -> low.

% Sorts a hand(bubble sort)
h_sort({{X1,C1},{X2,C2},C,D,E}) when X1>X2 -> h_sort({{X2,C2},{X1,C1},C,D,E});
h_sort({A,{X1,C1},{X2,C2},D,E}) when X1>X2 -> h_sort({A,{X2,C2},{X1,C1},D,E});
h_sort({A,B,{X1,C1},{X2,C2},E}) when X1>X2 -> h_sort({A,B,{X2,C2},{X1,C1},E});
h_sort({A,B,C,{X1,C1},{X2,C2}}) when X1>X2 -> h_sort({A,B,C,{X2,C2},{X1,C1}});
h_sort(X) -> X.

% Random number generator
random(Seed) -> (Seed*16807) rem 2147483647.

% Member function specialized on 2-tuples
card_member(_,[]) -> false;
card_member({X,C},[{X,C}|T]) -> true;
card_member(X,[_|T]) -> card_member(X,T).

% hand generator
generate_hand(Seed) -> generate_hand(Seed,5,[]).
generate_hand(Seed,0,L) -> {Seed,L};
generate_hand(Seed,N,L) ->
    R1=random(Seed),
    R2=random(R1),
    Card={(R1 rem 14)+1,R2 rem 4},
    case card_member(Card,L) of
      true -> generate_hand(R2,N,L);
      _ -> generate_hand(R2,N-1,[Card|L])
    end.

% generate a list of N "random" hands
```

```
generate_hands(N) -> generate_hands(1111,N,[]).
generate_hands(Seed,0,L) -> L;
generate_hands(Seed,N,L) ->
  {NewSeed,Hand}=generate_hand(Seed),
  generate_hands(NewSeed,N-1,[h_sort(list_to_tuple(Hand))|L]).

% evaluates a list of hands
eval_hands(L) -> eval_hands(L,[]).
eval_hands([],L) -> L;
eval_hands([H|T],L) -> eval_hands(T,[h_eval(H)|L]).

% summarize a list of evaluated hands
summarize(EvaluatedHands) -> summarize(EvaluatedHands,{0,0,0,0,0,0,0,0,0,0}).
summarize([],Acc) -> Acc;
summarize([Hand|T],{A,B,C,D,E,F,G,H,I,J}) ->
    summarize(T,
            case Hand of
              low -> {A,B,C,D,E,F,G,H,I,J+1};
              pair -> {A,B,C,D,E,F,G,H,I+1,J};
              two_pairs -> {A,B,C,D,E,F,G,H+1,I,J};
              three_of_a_kind -> {A,B,C,D,E,F,G+1,H,I,J};
              straight -> {A,B,C,D,E,F+1,G,H,I,J};
              flush -> {A,B,C,D,E+1,F,G,H,I,J};
              full_house -> {A,B,C,D+1,E,F,G,H,I,J};
              four_of_a_kind -> {A,B,C+1,D,E,F,G,H,I,J};
              straight_flush -> {A,B+1,C,D,E,F,G,H,I,J};
              royal_flush -> {A+1,B,C,D,E,F,G,H,I,J}
            end).

loop(N,Hands) -> loop(N,Hands,dummy).
loop(0,Hands,R) -> R;
loop(N,Hands,R) ->
    loop(N-1,Hands,eval_hands(Hands)).

% Entry program
start() ->
  Hands = generate_hands(5000),
  statistics(runtime),
  R = loop(2000,Hands),
  {_,Time} = statistics(runtime),
  io:nl(),
  io:format("runtime = "),
  io:write(Time),
  io:format(" msecs"),
  io:nl(),
  io:format("result = "),
  io:write(summarize(R)),
  io:nl().
io:nl().
```

type.erl

```
% File: type.erl
% Pattern matching benchmark - Uses intensive guard tests
% Copyright 1999 Patrick Piche
-module(type).
-export([start/0]).

-ifdef(ETOS).
-define(IS_INTEGER,is_integer).
-define(IS_ATOM,is_atom).
-define(IS_FLOAT,is_float).
-define(IS_TUPLE,is_tuple).
-define(IS_LIST,is_list).
-define(IS_BINARY,is_binary).
-define(IS_REF,is_ref).
-else.
-define(IS_CHAR,char).
-define(IS_INTEGER,integer).
-define(IS_ATOM,atom).
-define(IS_FLOAT,float).
-define(IS_TUPLE,tuple).
-define(IS_LIST,list).
-define(IS_BINARY,binary).
-define(IS_REF,ref).
-endif.

% Determine the type of an Erlang object (doesn't cover all possible types)
type([]) -> null;
type(X) when ?IS_INTEGER(X) -> integer;
type(X) when ?IS_BINARY(X) -> binary;
type(X) when ?IS_ATOM(X) -> atom;
type(X) when ?IS_FLOAT(X) -> float;
type(X) when ?IS_TUPLE(X) -> tuple;
type(X) when ?IS_REF(X) -> ref;
type([_|_]) -> cons.

typeall([],Last) -> Last;
typeall([Obj|R],Last) -> typeall(R,type(Obj)).

loop(N,Objs) -> loop(N,Objs,dummy).
loop(0,_,R) -> R;
loop(N,Objs,R) ->
  loop(N-1,Objs,typeall(Objs,dummy)).

% Entry program
start() ->
  Objs = [4453,1.3,a,{1,2,3},[1,2,3],[],make_ref(),
          list_to_binary([1,2,3]),123451234512345],
  statistics(runtime),
  R = loop(10000000,Objs),
  {_,Time} = statistics(runtime),
  io:nl(),
  io:format("runtime = "),
  io:write(Time),
  io:format(" msecs"),
  io:nl(),
  io:format("result = "),
  io:write(R),
  io:nl().
```

Annexe 3
Programme d'exemple

epop.erl

```
-module(epop).
-author('tobbe@serc.rmit.edu.au').
%%-----------------------------------------------------------------------
%% File    : epop.erl
%% Created : 3 Mar 1998 by tobbe@serc.rmit.edu.au
%% Function: A POP3 server/client according to RFC-1939
%%-----------------------------------------------------------------------
-vc('$Id: epop.erl,v 1.8 1998/03/12 20:01:07 tobbe Exp $ ').
-export([start/0,start/1,start/2,stop/0]).
-export([connect/2,connect/3,stat/1,scan/1,scan/2,retrieve/2,delete/2,
     reset/1,quit/1,store_mail/3]).
-export([sys_call/1,lcase/1]).
-export([init/3,listener/2,worker/3,funcall/2]).

-import(epop_db, [db/0,db_in/3,db_out/2,db_replace/3,db_filter/2,
                  db_foreach/2,db_member/2,db_member/3]).

-define(SERVER_NAME, epop_server).
-define(DEFAULT_PORT,  110).
-define(DEFAULT_DBMOD, epop_dets).  % Using the dets module

%% -----------------------
%% Exported client commands
%% -----------------------

connect(User,Passwd)       -> epop_client:connect(User,Passwd).
connect(User,Passwd,Opts)  -> epop_client:connect(User,Passwd,Opts).
stat(S)                    -> epop_client:stat(S).
scan(S)                    -> epop_client:scan(S).
scan(S,MsgNum)             -> epop_client:scan(S,MsgNum).
retrieve(S,MsgNum)         -> epop_client:retrieve(S,MsgNum).
delete(S,MsgNum)           -> epop_client:delete(S,MsgNum).
reset(S)                   -> epop_client:reset(S).
quit(S)                    -> epop_client:quit(S).

%% ----------------------------
%% Exported database command
%% (To be used by a SMTP server)
%% ----------------------------

store_mail(User,MsgID,Mail) -> epop_dets:store_mail(User,MsgID,Mail).

%% -----------------------
%% Exported server commands
%% -----------------------

start() -> start(?DEFAULT_PORT).

start(Lport) -> start(Lport,?DEFAULT_DBMOD).

start(Lport,DbMod) when integer(Lport),atom(DbMod) ->
    case whereis(?SERVER_NAME) of
      Pid when pid(Pid) -> {ok,Pid};
      _ ->
      Pid = spawn(?MODULE, init, [self(),Lport,DbMod]),
      register(?SERVER_NAME,Pid),
      receive {Pid,Msg} -> Msg after 5000 -> {error,timeout} end
    end.
```

```
stop() ->
    ?SERVER_NAME ! stop,
    ok.

%% Called from the epop_sys module

sys_call(F) ->
    ?SERVER_NAME ! {sys_call,F}.


%% ----------------
%% The Listen server
%% ----------------

init(From,Lport,DbMod) ->
    process_flag(trap_exit,true),
    put(dbmod,DbMod),
    case apply(DbMod,start,[]) of
      {ok,_} ->
          Lsock = setup_listen_sock(Lport),
          Accept = do_accept(Lsock,DbMod),
          From ! {self(),ok},
          listener(Lsock,Accept);
      Else ->
          error_logger:error_msg("epop: Could not start the database !~n"),
          From ! {self(),Else},
          exit(Else)
    end.

setup_listen_sock(Lport) ->
    Opts =  [{packet,raw},{reuseaddr,true},{active,false}],
    case catch gen_tcp:listen(Lport,Opts) of
      {ok, Lsock} -> Lsock;
      Else        -> exit(Else)
    end.

listener(Lsock,Accept) ->
    receive
      {Accept,connected} ->
          ?MODULE:listener(Lsock,do_accept(Lsock,get(dbmod)));
      {sys_call,F} ->
          spawn(?MODULE,funcall,[F,get(dbmod)]),
          ?MODULE:listener(Lsock,Accept);
      {'EXIT',Accept,_} ->
          ?MODULE:listener(Lsock,do_accept(Lsock,get(dbmod)));
      stop ->
          exit(normal);
      _ ->
          ?MODULE:listener(Lsock,Accept)
    end.

funcall(F,DbMod) -> F(DbMod).

do_accept(Lsock,DbMod) ->
    spawn_link(?MODULE,worker,[self(),Lsock,DbMod]).


%% -----------------
%% The Accept process
%% -----------------
```

```
worker(Listener,Lsock,DbMod) ->
    put(dbmod,DbMod),
    {ok,S} = gen_tcp:accept(Lsock),
    Listener ! {self(),connected},
    send_msg(S,ready),
    authorize(S).

%% ------------------------------
%% The Quit state (not in the RFC)
%% ------------------------------

quit(S,Db) ->
    {ok,User} = db_out(Db,user),
    send_msg(S,{signoff,User}),
    unlock_maildrop(User),
    exit(normal).


%% ------------------
%% The Authorize state
%% ------------------

%% ------------------------------------------------------------
%% The session database consists of a collection
%% of {Key,Value} tuples. Valid keys so far is:
%%
%%   {user,UserName}       -  The name of the user
%%   {passwd,PassWord}     -  The user password
%%   {{deleted,N},true}    -  A delete-message mark
%%   {scan_list,ScanList}  -  A scan list, where ScanList is a
%%                            sorted list of {MsgID,Size} tuples

authorize(S) ->
    goto_auth(S,db()).

goto_auth(S,Db) ->
    auth(S,recv(S),Db).

auth(S,[$U,$S,$E,$R|T],Db) -> % USER
    auth_name(S,T,Db);
auth(S,[$Q,$U,$I,$T|T],Db) -> % QUIT
    quit(S,Db);
auth(S,_,Db)               -> % Unexpected
    ok.

auth_name(S,Sname,Db) ->
    case regexp:match(Sname,"[^ \\r\\n]+") of
      {match,Start,Length} ->
          Name = string:substr(Sname,Start,Length),
          is_user_ok(S,Name,Db);
      _ ->
          send_msg(S,{userwrong,Sname}),
          goto_auth(S,Db)
    end.

is_user_ok(S,Name,Db) ->
    case get_passwd(Name) of
      {ok,Passwd} ->
          send_msg(S,{userok,Name}),
          NewDb = db_in(db_in(Db,user,Name),passwd,Passwd),
          auth_pass(S,recv(S),NewDb);
      _ ->
```

```
            send_msg(S,{userwrong,Name}),
            goto_auth(S,Db)
    end.

auth_pass(S,[$P,$A,$S,$S,$ |T],Db) ->
    %% NB: Spaces are valid in the password string
    case regexp:match(T,"[^\\r\\n]+") of
      {match,Start,Length} ->
            Passwd = string:substr(T,Start,Length),
            is_passwd_ok(S,Passwd,Db);
      _ ->
            send_msg(S,passwdwrong),
            goto_auth(S,Db)
    end.

is_passwd_ok(S,Passwd,Db) ->
    case db_member(Db,passwd,Passwd) of
      true ->
            set_lock(S,Db);
      _ ->
            send_msg(S,passwdwrong),
            goto_auth(S,Db)
    end.

set_lock(S,Db) ->
    {ok,User} = db_out(Db,user),
    case lock_maildrop(User) of
      ok ->
            send_msg(S,maildroplocked),
            transaction(S,Db);
      _ ->
            send_msg(S,lockfailed),
            goto_auth(S,Db)
    end.


%% ---------------------
%% The transaction state
%% ---------------------

transaction(S,Db) ->
    goto_trans(S,Db).

goto_trans(S,Db) ->
    trans(S,recv(S),Db).

trans(S,[$S,$T,$A,$T|_],Db) -> % STAT
    stat_reply(S,Db);
trans(S,[$L,$I,$S,$T|T],Db) -> % LIST
    list_reply(S,T,Db);
trans(S,[$R,$E,$T,$R|T],Db) -> % RETR
    retr_reply(S,T,Db);
trans(S,[$D,$E,$L,$E|T],Db) -> % DELE
    dele_reply(S,T,Db);
trans(S,[$N,$O,$O,$P|_],Db) -> % NOOP
    noop_reply(S,Db);
trans(S,[$R,$S,$E,$T|_],Db) -> % RSET
    rset_reply(S,Db);
trans(S,[$Q,$U,$I,$T|_],Db) -> % QUIT
    update(S,Db);
trans(S,_,Db)               -> % Unexpected
    unexpected_reply(S,Db).
```

```
%% ----------------------
%% Reply to a STAT request

stat_reply(S,Db) ->
    {ok,User} = db_out(Db,user),
    {ok,{N,M}}  = do_stat(User),
    send_msg(S,{statreply,N,M}),
    goto_trans(S,Db).

%% ----------------------
%% Reply to a LIST request

list_reply(S,Smsg,Db) ->
    NoArg = regexp:match(Smsg,"[ ]*\r\n"),
    Arg   = regexp:match(Smsg,"[ ]*[0-9]+[ ]*\r\n"),
    do_list_reply(S,Smsg,Db,NoArg,Arg).

%% No argument given !
do_list_reply(S,Smsg,Db,{match,1,_},_) ->
    NewDb = db_scan(Db),
    case scan_list(NewDb) of
      {ok,[]} ->
          send_msg(S,listnomsg);
      {ok,ScanList} ->
          send_msg(S,listbegin),
          send_scan_list(S,Db,ScanList),
          send_msg(S,terminateoctet)
    end,
    goto_trans(S,NewDb);
%% A numeric argument given !
do_list_reply(S,Smsg,Db,_,{match,1,Len}) when Len==length(Smsg) ->
    {match,Start,Length} = regexp:first_match(Smsg,"[0-9]+"),
    Num = l2i(string:substr(Smsg,Start,Length)),
    NewDb = db_scan(Db),
    case db_member(NewDb,{deleted,Num},true) of
      true  -> send_msg(S,nosuchmsg);
      false ->
          case scan_list(NewDb,Num) of
            {ok,[_MsgID,Size]} -> send_msg(S,{listmulti,Num,Size});
            _ -> send_msg(S,nosuchmsg)
          end
    end,
    goto_trans(S,NewDb);
%% A fulty argument given !
do_list_reply(S,_,Db,_,_) ->
    send_msg(S,nosuchmsg),
    goto_trans(S,Db).

%% ----------------------------------------
%% Number the scan list before sending it and
%% do not send messages marked for deletion.

send_scan_list(S,Db,ScanList) ->
    send_scan_list(S,Db,ScanList,1).

send_scan_list(S,Db,[[_MsgID,Size]|T],N) ->
    case db_member(Db,{deleted,N},true) of
      true  -> false;
      false -> send_msg(S,{listmulti,N,Size})
    end,
    send_scan_list(S,Db,T,N+1);
```

```
send_scan_list(_,_,[],_) ->
    true.

scan_list(Db) ->
    db_out(Db,scan_list).

scan_list(Db,N) ->
    case db_out(Db,scan_list) of
      {ok,ScanList} when length(ScanList)<N ->
          {error,nosuchmsg};
      {ok,ScanList} when length(ScanList)>=N ->
          {ok,lists:nth(N,ScanList)};
      _ ->
          {error,nosuchmsg}
    end.


%% ----------------------
%% Reply to a RETR request

retr_reply(S,Smsg,Db) ->
    do_retr_reply(S,Smsg,Db,regexp:match(Smsg,"[ ]*[0-9]+[ ]*\r\n")).

%% A numeric argument given !
do_retr_reply(S,Smsg,Db,{match,1,Len}) when Len==length(Smsg) ->
    {match,Start,Length} = regexp:first_match(Smsg,"[0-9]+"),
    Num = l2i(string:substr(Smsg,Start,Length)),
    case get_mail(Num,Db) of
      {ok,{Mail,Size}} ->
          send_msg(S,{retrreply,Size}),
          send_msg(S,{retrmail,Mail}),
          send_msg(S,terminateoctet),
          goto_trans(S,Db);
      _ ->
          send_msg(S,nosuchmsg),
          goto_trans(S,Db)
    end;
%% ERROR: No argument given !!
do_retr_reply(S,_,Db,_) ->
    send_msg(S,nosuchmsg),
    goto_trans(S,Db).

get_mail(Num,Db) ->
    case scan_list(Db,Num) of
      {ok,[MsgID,Size]} ->
          {ok,User} = db_out(Db,user),
          get_the_mail(User,MsgID);
      _ ->
          {error,nosuchmsg}
    end.


%% ----------------------
%% Reply to a DELE request

dele_reply(S,Smsg,Db) ->
    do_dele_reply(S,Smsg,Db,regexp:match(Smsg,"[ ]*[0-9]+[ ]*\r\n")).

%% A numeric argument given !
do_dele_reply(S,Smsg,Db,{match,1,Len}) when Len==length(Smsg) ->
    {match,Start,Length} = regexp:first_match(Smsg,"[0-9]+"),
```

```
    Num = l2i(string:substr(Smsg,Start,Length)),
    NewDb = mark_as_deleted(S,Num,Db),
    goto_trans(S,NewDb);
do_dele_reply(S,Smsg,Db,_) ->
    send_msg(S,nosuchmsg),
    goto_trans(S,Db).

mark_as_deleted(S,N,Db) ->
    case ok_to_delete(S,N,Db) of
      true  ->
          NewDb = do_delete(N,Db),
          send_msg(S,{deleted,N}),
          NewDb;
      false ->
          Db
    end.

do_delete(N,Db) ->
    db_in(Db,{deleted,N},true).

ok_to_delete(S,N,Db) ->
    Already = already_deleted(N,Db),
    Valid   = valid_mailno(N,Db),
    if
      Already==true ->
          send_msg(S,{alreadydeleted,N}),
          false;
      Valid==false  ->
          send_msg(S,nosuchmsg),
          false;
      true ->
          true
    end.

already_deleted(N,Db) ->
    case db_out(Db,{deleted,N}) of
      {ok,true} -> true;
      _         -> false
    end.

valid_mailno(N,Db) ->
    case scan_list(Db) of
      {ok,ScanList} when N=<length(ScanList) -> true;
      _ -> false
    end.


%% ------------------------
%% Reply to a NOOP request

noop_reply(S,Db) ->
    send_msg(S,ok),
    goto_trans(S,Db).



%% ------------------------
%% Reply to a RSET request

rset_reply(S,Db) ->
    NewDb = rm_delete_mark(Db),
    send_msg(S,ok),
```

```
    goto_trans(S,NewDb).

rm_delete_mark(Db) ->
    F = fun({{deleted,_},_}) -> false;
          (_) -> true
      end,
    db_filter(Db,F).


%% ----------------------------
%% Reply to a Unexpected request

unexpected_reply(S,Db) ->
    send_msg(S,unexpected),
    goto_trans(S,Db).


%% ---------------
%% The update state
%% ---------------

update(S,Db) ->
    NewDb = delete_marked_msgs(Db),
    quit(S,NewDb).



%% ---------------
%% Send and Receive
%% ---------------

send_msg(S,Msg) ->
    gen_tcp:send(S,msg(Msg)).

recv(S) ->
    %% receive {tcp,S,Data} -> Data end.
    case gen_tcp:recv(S,0) of
      {ok,Packet} -> Packet;
      Else        -> exit(Else)
    end.

%% --------
%% Messages
%% --------

msg(ready) ->
    "+OK POP3 server ready\r\n";
msg({userok,Name}) ->
    "+OK " ++ Name ++ " is a valid mailbox\r\n";
msg({userwrong,Name}) ->
    "-ERR sorry, no mailbox for " ++ Name ++ " here\r\n";
msg(passwdwrong) ->
    "-ERR invalid password\r\n";
msg(lockfail) ->
    "-ERR unable to lock maildrop\r\n";
msg(maildroplocked) ->
    "+OK maildrop locked and ready\r\n";
msg({statreply,N,M}) ->
    "+OK " ++ i2l(N) ++ " " ++ i2l(M) ++ "\r\n";
msg({listreply,N,M}) ->
    "+OK " ++ i2l(N) ++ " " ++ i2l(M) ++ "\r\n";
msg(nosuchmsg) ->
```

```
    "-ERR no such message\r\n";
msg(listbegin) ->
    "+OK scan listing follows\r\n";
msg(listnomsg) ->
    "+OK\r\n.\r\n";
msg({listmulti,N,M}) ->
    i2l(N) ++ " " ++ i2l(M) ++ "\r\n";
msg(terminateoctet) ->
    ".\r\n";
msg({retrreply,Size}) ->
    "+OK " ++ i2l(Size) ++ " octets\r\n";
msg({retrmail,Mail}) ->
    Mail ++ "\r\n";
msg({deleted,Num}) ->
    "+OK message " ++ i2l(Num) ++ " deleted\r\n";
msg({alreadydeleted,N}) ->
    "-ERR message " ++ i2l(N) ++ " already deleted\r\n";
msg(ok) ->
    "+OK\r\n";
msg(unexpected) ->
    "-ERR que ?\r\n";
msg({signoff,Name}) ->
    "+OK " ++ Name ++ " POP3 server signing off\r\n";
msg({lockfailed}) ->
    "-ERR unable to lock maildrop\r\n".


%% ---------------------------------------
%% Interface towards the Epop Mail Database
%% ---------------------------------------

get_passwd(User) ->
    apply(get(dbmod),get_passwd, [User]).

do_stat(User) ->
    apply(get(dbmod),stat, [User]).

lock_maildrop(User) ->
    apply(get(dbmod),lock_maildrop, [User]).

unlock_maildrop(User) ->
    apply(get(dbmod),unlock_maildrop, [User]).

%% ---------------------------------------
%% We store the scan list locally for this
%% session. This is ok since we have locked
%% the maildrop in the authentication state.

db_scan(Db) ->
    {ok,User} = db_out(Db,user),
    {ok,ScanList} = apply(get(dbmod),scan_list, [User]),
    case db_member(Db,scan_list) of
      true  -> db_replace(Db,scan_list,ScanList);
      false -> db_in(Db,scan_list,ScanList)
    end.

get_the_mail(User,MsgID) ->
    case apply(get(dbmod),get_mail, [User,MsgID]) of
      {ok,{Mail,Size}} -> {ok,{byte_stuff_it(Mail),Size}};
      _                -> {error,nosuchmsg}
    end.
```

```
%% NB: Should do byte stuffing here !
byte_stuff_it(Mail) ->
    Mail.


delete_marked_msgs(Db) ->
    {ok,User} = db_out(Db,user),
    F = fun({{deleted,N},_}) ->
            case scan_list(Db,N) of
                {ok,[MsgID,Size]} ->
                  apply(get(dbmod),delete_mail,[User,MsgID,Size]);
                _ ->
                  true % ignore !
            end;
         (_) -> true
      end,
    %% Delete all mails marked for deletion
    db_foreach(Db,F),
    %% Remove all the delete marks
    rm_delete_mark(Db).



%% ----------
%% Misc stuff
%% ----------

i2l(I) when integer(I) -> integer_to_list(I).

l2i(L) when list(L) -> list_to_integer(L).

lcase([C|Cs]) when C>=$A,C=<$Z -> [C+32|lcase(Cs)]; % A-Z
lcase([C|Cs])                  -> [C|lcase(Cs)];
lcase([])                      -> [].
```

Références

[AAB91] H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele Jr., G. J. Sussman, M. Wand. *Revised⁴ Report on the Algorithmic Language Scheme*. William Clinger and Jonathan Rees, November 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles Techniques, and Tools*. Addison-Wesley, 1986.

[AVW96] Joe Armstrong, Robert Virding, Cläes Wikström. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

[BOI88] P. Boizumault. *Prolog: l'implantation*. Masson, Paris, 1988.

[BM85] Marianne Baudinet, David MacQueen. *Tree Pattern Matching for ML (extended abstract)*. Université Stanford et Laboratoires AT&T Bell, décembre 1985.

[CJL99] Richard Carlsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, Robert Virding. *Core Erlang language specification, draft 0.99ß*. Novembre 1999.

[COL82] A. Colmerauer. *Prolog II, manuel de référence et modèle théorique*. G.I.A, Fac. Des Sciences de Luminy, Marseille, 1982.

[DW86] Saumya K. Debray, David S. Warren. *Detection and Optimization of Functional Computations in Prolog*. Department of Computer Science, Université de New York, Stony Brook, 1986.

[GC98] Marc Feeley. Gambit-C version 3.0, user manual. Compiler available at `http://www.iro.umontreal.ca/~gambit`.

[FL97] Marc Feeley, Martin Larose. *Etos: an Erlang to Scheme compiler.* Université de Montréal, 1997.

[HUG90] John Hughes. *Why Functional Programming Matters*. Chalmers University of Technology, 1990.

[ISR99] The Internet Scheme Repository: `http://www.cs.indiana.edu/scheme-repository`.

[JNP99] Erik Johansson, Sven-Olof Nyström, Mikael Pettersson, Konstantinos Sagonas. *HiPE: High Perfomance Erlang*. Université d'Uppsala rapport technique ASTEC 99/04, octobre 1999.

[PJ86] Simon L. Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, Hertfordshire, 1986.

[MP99] Mikael Pettersson. *Compiling Natural Semantics*. Springer, 1999.

[QUE90] Christian Queinnec. *Le filtrage: Une application de (et pour) LISP.* InterEditions, Paris, 1990.

[QG92] Christian Queinnec, Jean-Marie Geffroy. *Partial Evaluation applied to Symbolic Pattern Matching with Intelligent Backtrack*. École Polytechnique et INRIA-Rocquencourt, octobre 1992.

[SES96] Peter Sestoft. *ML pattern match compilation and partial evaluation*. Royal Veterinary and Agricultural University, Denmark, 1996.

[SET89] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

[SHA88] E. Shapiro. *Concurrent Prolog*. Vol 1 et 2, MIT Press, 1988.