Langages Applicatifs et Machines Abstraites pour la Couverture de Code Structurelle

Philippe Wang

UNIVERSITÉ PIERRE ET MARIE CURIE – PARIS 6
ÉCOLE DOCTORALE EDITE – ED130

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université Pierre et Marie Curie – Paris 6
Mention Informatique

Philippe Wang

Langages Applicatifs et Machines Abstraites pour la Couverture de Code Structurelle

Thèse dirigée par Emmanuel Chailloux,

préparée au Laboratoire d'Informatique de Paris 6 (UPMC),

et soutenue publiquement le jeudi 4 octobre 2012

devant le **Jury** composé de

Président	Pr. Bertil Folliot	–	Université Pierre et Marie Curie
Rapporteurs	Pr. Roberto Di Cosmo	–	Université Paris Diderot
	Dr. Virginie Wiels	–	Onera Toulouse
Examinateurs	Pr. Pascale Le Gall	–	Université d'Evry-Val d'Essonne
	Dr. Philippe Narbel	–	Université Bordeaux 1
	Dr. Bruno Pagano	–	Esterel Technologies
	Pr. Edmond Schonberg	–	New York University
Directeur	Pr. Emmanuel Chailloux	–	Université Pierre et Marie Curie

Les principaux outils utilisés pour produire ce document sont les suivants :

- pdfLaTeX 2_ε 3.1415926-1.40.11 (TeX Live 2010),
- OCaml 3.12,
- shell script (GNU Bash 3 + GNU Utils),
- GNU Make 3.8,
- GNU Emacs 22.

Les figures utilisant des couleurs sont les suivantes :

- figure VII.2, page 127 *
- figure VII.4, page 130
- figure VII.5, page 131
- figure VII.6, page 132
- figure VII.7, page 135
- figure VII.8, page 136
- figure VII.10, page 138 *
- figure VII.11, page 139
- figure VII.12, page 139
- figure VII.13, page 140
- figure VII.14, page 141

* : peut être converti en niveaux de gris sans perte d'informations.

2012/10/13 17:15:02 UTC - 1013-1681966-+-43 267-+-95-190-204-216-171-115-122-79-+-138-+-7-29-68

Thèse de doctorat préparée au Laboratoire d'Informatique de Paris 6 (LIP6)
au sein de l'équipe Algorithmes, Programmes and Résolution (APR)
située au troisième étage de la barre 26-00 du campus Jussieu ;
4, place Jussieu, 75005 Paris, France.

Résumé

« Langages Applicatifs et Machines Abstraites pour la Couverture de Code Structurelle »

« Cette thèse présente une étude qui répond à un besoin industriel d'avoir des outils pour aider à la qualité et au respect des processus de développement de logiciels critiques comme ceux du domaine de l'avionique civile. Il s'agit de l'étude de la couverture de code structurelle pour un langage de la famille ML. Dans ce contexte, ML apparaît comme un langage particulièrement riche en constructions de haut-niveau d'abstraction et expressif. Son utilisation est un élément de progrès mais soulève des problèmes d'adaptation des pratiques du génie logiciel classique pour les systèmes critiques. Notamment, la notion de couverture des conditions et des décisions ainsi que les critères de couverture dérivés se complexifient rapidement. Nous donnons alors en première contribution plusieurs sémantiques pour l'interprétation des définitions des conditions et des décisions pour un langage d'expressions de haut-niveau que nous avons complètement formellement défini. Ensuite, nous donnons la sémantique formelle pour une implantation pour la mesure de couverture par réécriture du code source, ce que nous appelons l'instrumentation intrusive. Puis, nous étudions une technique qui ne réécrit pas le code, ce qui permet d'avoir la possibilité d'utiliser le même binaire pour les tests et pour la production. Cette technique, que nous appelons non intrusive, consiste à générer les informations de correspondance entre le code source et le code machine, et éventuellement d'autres informations, pour que l'environnement d'exécution incluant une machine virtuelle puisse enregistrer les traces nécessaires à l'élaboration des rapports de couverture. Enfin, nous comparons ces deux approches, en terme de sémantique, d'utilisation et d'implantation. »

Mots clefs : couverture structurelle de code, MC/DC, instrumentation de code, langage de haut-niveau d'abstraction, outils pour logiciels critiques

Abstract

« Applicative Languages and Abstract Machines for Structural Code Coverage »

« This thesis presents a study on structural code coverage for a language of the ML family, in response to an industrial need in safety-critical software domain to develop tools. In this context, ML appears as a particularly rich and high-level language with a high degree of expressiveness. Its use is a progress but also raises issues when trying to apply classical safety-critical software engineering processes. In particular, the two notions of condition and decision, as well as coverage criteria associated with them, rapidly become very complex. The first contribution of this thesis answers the question of what conditions and decisions mean for a language of the ML family, by giving several formal definitions. Then, we present a formalised technique for structural code coverage which rewrites the source code to produce traces at run-time. We name it the intrusive instrumentation. We also formalise another technique which does not rewrite the source code, which allows to use the same binary for both testing activities and production. This second technique is called non intrusive and consists in generating at compile-time the information needed to match the machine code back to the source code. Other information are also generated for the execution environment to record specific traces that we need to generate a coverage report involving Boolean measures. Finally, we compare these two techniques both formally and practically, but also in terms of implementation. »

Keywords : structural code coverage, MC/DC, code instrumentation, high-level language, safety-critical software tools.

Remerciements

Après ces quelques années très riches en découvertes, en rencontres, en surprises, en apprentissage, en voyages, en tout en fait, c'est finalement avec un assez grand soulagement que je rédige *enfin* ces quelques mots de remerciements. Je vais pouvoir passer à la suite.

J'ai fait la connaissance d'Emmanuel il y a sept années déjà. Le premier de ses cours auquel j'ai assisté parlait de programmation par objets avec OCaml en comparaison avec le modèle à objets de Java sur lequel on avait passé le reste du semestre. À la sortie de ce cours, je démarrais une aventure impliquant de regarder le fonctionnement des entrailles d'OCaml, notamment son glaneur de cellules.

Je remercie Emmanuel pour la liberté de recherche et l'autonomie qu'il m'a accordées, ainsi que pour sa confiance et ses conseils. Et, je n'oublie pas, spécifiquement, le grand merci pour ses (très) nombreuses relectures des chapitres de ce manuscrit. Je le remercie également pour la vraie attention qu'il porte à l'intérêt de ses étudiants, c'est aussi ce qui fait le plaisir d'apprendre et d'enseigner avec lui. Merci Emmanuel.

Je remercie de tout cœur Roberto et Virginie pour avoir accepté de rapporter cette thèse et pour leurs remarques bienveillantes qui m'ont permis d'améliorer ce manuscrit. Merci aussi à Bertil, Bruno, Edmond, Pascale et Philippe pour m'avoir accordé l'honneur de faire partie de mon jury ainsi que pour leurs questions et remarques.

Merci aussi à Adrien Jonquet, Alexis Darrasse et Mathias Bourgoin qui ont pris part au projet Couverture qui est largement ancré dans cette thèse.

Je remercie tous les partenaires industriels que j'ai pu côtoyer, leurs discours et points de vue m'ont beaucoup aidé à comprendre l'esprit des normes de développement de logiciels critiques en vigueur. Je pense en particulier à Xavier Fornari, Bruno Pagano et Olivier Andrieu, de chez Esterel Technologies. Je n'oublie pas bien sûr Thomas Moniot, que je remercie beaucoup pour tous ses efforts de relectures et toutes ses remarques sur mon manuscrit en cours de rédaction. De chez AdaCore, je pense d'abord à Olivier Hainque parce qu'il a organisé la plupart des réunions auxquelles je suis allé, mais je pense aussi aux autres qui ont toujours eu un accueil très chaleureux.

~

Mes premières expressions en OCaml ont existé de manière éphémère il y a déjà plus de 8 ans. Je remercie Mathieu Jaume et Valérie Ménissier-Morain pour m'avoir initié à l'utilisation de ce langage qui refuse sans les essayer les recettes qui mélangent mal des choux et des carottes. Je dois avouer que ce n'était pas tout de suite le grand amour avec ce langage dont le compilateur râlait tout le temps. Ensuite, c'est Renaud Rioboo qui m'a fait dessiner mes premiers arbres d'inférence de types, les plus grands ont consommé beaucoup de papier ; pardon aux arbres qui en ont pâti mais c'était en somme nécessaire. Merci pour avoir été les premiers à m'enseigner la programmation en OCaml, ce merveilleux langage, et sensibilisé à la sémantique des langages de programmation.

Si Emacs n'est pas un environnement de développement parfait – il est truffé de défauts – c'est clairement le meilleur éditeur de texte à ma connaissance, et je remercie ceux qui m'ont appris à l'ap-

privoiser (ou peut-être que c'est plutôt Emacs qui m'a apprivoisé), je pense à Philippe Trébuchet, Valérie Ménissier-Morain et Christian Queinnec. Ma vision de la programmation aurait été toute autre sans cet Éditeur Merveilleux Aux Commandes Sibyllines qui m'a accompagné tout au long de la rédaction de cette thèse.

Je tiens à saisir l'occasion pour remercier avec admiration Michèle Soria, pour avoir fondé et pris la responsabilité d'APR, une équipe jeune et active mais avant tout humaine et chaleureuse où il fait bon vivre. Merci aussi pour l'avant thèse, je pense à la spécialité STL du Master d'informatique, pour moi synonyme de richesse et de liberté mais aussi de pédagogie, idéale pour satisfaire et développer une passion quand on commence à découvrir vraiment ce qui se cache derrière cette science informatique. Y enseigner par la suite a été également très enrichissant et intéressant. Merci Michèle. Et merci à toute l'équipe APR.

Il n'aurait jamais été question de motivation motivante motivée sans Benjamin Canou. J'aurais probablement moins eu envie d'emprunter le chemin du doctorat si je ne l'avais pas rencontré. Merci Benjamin, pour ta motivation, pour ton partage du goût pour la programmation, et pour ton humour « fin et délicat » aussi. Merci aussi à tous les autres occupants du bureau trois-cent-vingt-cinq, Vivien Ravet, Guillaume Pierron, Étienne Millon et Mathias Bourgoin pour les échanges animés sur la gastronomie, les goûts, la politique et bien d'autres sujets. Que vos *manos y tacos* vous apportent toujours la *fiesta* enchantée avec vos cafés.

Merci à tous ceux du LIP6 qui m'ont fait apprécier ce grand laboratoire. Pour toutes les discussions toujours intéressantes et enrichissantes qu'on a pu partager, merci à Maryse Pelletier, Jean-Luc Lamotte, Carine Pivoteau, Alexandre Chapoutot, Lionel Habib, Séverine Dubuisson, Marguerite Sos, Irphane Khan, Eugène Kamdem et bien d'autres.

Je remercie Pascal Manoury, d'une part pour ses relectures de chapitres et ses remarques sur nombre de mes formulations maladroites que j'espère avoir suffisamment raréfiées, et d'autre part pour sa disponibilité et son écoute. Merci Pascal.

Je remercie Gérard Berry pour ses talentueuses phrases expliquant les concepts compliqués de l'informatique avec des mots très simples. Ses leçons au Collège de France ont bien influencé ma vision de l'informatique. Je suis impatient d'y retourner l'écouter.

Merci à tous les enseignants que j'ai pu rencontrer, que ce soit en tant qu'étudiant ou en tant que collègue, en particulier à ceux qui n'ont pas adopté mécaniquement le vulgaire motto RTFM et qui ont été à l'écoute de leurs élèves.

Merci à tous les élèves que j'ai pu connaître, qui ont confirmé mon intérêt pour l'enseignement et qui l'ont renouvelé sans cesse. Je remercie en particulier Benoît Vaugon qui m'a probablement plus appris que je ne lui ai appris, et si ce n'est pas encore le cas, alors cela ne saurait tarder.

Je remercie l'EDITE, pour son efficacité, son dynamisme et son écoute, et en particulier merci à Marilyn Galopin et à Christian Queinnec. Je tiens aussi à remercier Bernard Robinet, je n'oublierai pas les quelques rares et précieux échanges extraordinaires que nous avions eus.

Je remercie bien sûr mes parents, pour leur patience, durant ces dernières années qui ont un peu débordé. Je remercie ma grande famille qui a eu confiance en moi et à qui je vais enfin pouvoir dire que ça y est, cette aventure est finie et de nouvelles aventures se profilent à l'horizon.

Je remercie Sandra, qui m'a soutenu tout au long et qui a fait preuve d'une (très) grande patience durant ces années de thèse. Merci ma douce Sandra.

Pour finir, je remercie aussi tous ceux que j'ai oublié de nommer sur ces deux pages. Si vous ne trouvez pas votre nom, je dédicacerai (électroniquement ou avec un stylo) votre exemplaire avec plaisir pour le faire apparaître.

Enfin, ce fut une bien belle aventure. Et si c'était à refaire ? Je referais tout différemment, mais sans une seconde d'hésitation pour me relancer : au pire, je referais tout pareil !

« *Conservons notre curiosité, respectons la vie,*
ne cessons jamais de nous émerveiller. »
– Théodore Monod

Table des matières

I Introduction **13**

Objectifs de cette thèse . . . 16
Contributions de cette thèse . . . 17
Plan du manuscrit . . . 18

II Développement de logiciels critiques et tests de couverture structurelle **19**

II.1 Criticité et sûreté des applications logicielles . . . 21
II.1.1 Développement d'un logiciel critique . . . 21
II.2 Certifications, tests et preuves formelles . . . 21
II.2.1 Activités de tests . . . 22
II.2.2 Certificats et preuves . . . 22
II.3 Normes ou guides de développement de logiciels critiques . . . 23
II.3.1 L'avionique civile en tant que domaine critique . . . 23
II.3.2 Normes pour d'autres domaines critiques . . . 25
II.4 Cycle de vie d'un logiciel et processus de développement . . . 25
II.4.1 La vie d'un logiciel . . . 25
II.4.2 Agencement de la vie d'un logiciel . . . 26
II.4.3 L'importance du processus de développement pour un logiciel critique . . . 28
II.5 Documentation et traçabilité pour les logiciels critiques . . . 29
II.6 Processus de certification (ou de qualification) . . . 30
II.7 Activités de tests . . . 31
II.8 Évolution des activités de tests . . . 32
II.8.1 Évolution des coûts . . . 32
II.8.2 Évolution des techniques . . . 32
II.9 Couverture de code . . . 32
II.9.1 Objectifs . . . 32
II.9.2 Outils pour la couverture de code . . . 33
II.9.3 Un exemple de couverture de code . . . 34
II.10 Conclusion du chapitre . . . 37

III Le Langage mCaml et la couverture structurelle des expressions **39**

III.1 Le langage mCaml . . . 42
III.1.1 Grammaire informelle des expressions du langage mCaml . . . 43
III.1.2 Sémantique opérationnelle du langage mCaml . . . 44
III.1.3 Arbre d'évaluation . . . 52
III.2 Spécification de la couverture des expressions pour mCaml . . . 53

III.2.1 Règles formelles de couverture des expressions pour mCaml 54
III.2.2 Taux de couverture 57
III.3 Conclusion du chapitre 57

IV Couverture des conditions et des décisions pour mCaml 59
IV.1 Critères de couverture pour les expressions booléennes 62
IV.1.1 Couverture des conditions (CC) 62
IV.1.2 Couverture des décisions (DC) 63
IV.1.3 Couverture des conditions/décisions (C/DC) 63
IV.1.4 Couverture des conditions multiples (MCC) 63
IV.1.5 Couverture des conditions/décisions modifiée (MC/DC) 64
IV.1.6 Autres critères de couverture pour les expressions booléennes 69
IV.2 Conditions et décisions pour mCaml : quatre sémantiques formelles 71
IV.2.1 Notations 71
IV.2.2 Sémantique 1 : repérage syntaxique aux branchements conditionnels 72
IV.2.3 Sémantique 2 : sémantique intermédiaire, généralisation aux opérateurs booléens 74
IV.2.4 Sémantique 3 : une extension de la sémantique 2, généralisation aux applications à retour booléen 77
IV.2.5 Sémantique 4 : généralisation aux expressions booléennes 78
IV.3 Choix de la sémantique 84
IV.4 Fonction de comptage des conditions d'une expression 84
IV.5 Conclusion du chapitre 85

V Couverture de code structurelle intrusive pour mCaml 87
V.1 Instrumentation de code mCaml pour la couverture des expressions 90
V.1.1 Instrumentation la plus générale 90
V.1.2 Instrumentation simple 91
V.1.3 Instrumentation avancée pour la couverture structurelle 92
V.1.4 Sémantique opérationnelle préservée 99
V.2 Instrumentation de code mCaml pour la mesure de satisfaction du critère MC/DC . . 99
V.3 Conclusion du chapitre 104

VI Couverture de code structurelle non-intrusive pour mCaml 105
VI.1 Instrumentation pour la couverture structurelle simple 108
VI.1.1 Informations de mise au point produites par `ocamldebug` 109
VI.1.2 Génération des marques pour la couverture simple des expressions du langage mCaml 110
VI.2 Instrumentation pour les mesures des conditions et décisions 113
VI.2.1 Instrumentation de l'environnement d'exécution pour la mesure MC/DC 113
VI.2.2 Production de rapports de couverture avec la mesure MC/DC 116
VI.3 Conservation de la sémantique opérationnelle 117
VI.4 Conclusion du chapitre 117

VII Comparaison entre les approches intrusive et non-intrusive 119
VII.1 Comparaison sémantique des deux techniques 121
VII.1.1 Équivalence des résultats produits par les deux techniques en l'absence de traitement particulier des appels terminaux 121

VII.1.2 Une différence notable des deux approches : les appels terminaux 125
VII.2 Comparaison pratique des approches intrusive et non intrusive 127
VII.2.1 Du point de vue de l'utilisateur 127
VII.2.2 Avantages et inconvénients 128
VII.3 MLcov : couverture structurelle par instrumentation du code source 129
VII.3.1 Caractéristiques générales de l'outil MLcov 129
VII.3.2 Sémantique utilisée par MLcov 130
VII.3.3 Un exemple de rapport de couverture avec MLcov 130
VII.4 Zamcov : couverture structurelle non intrusive 137
VII.4.1 Génération des informations de couverture 137
VII.4.2 Illustration 138
VII.5 Conclusion du chapitre 142

VIII Conclusion et Perspectives **143**

Bibliographie **151**

A Sémantique formelle du langage mCaml (annexe du chapitre 3) **157**
A.1 La grammaire en BNF 157
A.2 Système de types du langage mCaml 161
A.2.1 Notations 161
A.2.2 Environnements initiaux 164
A.2.3 Les définitions de types 164
A.2.4 Le typage des définitions 166
A.2.5 Le typage des expressions 166
A.3 Les levées d'exceptions dans la sémantique opérationnelle 171

B Notes explicatives **175**
B.1 À propos de la ZAM, machine virtuelle OCaml 175
B.1.1 Historique 175
B.1.2 Caractéristiques générales 175
B.2 Appels terminaux et appels récursifs terminaux 178
B.2.1 Appels terminaux 178
B.2.2 Appels récursifs terminaux 178
B.2.3 Optimisation des appels terminaux 178
B.2.4 Les appels terminaux avec la ZAM 179
B.2.5 Les appels externes avec la ZAM 180

C Notes d'implantation **181**
C.1 Implantation de la génération des traces des conditions et des décisions par réécriture du code source 181
C.2 Génération de tests pour MC/DC 183

CHAPITRE I

Introduction

« Un ordinateur, c'est une machine hyper stupide et hyper obéissante.
C'est le contraire d'un Homme.
Ça va très très vite, ça ne se trompe jamais, c'est stupide. »
– Gérard Berry

Sommaire

Objectifs de cette thèse 16
Contributions de cette thèse 17
Plan du manuscrit 18

Matériel numérique, logiciels, et mises à jour

Une grande partie du matériel numérique fonctionne grâce à la présence de logiciels. Même lorsqu'un matériel numérique fonctionne sans logiciel – cela arrive quand son rôle est encodé directement dans son circuit électronique – il y a de fortes chances qu'il communique avec un matériel qui, lui, possède un logiciel.

Les logiciels permettent de fabriquer un dispositif numérique et de l'utiliser de plusieurs manières. En effet, plutôt que d'inscrire une fois pour toutes un rôle très spécifique et plus ou moins complexe dans le système fabriqué en petite quantité, autant produire en très grande quantité un matériel sachant endosser de nombreux rôles, pour peu qu'on lui dise correctement comment faire.

Par exemple, une machine sachant lire, comparer, additionner et soustraire des nombres entiers positifs, et permettant de définir des fonctions, est une machine à qui on peut demander simplement des multiplications sur des nombres entiers positifs. Il suffit d'écrire en langage machine la fonction mathématique $produit : \mathbb{N}^2 \to \mathbb{N}$ suivante, définie inductivement :

$$produit(x,y) = \begin{cases} 0 & \text{si } x = 0 \\ 0 & \text{si } y = 0 \\ y & \text{si } x = 1 \\ x & \text{si } y = 1 \\ x + produit(x, y-1) & \text{sinon} \end{cases}$$

Ainsi, si la machine est bien conçue et bien fabriquée, alors on peut lui ajouter la capacité de multiplication. Et si on se trompe sur la définition de la multiplication, on peut corriger la définition sans changer de matériel puisqu'elle est logicielle. La même erreur commise sur le matériel entraîne le recyclage du matériel (ou sa mise au rebut).

Avec les machines modernes, on trouve un compromis sur le nombre d'instructions gérées par le processeur. On ne veut pas que le processeur ait trop de fonctionnalités car cela coûte cher, ni trop peu pour des raisons pratiques et de performances.

Par exemple, on n'a pas forcément l'envie de redéfinir la multiplication à partir de l'addition – ce travail supplémentaire peut être mal accueilli puisqu'il coûte. On veut que le processeur sache tout de suite faire les opérations de base, pour pouvoir se concentrer sur des problématiques qui s'abstraient de ces opérations. De manière réaliste, il faut aussi considérer les performances. Une multiplication bien encodée dans un circuit est potentiellement beaucoup plus rapide à effectuer que sa version basée sur l'utilisation de l'addition et des applications de fonctions.

Les logiciels sont alors un moyen immatériel de contrôler mais surtout de mettre à jour du matériel électronique sans en changer le moindre composant électronique [1]. Par exemple, pour un appareil photo numérique, si on trouve un moyen de faire la mise au point plus rapidement sans demander une puissance de calcul supérieure, une mise à jour du logiciel interne de l'appareil suffit à lui donner cette capacité.

Le logiciel peut être en charge d'une partie (voire de la totalité) des fonctionnalités d'un système numérique du point de vue de l'utilisateur final. Cela représente généralement un gain de coût de développement puisqu'il est bien plus facile de modifier ou de faire évoluer un programme qu'un circuit électronique.

Toutefois le logiciel est aussi victime de sa facilité de mise en œuvre. Celle-ci a rapidement conduit à la complexification des dispositifs logiciels (modèles et langages de programmation, environnement

1. *À mi-chemin, les circuits logiques programmables sont du matériel dont le fonctionnement matériel est programmable une ou plusieurs fois.*

système, etc.) Plus un système est complexe, plus il est facile d'y introduire des erreurs et plus il est ardu de les éradiquer. C'est ce qu'illustrent les quelques célèbres exemples suivants.

Quelques accidents

Quelques exemples dans un contexte non critique

- en 1993/1994[41, 43], certains Pentium arrondissaient beaucoup trop les calculs sur les nombres à virgule flottante [2] ;
- le 20 avril 1998, le fameux BSOD (*blue screen of death*) fait son apparition durant la démonstration de lancement du système d'exploitation Microsoft Windows 98 ;
- l'année bissextile 2008 a révélé une boucle infinie dans le code des lecteurs MP3 Microsoft Zune ;
- en 2008, une faille vieille de près de 2 ans a été découverte et rendue publique dans les paquets Debian de OpenSSL ;
- en 2009, Mac OS X.6 fraîchement sorti supprimait sauvagement des fichiers lorsque le compte invité était utilisé ;
- en 2012, le passage à l'heure d'été a été effectif avec une semaine de retard sur les Freebox.

Quelques exemples dans des domaines plus critiques

- entre juin 1985 et janvier 1987, des traitements contre le cancer par l'appareil Therac-25 ont entraîné morts et blessures graves[53] par overdose de radiation ;
- en 1996, la première fusée Ariane V a explosé lors de son premier vol à cause d'un dépassement de capacité d'un calcul arithmétique [26] ;
- en 1997, Mars Pathfinder (un robot envoyé sur la planète Mars) fut victime d'un interblocage suite à une inversion de priorités [42].

Le gain introduit par la facilité du développement de logiciel a été fortement compromis par ces erreurs non détectées à temps.

Objectifs de cette thèse

Un des éléments importants du progrès dans le domaine du logiciel est l'augmentation de l'utilisation de **langages de programmation de haut niveau d'abstraction**. L'objectif principal de cette thèse est d'introduire dans le domaine des logiciels critiques cet élément de progrès. Cela peut dans certains contextes permettre aux développeurs de s'affranchir d'une quantité de problèmes qui ne sont pas inhérents à leurs activités. Par exemple, lors du développement d'un logiciel produisant un binaire destiné à être embarqué dans un système critique, il ne paraît pas nécessaire d'avoir à contrôler « à la main » la gestion de la mémoire, un gestionnaire automatique de mémoire faisant bien mieux avec un effort bien moindre dans ce contexte. Cela est généralisable à tous les outils de développement qui ne se retrouvent pas embarqués au sein même des systèmes critiques.

Le développement des logiciels critiques est soumis à un nombre important de contraintes : documentation, revue, tests et mesures de couverture, etc. Plus particulièrement, dans cette thèse, on

2. Les nombres à virgule flottante sont généralement des approximations de nombres réels à cause de leur encodage : sur un nombre fini de bits, on ne peut pas proposer un encodage des nombres réels sans approximation. Ces approximations bien connues ne sont pas des erreurs. Mais lorsque les arrondis ne viennent plus de l'encodage même des données mais des opérations de calculs, le système est considéré comme étant erroné.

s'intéresse à la couverture de code structurelle, qui est intimement liée aux activités de tests. Et plus précisément, on étudie la génération de rapports de couverture structurelle de code pour un langage de haut-niveau de la famille ML.

Un rapport de couverture de code structurelle doit apporter la preuve que toutes les parties d'un programme ont été utilisées lors de la procédure de test. Si un rapport montre que du code n'a pas été activé, cela suscitera des interrogations : sert-il à quelque chose ? S'il ne sert à rien, il faut probablement le supprimer. Mais s'il sert, alors il faut justifier sa présence ou bien refaire des tests qui permettent d'établir un nouveau rapport qui montre que le code a pu être activé. En effet, on doit éviter à un moment critique l'activation d'un code jamais activé durant la phase de vérification.

Plusieurs critères de couverture de code structurelle existent. Dans cette thèse, nous en utilisons un certain nombre, notamment en ce qui concerne la couverture des expressions à valeurs booléennes. Ces expressions ont la particularité d'entraîner les décisions lors des branchements. Par exemple, le critère MC/DC (couverture des conditions/décisions modifiée), qui sera précisément défini dans le chapitre IV, pose une problématique importante que nous développerons dans plusieurs chapitres. Ce critère est notamment exigé pour les développements de logiciels pour l'avionique civile tel que c'est indiqué dans le guide DO-178B [3]. L'avionique civile est actuellement connue pour être le domaine le plus exigeant des domaines critiques, en matière de qualité du processus de développement.

Cette thèse traite deux sujets principaux. Le premier est l'interprétation des critères de couverture structurelle de code pour un langage de haut-niveau de la famille ML. Le second est l'application du premier pour l'élaboration d'un outil de couverture de code. Deux approches d'implantation de l'outil sont alors présentées. D'abord, nous présentons une version par réécriture du code pour la génération de traces d'exécution, que nous appelons « intrusive » car au final le programme qui sert à la mesure de couverture n'est pas exactement le même puisque c'est le programme de base en version enrichie. Cela est l'approche habituelle. Ensuite, nous présentons une version sans réécriture du code, et produisant les mêmes mesures, que nous appelons « non-intrusive » du fait que le binaire servant à la mesure est intact. C'est alors l'environnement d'exécution qui est enrichi pour pouvoir faire les mesures.

Contributions de cette thèse

- Les spécifications formelles des critères de couverture des conditions, des décisions et d'autres critères usuels (dont MC/DC) dans le domaine de la mesure de couverture de code structurelle pour les logiciels critiques, pour le langage support (mCaml) de la thèse qui est également formellement défini. *Le langage* mCaml *est un langage d'expressions, en ce sens qu'il n'y a pas de distinction entre expressions et instructions.*
- Comme la notion de condition et de décision n'est en fait absolument pas simple pour un langage d'expressions comme mCaml à moins de fortement le restreindre, nous donnons quatre sémantiques des conditions et des décisions pour ce langage. Nous commençons par appliquer directement les définitions usuelles en posant de fortes restrictions sur le langage. Cela réduit alors son expressivité. Progressivement, en trois étapes, nous supprimons les contraintes posées sur le langage en les remplaçant par des obligations de mesures.
- Les spécifications formelles d'une technique classique de génération de traces d'exécution pour les critères de mesures précédemment définies, par réécriture de code (appelée « intrusive »), pour le langage mCaml .

3. La DO-178C [74] commence en 2012 à remplacer la version précédente DO-178B [73] qui date de 1992.

- Les spécifications formelles d'une autre technique innovatrice, équivalente en résultats produits, sans réécriture de code (appelée « non-intrusive ») mais avec instrumentation de l'environnement d'exécution.
- Une large comparaison des deux approches intrusive et non-intrusive.

Plan du manuscrit

Dans le chapitre suivant, nous décrivons une partie de l'ingénierie du développement de logiciels dans le cadre de systèmes critiques, la place des activités de vérifications, et en particulier la couverture structurelle. Ensuite, le chapitre III présente le langage de programmation mCaml, basé sur ML. Le langage mCaml est défini formellement par sa grammaire ainsi que par son système de types et sa sémantique opérationnelle. Toutefois, pour se concentrer sur ce qui nous importe le plus, la grammaire formelle et le système de types sont déportés en annexe A, ce qui laisse place dans le chapitre III à une grammaire informelle (plus facile à lire) et la sémantique opérationnelle du langage. Nous proposons alors une définition formelle du critère de couverture des expressions pour mCaml. Il s'agira de ne laisser aucun doute sur la définition. Ce langage sera utilisé dans tous les chapitres suivants. Le chapitre IV est consacré aux critères de couverture des expressions à valeurs booléennes. En effet, passer de définitions données pour un langage minimaliste à des définitions pour un langage riche est une étape qui demande une réflexion importante. Différentes interprétations, donc différents compromis, sont présentés et discutés. Les définitions établies, le chapitre V présente la couverture de code structurelle intrusive pour mCaml. Il s'agit de présenter une réécriture pour l'injection d'instructions d'émission de traces, ainsi que les analyses statiques nécessaires pour mesurer MC/DC. Le chapitre VI traite d'une nouvelle technique, non intrusive en ce sens qu'elle ne réécrit pas le code mais le laisse intact. Le programme utilisé pour la couverture de code peut alors être exactement le même programme que celui utilisé pour la version en production. Le chapitre VII compare ces deux approches et détermine dans quel cadre elles sont équivalentes. Deux outils, l'un adoptant la méthode intrusive et l'autre la méthode non intrusive sont également présentés dans ce chapitre. Enfin, le chapitre VIII conclut la thèse et présente les perspectives.

CHAPITRE II

Développement de logiciels critiques et tests de couverture structurelle

« No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different ? » – Lauren Ruth Wiener [86]

Sommaire

II.1 Criticité et sûreté des applications logicielles 21
II.1.1 Développement d'un logiciel critique 21
II.2 Certifications, tests et preuves formelles 21
II.2.1 Activités de tests 22
II.2.2 Certificats et preuves 22
II.3 Normes ou guides de développement de logiciels critiques 23
II.3.1 L'avionique civile en tant que domaine critique 23
II.3.2 Normes pour d'autres domaines critiques 25
II.4 Cycle de vie d'un logiciel et processus de développement 25
II.4.1 La vie d'un logiciel 25
II.4.2 Agencement de la vie d'un logiciel 26
II.4.3 L'importance du processus de développement pour un logiciel critique 28
II.5 Documentation et traçabilité pour les logiciels critiques 29
II.6 Processus de certification (ou de qualification) 30
II.7 Activités de tests 31
II.8 Évolution des activités de tests 32
II.8.1 Évolution des coûts 32
II.8.2 Évolution des techniques 32
II.9 Couverture de code 32
II.9.1 Objectifs 32
II.9.2 Outils pour la couverture de code 33
II.9.3 Un exemple de couverture de code 34
II.10 Conclusion du chapitre 37

Résumé du chapitre 2

Ce chapitre est consacré au génie du développement de logiciels critiques. Il y est question de la place de la traçabilité et des activités de vérification dans le processus de certification. Le but est de sensibiliser le lecteur aux méthodes contraignantes rigoureuses qui sont en vigueur ainsi qu'aux coûts engendrés par les activités de vérification. Ce chapitre sert aussi à introduire la notion de couverture de code.

Les pratiques les plus exigeantes sont celles du domaine des logiciels de l'avionique civile. Suivant le guide DO-178B, elles se basent sur la qualité du processus de développement, partant du constat qu'il n'est pas possible de connaître exactement la quantité d'erreurs résidant dans un logiciel à la fin de son développement. Le principe est de considérer qu'on délivre le meilleur logiciel possible en appliquant les meilleures méthodes de développement connues.

D'un point de vue général, cette thèse propose une formalisation d'outils de couverture de code structurelle pour un langage de haut-niveau d'abstraction qui est précisément défini dans le chapitre III.

Ce chapitre aborde très succinctement le très large domaine du génie logiciel [8, 9, 7] pour donner le contexte dans lequel se place la couverture de code, c'est-à-dire au sein des activités de vérification d'un processus de développement d'un logiciel (éventuellement) critique.

II.1 Criticité et sûreté des applications logicielles

Définitions

• Un *système critique* est un système dans lequel une panne peut avoir des conséquences dramatiques, tels des morts ou des blessés graves, des dommages matériels importants, ou des conséquences graves pour l'environnement ou l'économie.

• Un *logiciel critique* est un composant d'un système critique. Son dysfonctionnement mettrait en péril la sécurité de personnes ou aurait d'autres conséquences graves.

• On parle de *sûreté de logiciels* ou de *logiciels sûrs* quand on veut s'assurer qu'ils ne se comportent pas de manière non conforme aux attentes (*e.g.*, arrêt intempestif, calculs erronés).

Remarque. La *sécurité des logiciels* se base sur l'étude de sa vulnérabilité et sa capacité à résister aux attaques. C'est un domaine qui n'est pas traité dans cette thèse et qu'il ne faut pas la confondre avec la sûreté des logiciels.

II.1.1 Développement d'un logiciel critique

Les développements de logiciels critiques suivent des pratiques plus contraignantes que ceux pour des logiciels non critiques, afin d'obtenir le meilleur niveau de qualité et de sûreté. *On rappelle qu'une défaillance d'un logiciel critique causant la mort de personnes pourrait avoir des coûts sans limite.*

L'avionique civile est le domaine qui impose les processus de développement les plus stricts et contraignants. Le ferroviaire possède des normes similaires avec toutefois un contexte de déploiement différent.

Le guide DO-178B[73] donne probablement la plus connue des pratiques de développement de logiciels critiques actuellement, notamment du fait de sa rigueur. C'est un guide destiné au domaine des logiciels utilisés dans l'avionique civile. Un logiciel développé pour un avion à usage civil doit répondre aux exigences de ce document.

La version DO-178C[74], publiée en décembre 2011, commence à remplacer la DO-178B. En effet, d'une part la technologie évolue. On veut introduire de nouvelles techniques de développement de logiciels, mais aussi et surtout de vérification. D'autre part, les besoins évoluent aussi. On n'attend pas la même chose d'un avion aujourd'hui qu'on en attendait il y a vingt ans [1].

II.2 Certifications, tests et preuves formelles

Définition. Un certificat est un document qui atteste du suivi et du respect d'une certaine quantité d'exigences. Il peut être un contrôle de savoirs ou savoir-faire. Il peut être la déclaration de possession d'un bien matériel. Ou encore il peut attester du respect d'un processus de production. En général, un certificat doit être vérifiable, en ce sens qu'il doit exister une démarche permettant de le vérifier.

1. La révision B de la D0-178 date de 1992 [73], la révision A précédente datant de 1985 [72].

À propos de la notion de « certificat »
Voici quelques exemples de certificats dans des contextes très différents.
- un certificat d'aptitude professionnelle (C.A.P.) ;
- un certificat d'immatriculation (ex-carte grise) ;
- un certificat d'agriculture biologique ;
- un certificat SSL/TLS (Secure Sockets Layer/Transport Layer Security).

Pour les logiciels critiques, un certificat est délivré pour un logiciel dont le développement a suivi un cahier des charges précis et rigoureux. Un certificat n'a alors de valeur que si le cahier des charges est pertinent et que l'autorité de certification est de confiance et compétente. *Parmi les quelques exemples de certificats d'autres domaines présentés plus haut, c'est le certificat d'agriculture biologique qui en est le plus analogue.*

II.2.1 Activités de tests

Les activités de tests font partie de la plupart des processus de certification des logiciels.

Selon Myers [62], leur premier but est de trouver les erreurs, si bien qu'un ensemble de tests ne mettant aucune erreur en évidence est a priori considéré en échec. Il s'agit de vérifier que pour certaines entrées, on obtient bien les sorties attendues, mais surtout il s'agit de chercher et trouver les scénarios où les sorties attendues ne correspondent pas aux sorties obtenues. Ces activités permettront également de vérifier qu'on n'a pas écrit de code inatteignable.

Leur second but est de montrer qu'on a bien envisagé tous les cas de figure et qu'on les a testés, pour ne pas avoir de mauvaise surprise lors de la mise en production.

Si le développeur peut faire ses propres tests pour dénicher ses propres erreurs, au final le testeur doit préférablement être une autre personne afin de ne pas biaiser les tests. Le testeur vérifie ainsi le travail du développeur, donc la conformité de la production du développeur par rapport aux spécifications du cahier des charges. La plupart du temps, le testeur n'a pas accès au code produit par le développeur. Il doit produire ses tests d'après les spécifications, et non d'après le code du développeur.

Le coût des activités de tests est grand par rapport au coût total du développement d'un logiciel. Mais il est généralement préférable de payer ce coût le plus tôt possible pour éviter les mauvaises surprises une fois le produit mis en production.

Cependant, devant la part importante du coût du test, découle rapidement une volonté de réduire ce coût sans altérer la qualité du produit.

De plus, les logiciels devenant de plus en plus complexes, que ce soit à cause de leurs tailles ou des fonctionnalités intrinsèques, les coûts de développement et de tests augmentent.

II.2.2 Certificats et preuves

Vis-à-vis de la sûreté de fonctionnement, un certificat n'est pas une preuve au sens mathématique ou formel du terme. Ainsi, un logiciel certifié n'est généralement pas un logiciel dont le fonctionnement et l'implantation ont été entièrement prouvés mathématiquement. Cependant, une preuve mathématique peut être un certificat, en partie ou entièrement.

En effet, les avancées dans les méthodes formelles sont tout à fait intéressantes. Par exemple, le projet CompCert[51] propose un compilateur ciblant un assembleur PowerPC pour un grand sous-ensemble du langage C, dont la majeure spécificité est de fournir la preuve mathématique des différentes transformations de langage effectuées par le compilateur. Cela signifie que la propriété de

conservation de la sémantique du programme, lors des traductions d'un langage à un autre langage, est prouvée.

Sans aller jusqu'à une certification totale d'un logiciel par la preuve, son utilisation partielle a été validée dans le domaine de l'avionique civile. Notamment, CAVEAT, un prouveur pour programmes écrits en langage C et développé au Commissariat à l'Énergie Atomique, a été utilisé dans le processus de vérification d'un logiciel critique destiné à l'avionique civile par Airbus [79].

Les preuves servent à montrer et à garantir des propriétés de sûreté. Cependant les tests ne pourront probablement jamais être remplacés entièrement par les preuves, car on veut s'assurer avec les tests que les spécifications sont bien implantées mais aussi qu'elles expriment effectivement ce que leurs auteurs pensent qu'elles expriment. Ce sera nécessaire tant que les langues naturelles exprimeront des propos sujets à interprétation et interviendront dans la conception des logiciels. D'autre part, actuellement, il est techniquement plus facile de faire des tests que de faire des preuves dès que les programmes deviennent complexes.

La phrase suivante de Donald Knuth illustre la situation de la preuve face au test : « *Beware of the above code ; I have only proved it correct, not tried it.* ».

II.3 Normes ou guides de développement de logiciels critiques

II.3.1 L'avionique civile en tant que domaine critique

Avant l'arrivée des logiciels critiques, il y avait de manière plus générale les systèmes critiques qui n'embarquaient pas de logiciels. Les premiers avions de ligne civils datent des environs de 1920. Ils permettaient de transporter une dizaine de personnes. Après la seconde guerre mondiale, sont apparus les avions de ligne transportant une centaine de personnes (*e.g.*, Boeing 377). Aujourd'hui, les avions de ligne transportent jusqu'à plus de 800 passagers (*e.g.*, Airbus A380) et surtout, ils **embarquent des logiciels de plus en plus nombreux et complexes**.

Pour qu'un logiciel critique puisse être utilisé, il faut obtenir l'aval de l'autorité de régulation. Pour l'avionique civile, aux États-Unis, il s'agit de la *Federal Aviation Administration* (FAA) [2]. Son homologue européen est l'Agence Européenne de la Sécurité Aérienne (AESA) [3]. Dans le contexte des systèmes critiques, ces agences sont chargées de **promouvoir le plus haut niveau possible de sécurité pour les passagers**.

Ces agences exigent que les systèmes critiques soient conçus de la manière la plus sûre possible. Pour les logiciels, tout comme pour le matériel (au sens large, c'est-à-dire incluant le matériel mécanique et le matériel électronique), elles recommandent **le suivi de normes ou de guides**.

Aux États-Unis, la *Radio Technical Commission for Aeronautics* (RTCA, Inc.) [4] est l'organisation chargée de développer des guides techniques pour les autorités gouvernementales et industrielles servant à l'avionique, et la FAA est son principal soutien. En 1980, la RTCA a formé un comité (SC-145) pour établir un guide pour le développement des logiciels destinés à l'avionique, c'est lui qui a produit le document « *Software Considerations in Airborne Systems and Equipment Certification* », édité en janvier 1982 sous la référence DO-178 [46]. Un autre comité a été formé pour produire une première révision du document, ce qui a donné en 1985 la publication de la DO-178A [72]. En 1989, un nouveau comité a été formé pour travailler sur une nouvelle version afin de traiter de nouvelles problématiques dont la qualification des outils, ce qui a donné en 1992 la publication de la version la plus connue à

2. http://www.faa.gov/
3. http://fr.wikipedia.org/wiki/Agence_europeenne_de_la_securite_aerienne
4. http://www.rtca.org/

ce jour qui est la DO-178B [73]. En décembre 2011, la DO-178C tant attendue arrive et complète la DO-178B en répondant à de nouvelles demandes dont l'introduction des « méthodes formelles » [74]. Celles-ci avaient cependant déjà été employées dans le cadre de développements DO-178B [80].

La nouvelle révision DO-178C remplacera peu à peu son prédécesseur DO-178B ; mais il faut du temps pour que la transition se fasse notamment en raison des projets qui ont démarré avant sa publication. *Dans cette thèse, on fera par défaut référence à la version DO-178B. La plupart du temps, ce qui fait référence dans cette thèse à la DO-178B reste valide pour la DO-178C.*

L'équivalent européen de la RTCA est la *European Organization for Civil Aviation Equipment* (EUROCAE) [5]. Ces deux organisations consœurs co-éditent le guide DO-178, mais la EUROCAE lui donne la référence ED-12.

DO-178B est le guide le plus connu à ce jour dans le milieu des éditeurs de logiciels critiques.

II.3.1.1 DO-178B en tant que norme.

Ce guide est utilisé et traité en pratique comme une norme obligatoire, bien qu'officiellement, il n'y ait pas d'obligation de le suivre pour obtenir l'aval des autorités de certification. L'important est de suivre une rigueur permettant de réduire au minimum possible les erreurs dans les produits.

Cependant, il sera généralement « plus simple » de se conformer au document le plus standard (*i.e.*, en pratique le plus utilisé et le plus accepté) pour augmenter les chances d'être accepté et validé par les autorités qui se basent sur ces guides.

II.3.1.2 Cinq niveaux de criticité pour la DO-178B

- niveau A : un défaut du système ou sous-système étudié
  peut provoquer un problème catastrophique.
  Sécurité du vol ou atterrissage compromis. Crash de l'avion.
- niveau B : un défaut du système ou sous-système étudié peut provoquer
  un **problème majeur entraînant des dégâts sérieux voire la mort** de quelques occupants.
- niveau C : un défaut du système ou sous-système étudié peut provoquer un **problème sérieux entraînant un dysfonctionnement des équipements vitaux** de l'appareil.
- niveau D : un défaut du système ou sous-système étudié peut provoquer
  un **problème pouvant perturber la sécurité du vol**.
- niveau E : un défaut du système ou sous-système étudié peut provoquer
  un problème **sans effet sur la sécurité du vol**.

Figure II.1 – Les cinq niveaux de criticité selon la DO-178B

Cinq niveaux de criticité (de A à E) s'appliquent à l'avionique civile, selon la figure II.1.

Ce qui est soumis au niveau E n'implique aucune contrainte particulière. Le logiciel peut faire à peu près n'importe quoi, les autorités n'y feront pas attention puisque la sécurité du vol n'est aucunement affectée.

Dès le niveau D, le logiciel est soumis à des contraintes, comme une documentation vérifiée et la couverture fonctionnelle. Le logiciel doit être géré en configuration et la traçabilité doit être assurée.

5. http://www.eurocae.net/

À chaque niveau supérieur, des contraintes s'ajoutent en plus de celles du niveau directement inférieur.

Au niveau C, la couverture de code structurelle est demandée. Il ne peut plus y avoir de code mort non justifié. Le modèle de développement doit être précisé.

Au niveau B, on demande la couverture des décisions. Les tests doivent être effectués par une équipe indépendante.

Au niveau A, on demande la couverture des conditions/décisions modifiée (MC/DC), critère qui sera précisément défini dans le chapitre IV (page 59).

En pratique, les éditeurs de logiciels distinguent assez peu les niveaux A et B du fait de contraintes très similaires.

II.3.2 Normes pour d'autres domaines critiques

Niveaux de criticité. Pour d'autres domaines que l'avionique civile, les niveaux de criticité sont connus sous les références SIL1 à SIL4 (*Safety Integrity Level*) [55], avec parfois une définition d'un SIL0 équivalent au niveau E de la DO-178B.

Dans le domaine du ferroviaire, les standards EN 50128 (Logiciel pour les systèmes de contrôle et de protection ferroviaires) et EN 50129 (Systèmes de signalisation électroniques reliés à la sécurité) décrivent les exigences à respecter en fonction des niveaux de sécurité (SIL0 à SIL4).

Dans l'industrie en général, c'est le standard international IEC-61508 (*Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*) [6] qui s'applique aux domaines tels que les centrales nucléaires, les matériels médicaux et l'industrie pétrolière.

Il existe de nombreux autres standards spécifiques, pour l'aérospatial, les équipements médicaux (*e.g.*, dosage de rayons X), etc.

Les autres domaines imposent des contraintes similaires, ou généralement moindres, à celles imposées par le domaine de l'avionique civile.

II.4 Cycle de vie d'un logiciel et processus de développement

II.4.1 La vie d'un logiciel

La vie d'un logiciel commence lorsqu'on exprime les besoins qui justifient sa création. Une fois le logiciel créé, sa vie continue tant qu'il est utilisé et maintenu.

Entre temps, on peut distinguer de nombreuses phases de réalisation qui s'agencent selon l'environnement dans lequel le logiciel prend vie. Si l'étape d'implantation peut paraître la plus évidente et indispensable des phases, d'autres ont également un rôle important à jouer.

Ainsi, une fois le besoin identifié, une étude de faisabilité peut être menée. Pour une entreprise, une analyse des coûts et bénéfices est difficilement évitable. Par raffinements, on détermine à partir du besoin les exigences fonctionnelles détaillées de haut-niveau puis les exigences de plus bas-niveau permettant de passer à l'étape de l'écriture même du logiciel, le codage. Le code pourra subir différents jeux de tests pour chercher les erreurs, qui peuvent se cacher dans le code mais aussi dans les différents niveaux de spécifications. Après les étapes de tests, le logiciel peut être prêt à être livré au client, qui trouvera peut-être de nouvelles erreurs ou de nouveaux besoins, ce qui nous amène à l'étape de maintenance du logiciel ou bien au début du cycle pour concevoir un nouveau logiciel.

6. http://www.iec.ch/functionalsafety/

Toutes ces étapes de développement ont des importances très variables, et rencontrent des obstacles qui dépendent de l'environnement de développement et de la compétence des réalisateurs du projet.

II.4.2 Agencement de la vie d'un logiciel

Pour formaliser le cadre de développement des logiciels, différents modèles de gestion de projets ont été définis (parfois basés sur l'expérience d'autres domaines). La figure II.2 présente le cycle de développement d'un logiciel de manière générale.

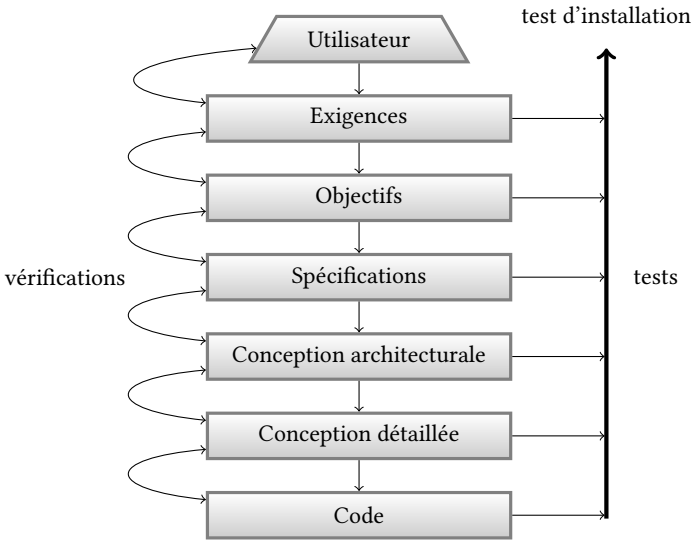


FIGURE II.2 – Une représentation du processus général de développement logiciel

On peut observer deux catégories de modèles de processus de développement logiciel : celle qui privilégie les tâches en proposant un cycle de développement long, et celle qui privilégie les unités de temps en proposant des cycles de développement courts.

II.4.2.1 Un exemple de cycle long : le cycle en V

Le cycle en V (figure II.3) propose un cycle global pour le développement d'une application. C'est une proposition d'amélioration du modèle en cascades (*waterfall model* en anglais), en renforçant les étapes d'intégration et de vérification.

Ce modèle met en avant la traçabilité et l'importance du raffinement successif des exigences, qui sont représentés par les flèches sur la figure II.3.

Ce modèle est bien accepté par les autorités de certification DO-178B notamment grâce à la mise en avant de la traçabilité.

II.4.2.2 Un exemple de cycle court : le cycle Agile

Agile, représenté en figure II.4, propose une interaction fréquente (*i.e.*, mensuelle, hebdomadaire, voire quotidienne) entre toutes les différentes étapes du processus. On n'attend pas qu'une grande étape soit terminée pour passer à la suivante ; chaque jour et par raffinement on aborde toutes les étapes du développement ou un grand nombre d'entre elles.

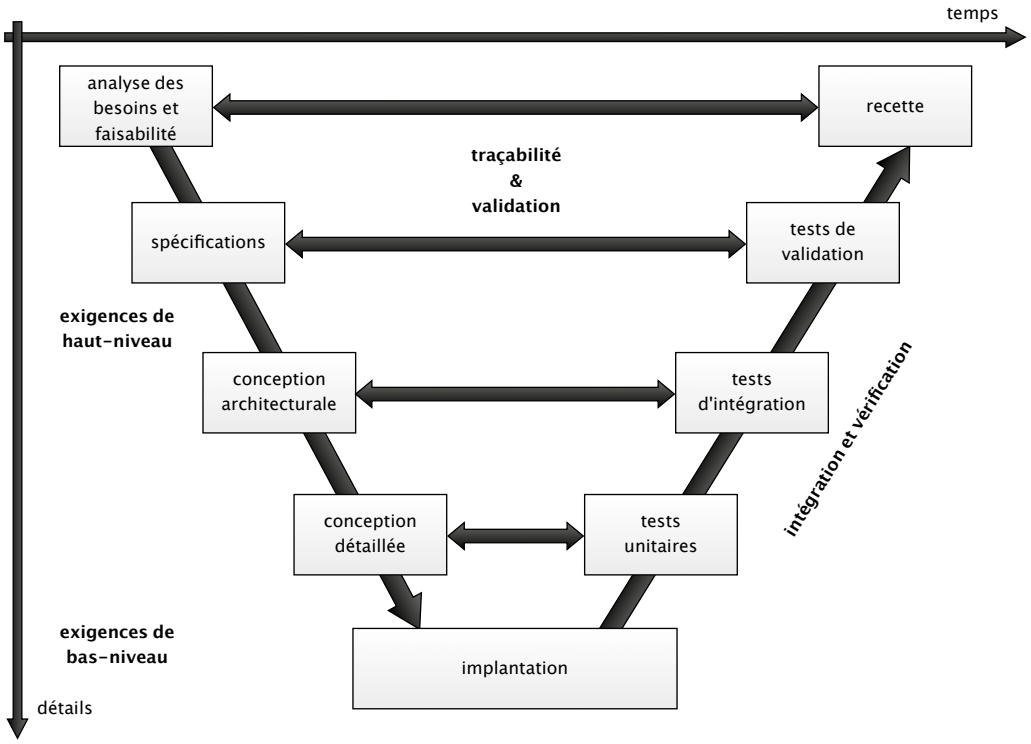


FIGURE II.3 – Une représentation du modèle de développement logiciel « Cycle en V »

Ce modèle présente l'avantage d'être très robuste aux exigences changeantes du client car l'interaction avec lui est très fréquente. C'est particulièrement intéressant quand le besoin est difficile à bien déterminer ou lorsqu'il dépend de facteurs incontrôlables du marché.

Le manifeste régissant ce modèle est présenté dans la figure II.5.

II.4.2.3 Avantages et inconvénients des différents cycles de développement

Pour les processus DO-178B, il est moins intéressant d'adopter un processus de développement à cycle court car le besoin est plus rapidement défini que la moyenne et qu'on perd un peu en traçabilité à cause des très nombreux cycles, alors qu'un cycle long demande à ce qu'une étape soit bien traitée avant de passer à la suivante.

D'autre part, le cycle Agile privilégie l'obtention rapide d'un logiciel qui fonctionne au détriment de la documentation. C'est-à-dire qu'on arrive plus rapidement à un logiciel qui fonctionne mais qui risque d'être peu ou mal documenté. Le manifeste du modèle Agile (figure II.5, page 29) donne une importance secondaire au respect du processus et à la documentation, ce qui est difficilement compatible avec la DO-178B qui se base sur la qualité du processus et de la documentation.

Pour les logiciels modernes destinés au grand public qui sont de plus en plus nombreux (notamment avec l'arrivée des smartphones et des tablettes, sur systèmes Android ou iOS), les cycles courts sont de plus en plus appréciés car ils permettent de proposer très rapidement un produit au client et ainsi commencer à jauger le marché ainsi qu'à vendre donc amorcer les recettes.

Chaque modèle ayant des avantages et des inconvénients qui varient selon le projet auquel on souhaite l'appliquer, c'est au chef de projet de choisir le modèle le plus adéquat au problème et contraintes

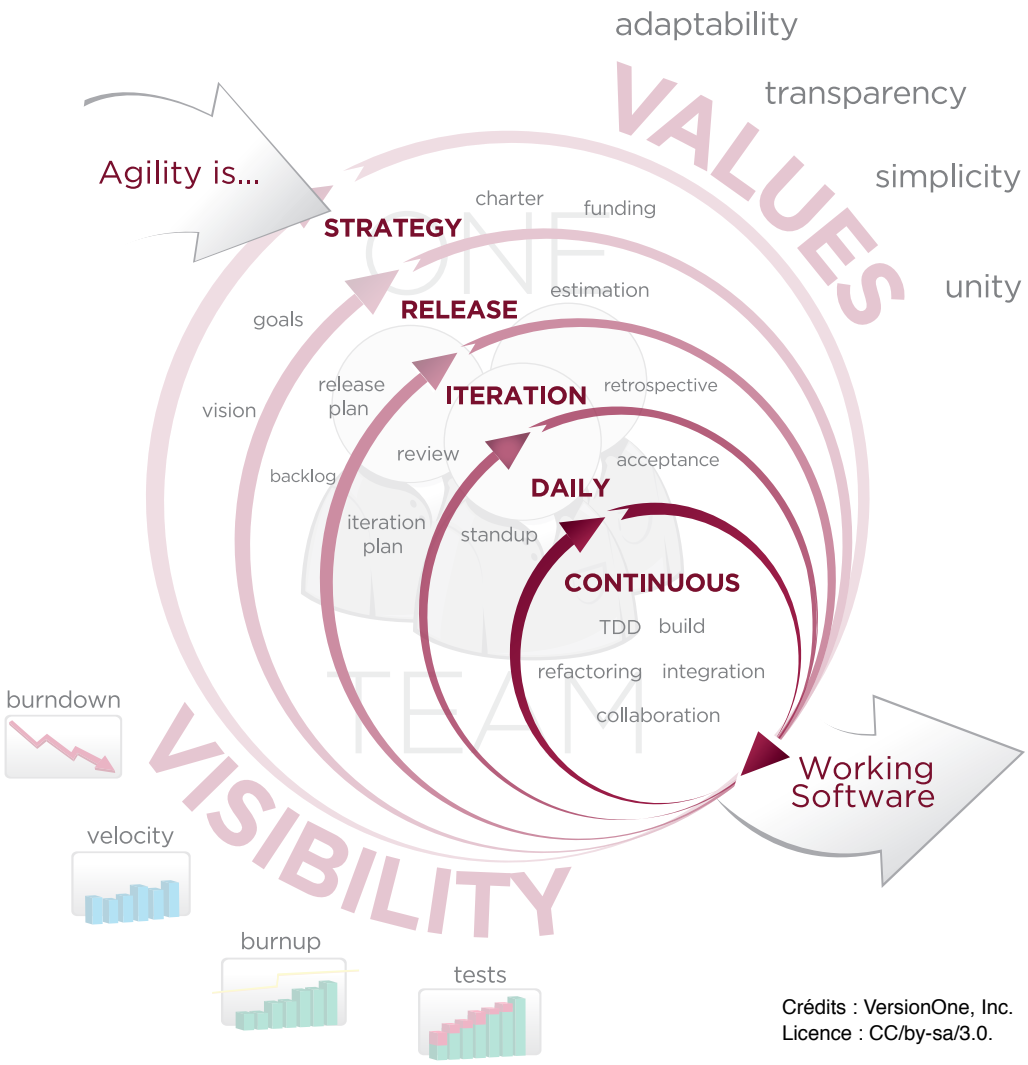


FIGURE II.4 – Une représentation du modèle de développement logiciel « Agile »

qui lui sont proposés.

Néanmoins il est toujours possible d'adopter un modèle et d'y ajouter des contraintes pour le rendre plus adéquat au problème. Par exemple, on peut adopter le cycle Agile et le renforcer au niveau de la traçabilité et de la documentation.

II.4.3 L'importance du processus de développement pour un logiciel critique

Partant du constat qu'il est impossible, sinon extrêmement difficile, de compter exactement le nombre d'erreurs existant dans un logiciel [7], DO-178B s'appuie sur le processus de développement.

Gérard Ladier [48] donne l'analogie selon laquelle « on ne délivre pas d'eau propre avec un tuyau sale. » On peut bien comprendre qu'avec des processus de développement non maîtrisés, il est difficile de faire valoir sa légitimité face aux autorités de certification.

7. Les résultats d'indécidabilité de la logique nous disent même que ce but est impossible à atteindre : il n'y a pas de méthode systématique ou d'algorithme permettant de savoir si une propriété est vérifiée ou non.

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value :

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	Mike Beedle	Arie van Bennekum
Alistair Cockburn	Ward Cunningham	Martin Fowler
James Grenning	Jim Highsmith	Andrew Hunt
Ron Jeffries	Jon Kern	Brian Marick
Robert C. Martin	Steve Mellor	Ken Schwaber
	Jeff Sutherland	Dave Thomas

FIGURE II.5 – *Manifesto for Agile Software Development*

Ainsi, pour obtenir un bon produit, on peut résumer les fondements de la DO-178B avec les cinq principes suivants :

- Le processus de génie logiciel est bien défini.
- Tout doit toujours être vérifié.
- Le respect des objectifs est assuré par une autorité indépendante.
- Les normes doivent être acceptées par tous.
- Les entreprises sont responsables des moyens.

II.5 Documentation et traçabilité pour les logiciels critiques

Dans le processus de développement d'un logiciel critique, l'ensemble du processus et du travail produit doit être passé en revue. Leur **documentation** est donc indispensable. Il existe plusieurs sortes de documentation (informative, descriptive, analytique).

La revue réduit la quantité de travail car on sait à tout moment quelles tâches ont été accomplies, ce qui évite de faire plusieurs fois le même travail par manque de suivi. Il existe des outils pour aider à la revue de projet (*e.g.*, RelyCheck [8]).

La traçabilité est la possibilité de connaître à tout moment du développement les composants d'un produit et leurs provenances tout au long de sa chaîne de production et de distribution.

8. http://relycheck.com/RelyCHECK_Brochure.pdf

Les différents processus de développement logiciel accordent une importance différente à la traçabilité, puisque chaque modèle de développement est un ensemble de compromis qui mettra en avant tels ou tels autres avantages.

La traçabilité peut être une obligation. Pour les développements de logiciels critiques, notamment dans le cadre d'un développement DO-178B, la traçabilité est obligatoire.

II.6 Processus de certification (ou de qualification)

Le processus de certification (ou de qualification [44] [45], c'est une question de vocabulaire qui varie selon le domaine et l'objet concerné) d'un logiciel critique consiste en pratique à rassembler les responsables du projet et les représentants des autorités de certification pendant plusieurs jours afin de passer en revue toute la documentation de l'ensemble du processus, pour **vérifier que tout a bien été respecté**.

Cela implique donc que l'ensemble du processus de développement est directement impacté par le processus de certification, puisque s'il ne l'a pas pris en compte, le résultat ne peut pas passer la certification.

> Kornecki et Zalewski [47] présentent un état de l'art en 2009 du processus de certification appliqué aux différents objets logiciels (*e.g.*, générateur de code, outil de vérification du code, logiciel critique embarqué) intervenant dans le cadre d'un développement de logiciels pour l'avionique civile.

On remarque que l'autorité de certification ne lit pas le code du logiciel ou du moins pas exhaustivement. Elle regarde surtout comment il a été produit et comment il a été vérifié. La vérification de la traçabilité entre le code et la conception détaillée par l'autorité pourra impliquer de prendre du code et vérifier sa provenance, ainsi que dans l'autre sens de prendre une exigence et vérifier son implantation.

Cela ne veut pas dire que personne n'a relu le code. Par exemple, si des règles de codage (*e.g.*, interdiction d'allocation dynamique) interviennent dans le processus de développement, un document doit attester que ces règles ont été respectées.

> **À propos de la qualification des outils selon la DO-178B [73][44][45]**
> La qualification d'un outil utilisé au sein d'un développement DO-178B est le processus nécessaire pour obtenir le crédit de certification pour un outil logiciel **dans le contexte d'un système avionique spécifique**.
> Le crédit de certification est l'acceptation par l'autorité de certification qu'un processus, produit ou démonstration satisfait les exigences d'une certification.
> Cela signifie qu'un outil doit être qualifié à chaque fois qu'il est utilisé pour un nouveau système. Bien entendu, si un outil a déjà été qualifié une fois, il pourra en toute probabilité être plus facilement qualifié une nouvelle fois. On parle « d'outil qualifiable » pour désigner un outil qui a été conçu dans l'optique d'être qualifié plusieurs fois.

II.7 Activités de tests

Définir et exécuter une campagne de tests sont une étape obligatoire du processus de certification. Mener une campagne de tests consiste pour un composant logiciel à définir un ensemble de valeurs d'entrées pour ce composant, et les résultats attendus. La seconde phase de la campagne consiste à exécuter le composant sur les entrées définies et recueillir les résultats obtenus. L'ultime étape consiste à comparer les résultats attendus avec les résultats obtenus et en tirer les conséquences.

Le but de cette activité est donc d'identifier les divergences de comportement du composant entre ce qui était attendu de sa spécification et ce qu'il exécute effectivement. Si des divergences apparaissent, le développement est repris et la campagne est itérée une nouvelle fois.

Ainsi on s'assure de la sûreté de fonctionnement du composant logiciel si l'ensemble des tests prévus couvre bien l'ensemble des comportements possibles du programme.

Les activités de tests sont difficiles car pour que les tests soient pertinents, leur conception ne doit pas être guidée par l'idée de montrer que « ça marche » mais plutôt par celle de mettre en évidence les erreurs.

Pour cela, d'une part, il faut comprendre les spécifications, et d'autre part, il faut imaginer des tests permettant de mettre en évidence les erreurs.

Le test des triangles de Myers [62] permet de savoir si on a l'âme d'un testeur (si on ne l'a pas, cela peut s'acquérir). Voici l'énoncé des spécifications d'un programme :

> *The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.*

Myers donne alors une liste de 14 types de tests et accorde un point par type de test trouvé. Par exemple, un type de test est un triangle quelconque valide, un autre est une valeur négative parmi les trois, encore un autre est un ensemble de trois valeurs où la somme de deux des valeurs est inférieure au troisième. Myers observe que les programmeurs expérimentés parviennent à fournir un jeu de tests couvrant en moyenne 7,8 des 14 cas de tests, soit environ 56% des cas de tests.

En fait, ces spécifications sont assez incomplètes, ce que montrent les 14 types de tests de Myers. Par exemple, il n'est pas précisé que faire lorsqu'il y a une erreur sur les paramètres d'entrée.

Une erreur est de faire l'hypothèse que tous les triangles non équilatéraux et non isocèles sont scalènes (quelconques). En effet, un « triangle » dont les côtés ne se rejoignent pas n'est pas un triangle, c'est donc une erreur de répondre qu'il est scalène.

Répondre exactement au texte de spécification peut être une erreur. En effet, si on rend des résultats faux, ou si on rend un résultat non spécifié lorsque les entrées sont erronées, on peut risquer d'engendrer une faille de sécurité dans le programme.

Un concept sous-jacent à l'exemple choisi par Myers est celui du « taux de couverture » obtenu par l'exécution des tests.

Un bon jeu de tests doit avoir prévu tous les cas.

- fonctionnement normal du système y compris aux limites (nominal) ;
- réaction aux données anormales (hors limites, robustesse).

Tous les cas définissent la couverture fonctionnelle d'un composant. Le ratio entre le nombre de comportements observés et le nombre de tous les cas possibles donne un taux de couverture. Un bon jeu de tests donne donc un taux de 100%. Toutefois définir avec précision l'espace fonctionnel à couvrir avec un ensemble de tests est impossible (*cf.* remarque en bas de page 28).

Une méthode complémentaire permettant de s'assurer de la qualité d'un programme est de considérer l'ensemble des instructions qui le composent. Si par un jeu de tests, on peut activer l'ensemble de toutes ces instructions et observer que l'ensemble des résultats obtenus satisfait l'ensemble des résultats attendus, on obtient également un critère de sûreté de fonctionnement du programme. Et cette fois on a également un moyen précis de mesurer le taux de couverture assuré par les tests : nombre d'instructions activées par rapport au nombre fini d'instructions du programme. C'est ce qu'on appelle le taux de couverture structurelle d'un programme.

II.8 Évolution des activités de tests

II.8.1 Évolution des coûts

Selon Myers en 1979 [62], le temps passé dans les activités de tests pour un logiciel était d'environ la moitié du temps total de son développement, pour un coût dépassant un peu la moitié du coût total. Environ trois décennies plus tard, Myers *et al.* ont confirmé ces chiffres à deux reprises, en 2004 [64] et 2011 [63].

II.8.2 Évolution des techniques

Pourtant ces 30 dernières années, les techniques de tests ont évolué, permettant ainsi la réduction des coûts des tests. C'est le cas aussi pour ce qui concerne les environnements de développement de manière générale.

Mais dans le même temps, les logiciels sont devenus beaucoup plus complexes, ce qui explique en bonne partie que les coûts des activités de tests ont gardé la même proportion.

D'autre part, les auteurs de livres sur l'activité de tests affirment tous qu'il y a très peu de formations aux tests dans les écoles et universités.

II.9 Couverture de code

On entend ici par couverture de code ou mesure de couverture de code ce que nous avions décrit en section II.7 sous le nom de couverture structurelle.

II.9.1 Objectifs

Diverses caractéristiques suspectes peuvent être révélées par cette mesure, comme par exemple du code mort [9], du code désactivé [10] ou du code mettant en évidence l'absence de certains tests ou le non respect d'une exigence.

9. Selon la DO-178B, le code mort est du code inatteignable (quelque soit les conditions d'exécution) et qui ne correspond pas à une exigence de plus haut niveau (donc sa traçabilité ne peut pas être établie). Par exemple, l'expression $\texttt{e}_1$ est inatteignable dans l'expression suivante : **`if false then`** $\texttt{e}_1$ **`else`** $\texttt{e}_2$.

10. Selon la DO-178B, le code désactivé est du code non atteint lors des tests, mais dont la présence est justifiée par des exigences de plus haut-niveau selon la traçabilité.

> **À propos de la pratique de mesure de couverture de code structurelle**
> La mesure de couverture de code structurelle se fait généralement à travers la mesure de la couverture fonctionnelle, qui consiste à vérifier que les comportements du programme correspondent bien aux spécifications. Ainsi, les tests permettant d'établir la couverture structurelle ne doivent pas être des tests spécifiques à cette mesure.
> C'est d'une part un gage de qualité dans le processus de vérification, et d'autre part une difficulté.

La couverture pour la DO-178B

La DO-178B demande le respect de l'ensemble des tests requis par le plan de tests. Un rapport de couverture (fonctionnelle ou structurelle) montre quels tests ont été satisfaits et lesquels ont révélé des erreurs ou autres problèmes.

Si certains tests ont révélé des problèmes, il faut soit retourner à une étape précédente du développement, soit fournir une documentation justifiant que l'existence de cet état ne met pas en cause la sûreté du système.

II.9.2 Outils pour la couverture de code

Pour mesurer la couverture de code, l'utilisation d'outils est devenue indispensable du fait de la taille et de la complexité des programmes développés de nos jours. De nombreux outils existent pour les langages les plus utilisés dans le monde des logiciels critiques, que sont le langage C [36] [65] et le sous-ensemble SPARK [2] du langage Ada.

Le travail mené au cours de cette thèse a été précisément de concevoir et réaliser un outil permettant l'utilisation d'une autre sorte de langage, précisément OCaml, dans le processus de développement d'un logiciel critique pour l'avionique civile. Ce travail prend son origine dans une collaboration avec la société Esterel Technologies que nous présentons brièvement à la section suivante.

Expérience avec OCaml

Dans cette section, on résume l'expérience de l'entreprise Esterel Technologies avec OCaml utilisé en tant que langage d'implantation du générateur de code de SCADE, présentée dans les articles [68][66][67]. Ce travail avait été effectué en partenariat avec les laboratoires PPS et LIP6.

L'entreprise Esterel Technologies développe et commercialise SCADE, un environnement de développement d'applications critiques. Les applications sont développées en langage Lustre puis traduites automatiquement en langage C. (Quelques références sur le langage Lustre : [35] [34] [12].) La suite SCADE offre la possibilité de programmer en langage graphique Scade (à la place d'écrire du code Lustre). Le processus de traduction du Scade vers C est réalisé par le générateur de code appelé KCG [11].

Pour passer à la version 6 de SCADE, Esterel Technologies a développé en OCaml un prototype de KCG [30]. La société a choisi de garder le même langage d'implantation pour passer du prototype à la nouvelle version de KCG. Le langage choisi pour développer le prototype (OCaml) facilite la traçabilité entre la spécification du générateur de code et son implantation. Cela tient au fait qu'un générateur de code est un programme de manipulation d'arbres de syntaxe abstraite, et que les langages de la famille ML, dont fait partie OCaml, offre des facilités pour ce genre de traitements (type somme, filtrage par motifs).

11. `http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation`

Toutefois, ce choix a eu un coût. Comme ce logiciel produit du code destiné à être embarqué dans des systèmes critiques, il faut soit appliquer le processus de certification au niveau du code produit, ce qui est assez difficile (car c'est du code généré et que c'est très gros), soit appliquer le processus de certification au générateur de code, avec les mêmes contraintes que si c'était pour le code généré lui-même. *De plus, comme le soulignent Beine et Jungmann [4], la génération de code est une activité récente dans le monde des logiciels critiques.*

Là s'est posé un problème majeur. En effet, les langages multiparadigmes de haut-niveau comme OCaml ne sont pas très présents dans le monde des logiciels critiques. Il n'y avait donc pas d'outil existant pour OCaml et il n'y avait pas non plus de langage proche d'OCaml pour lequel les outils existaient. Il a donc fallu, entre autres, développer un outil de couverture de code pour le langage OCaml.

C'est ainsi qu'en 2006, MLcov a été prototypé au laboratoire PPS. Ensuite, il a été développé en 2006/2007 par Esterel Technologies. MLcov est un outil de couverture de code pour les programmes développés en OCaml.

La force du produit SCADE est qu'il peut être utilisé dans un processus de qualification DO-178B au niveau A. Notamment, Airbus emploie SCADE pour une partie de ses développements [27] ; une version de SCADE a notamment été utilisée par Airbus pour développer des logiciels critiques embarqués dans les avions A340 et A380 [12].

II.9.3 Un exemple de couverture de code

On donne ici une spécification d'un programme. On propose ensuite une implantation. Puis on donne un rapport de tests fonctionnels (un peu simplifié) appliqué au couple « spécification, implantation » et cet ensemble nous servira d'exemple tout au long de ce manuscrit.

Spécifications Fournir une bibliothèque qui contient une fonction `is_scalene` qui prend trois entiers, et rend **`true`** si, et seulement si, les trois entiers sont différents et peuvent être les longueurs des côtés d'un triangle.

Dans les autres cas, la fonction doit rendre **`false`**. Notamment, la fonction doit rendre **`false`** lorsque les trois entiers ne peuvent pas représenter un triangle.

Les triangles rectangles (*i.e.*, les triangles ayant un angle droit) n'ont pas besoin de traitement particulier, conformément à la définition de « scalène. »

En français, « Le Trésor de la Langue Française informatisé » dit :

> **scalène**, adj. et subst. masc.
> *GÉOM.* (Triangle) dont les trois côtés sont inégaux.

En anglais, « *New Oxford American Dictionary* » dit :

> **scalene**
> *adjective*
> (of a triangle) having sides unequal in length.
>
> *noun*
> a scalene triangle.

12. `http://web.archive.org/web/20110525193749/http://www.esterel-technologies.com/technology/success-stories/airbus`

Addenda : précisions sur les spécifications Selon les définitions, rien n'empêche un triangle scalène d'être plat, c'est-à-dire d'avoir ses trois sommets alignés. En revanche, on ne peut pas avoir un côté de longueur 0, car les deux autres côtés seraient obligatoirement égaux. Cependant, pour ne pas inutilement compliquer notre exemple, on considérera que la fonction `is_scalene` doit rendre **`false`** lorsque ses trois entrées forment un triangle plat.

Nous pouvons aussi remarquer que le domaine de définition donné dans les spécifications n'est en fait pas précisé. En effet, les valeurs maximales admissibles pour les entrées ne sont pas spécifiées. Toujours dans l'optique de préserver la simplicité de notre exemple, on émettra l'hypothèse que la somme des trois entrées est toujours inférieure à la valeur maximale admissible des entiers.

Implantation en OCaml

Voici un premier code avec une erreur introduite expressément, pour illustrer plus loin la détection d'erreurs via les tests fonctionnels. *(L'erreur est révélée après le code de la seconde version.)* On donne la version en langage C à droite de la version en OCaml.

```
(* is_scalene en OCaml *)
let all_different = fun a b c ->
  a <> b && b <> a && c <> a

let all_positive = fun a b c ->
  a > 0 && b > 0 && c > 0

let is_triangle = fun a b c ->
  all_positive a b c
  && a + b > c
  && a + c > b
  && b + c > a

let is_scalene = fun  a b c ->
  is_triangle a b c
  && all_different a b c
```

```
/* is_scalene en C */
int all_different (int a, int b, int c) {
  return (a != b && b != a && c != a);
}
int all_positive (int a, int b, int c) {
  return (a > 0 && b > 0 && c > 0);
}
int is_triangle (int a, int b, int c) {
  return (all_positive (a, b, c)
         && a + b > c
         && a + c > b
         && b + c > a);
}
int is_scalene (int a, int b, int c) {
   return (is_triangle (a, b, c)
         && all_different (a, b, c));
}
```

Voici un second code sans l'erreur qui avait été introduite expressément.

```
let all_different = fun a b c ->
  a <> b && b <> c && c <> a

let all_positive = fun a b c ->
  a > 0 && b > 0 && c > 0

let is_triangle = fun a b c ->
  all_positive a b c
  && a + b > c
  && a + c > b
  && b + c > a

let is_scalene = fun a b c ->
  is_triangle a b c
  && all_different a b c
```

L'erreur a été insérée à la ligne 2 et ne consiste à changer qu'un seul caractère.

En code ASCII [13], il n'y a qu'un bit qui a changé, le caractère « a » étant codé par `0110000`**`0`**`1` tandis que le caractère « c » est codé par `0110001`**`1`**.

Cela soutient le fait que trouver les erreurs qui se glissent dans les programmes est une activité particulièrement difficile.

Un « bon » jeu de tests fonctionnels doit mettre l'erreur en évidence.

Tests fonctionnels La table II.1 (page 37) rapporte l'exécution d'un jeu de tests fonctionnels, où figurent les entrées, la sortie attendue, le résultat obtenu, ainsi que les commentaires sur la nature de chaque test.

On omet volontairement le « test aux limites » car il ne nous apporte pas d'informations de couverture de code structurelle ici. Tester `is_scalene` aux limites aurait consisté à utiliser des entrées risquant de faire basculer les entiers positifs dans le domaine négatif et vice versa, du fait de l'encodage des entiers sur des machines de calculs. On remarque que les tests avec des entrées négatives sont des tests hors limites.

Le point d'entrée est la fonction `is_scalene` qui attend trois paramètres. Les autres fonctions ne sont pas visibles depuis l'environnement des tests fonctionnels. *Remarque : on aurait pu fournir l'ensemble de l'implantation dans une seule fonction, et il aurait fallu que le même jeu de tests fonctionne pour les deux versions.*

Détection d'erreurs Avec la version erronée du programme, le test numéro 7 (table II.1, page 37) montre que le résultat obtenu ne correspond pas à la sortie attendue.

Couverture structurelle La couverture de code structurelle obtenue par les tests fonctionnels (table II.1) est complète :

- chaque expression a été évaluée (couverture complète au sens des expressions) ;
- chacune des conditions a joué un rôle déterminant dans les résultats (couverture complète au sens MC/DC).

Les chapitres III (page 39) et IV (page 59) définiront précisément ces notions de couverture.

L'important à ce stade est de remarquer que si aucun des tests rendant **`false`** n'avait été fait, aucune des valeurs booléennes du programme n'aurait été évaluée à **`false`**. On n'aurait pas obtenu la satisfaction de la couverture des conditions (qui demande à ce que chaque condition ait été évaluée aux deux valeurs booléennes.) Et si on avait testé la version du programme avec erreur, l'erreur n'aurait pas été détectée.

On peut noter que l'introduction volontaire d'erreurs dans les programmes fait partie de certaines activités de tests : on teste alors le jeu de tests, pour voir s'il permet effectivement de trouver les erreurs introduites expressément.

13. ASCII est l'abréviation de « *American Standard Code for Information Interchange* ». C'est la norme de codage de caractères en informatique la plus connue. C'est également la plus largement prise en charge, surtout que de nombreux autres codages (*e.g.*, ISO/CEI 8859, Unicode) sont des extensions de celle-ci.

#	entrées (a, b et c)			sortie attendue	résultat obtenu	commentaire
	a	b	c			
Triangle valide						
1	3	2	4	true	true	b < a < c
2	1238	2389	2839	true	true	a < b < c
3	23982389	19828390	12293381	true	true	c < b < a
4	9820	9932	8293	true	true	c < a < b
Triangle isocèle						
5	10	10	4	false	false	a = b
6	42	8	42	false	false	a = c
7	55	54	54	false	true	b = c
Triangle équilatéral						
8	89	89	89	false	false	a = b = c
Côtés trop courts (pas un triangle)						
9	1	1	2	false	false	a + b = c
10	111	222	111	false	false	a + c = b
11	2	4	1	false	false	b > a + c
12	32	3	1	false	false	a > b + c
Valeurs négatives (pas un triangle)						
13	-1	3	3	false	false	a < 0
14	4	-4190	4293	false	false	b < 0
15	34289	12833	-92238	false	false	c < 0
16	-3493	-3583	-3324	false	false	tous < 0
Zéro (pas un triangle)						
17	0	23	21	false	false	a = 0
18	7	7	0	false	false	c = 0
19	0	0	0	false	false	tous = 0
20	29	0	0	false	false	b = c = 0
21	0	0	2	false	false	a = b = 0
22	0	23	0	false	false	a = c = 0

Table II.1 – Un jeu de tests fonctionnels (sans tests aux limites des entiers)

II.10 Conclusion du chapitre

Nous nous plaçons donc dans ce monde de logiciels critiques où les outils de couverture de code semblent encore indispensables pour quelques années à venir. Nous proposons une étude formelle de l'introduction d'un langage applicatif de haut-niveau avec son outil de couverture de code dans ce domaine. Cette étude prend en considération les problématiques et exigences de la DO-178B et en particulier l'exigeant critère de mesure de couverture MC/DC.

CHAPITRE III

Le Langage mCaml et la couverture structurelle des expressions

« *Well-typed programs cannot "go wrong".* »
– Robin Milner [59]

Sommaire

III.1 Le langage mCaml 42
III.1.1 Grammaire informelle des expressions du langage mCaml 43
III.1.2 Sémantique opérationnelle du langage mCaml 44
III.1.3 Arbre d'évaluation 52
III.2 Spécification de la couverture des expressions pour mCaml 53
III.2.1 Règles formelles de couverture des expressions pour mCaml 54
III.2.2 Taux de couverture 57
III.3 Conclusion du chapitre 57

Résumé du chapitre 3

Ce chapitre présente d'abord de manière informelle la grammaire du langage mCaml qui nous sert de base pour le reste de la thèse. Ensuite, on définit formellement sa sémantique opérationnelle. La grammaire formelle complète ainsi que le système de types complet sont donnés en annexe A, page 157.

Enfin, la définition formelle de la couverture des expressions est donnée pour ce langage.

Le langage de programmation est un élément important lors de la conception d'un logiciel. Par exemple, si le langage ne permet pas d'exprimer des constructions linguistiques très enclines aux erreurs, alors on restreint les erreurs possibles.

Les langages de la famille ML offrent un compromis entre la pureté du λ-calcul et les effets de bord. Ils apportent également une rigueur due au système de vérification des types, permettant d'évincer de nombreuses constructions à risque tout en laissant une assez grande liberté. Ses principaux représentants actuels sont SML, OCaml [1] [52], et F# [69].

Résumé de la genèse de ML ML prend naissance au début des années 1970 au sein du projet LCF[58]. Divers dialectes de ML ont été développés au début des années 1980, ce qui a motivé l'établissement d'un standard pour le noyau commun à ces divers dialectes. Milner a alors publié en 1990 une définition formelle de Standard ML (ou SML)[60]. Ainsi, SML ou SML'90 est né avec les règles de son système de types et sa sémantique opérationnelle. Une révision a été publiée en 1997 [61], délaissant les traits les moins utiles du langage dans le but de le simplifier. Deux des implantations les plus diffusées de nos jours sont SML/NJ (Standard ML of New Jersey) [78] et MLton[28].

Caml est le dialecte français de ML, dont la première implantation date de 1987. Il est basé sur la machine abstraite catégorique (CAM) [20], ce qui explique en partie son nom, et elle-même basée sur la machine abstraite LLM3 et la gestion mémoire du langage Le_Lisp. Caml est alors conçu pour l'implantation de logiciels comme Coq [81]. Au tout début des années 1990, une nouvelle implantation de Caml plus légère voit le jour sous le nom de Caml light [85]. En 1996, OCaml succède à Caml Light en ajoutant notamment un système de modules plus expressif ainsi que le paradigme de programmation par objet décrit par Rémy et Vouillon [76] [77].

F# est développé par l'entreprise Microsoft et reprend la syntaxe d'OCaml pour le noyau impératif et fonctionnel (ce qui permet la compatibilité pour une certaine classe de programmes), en ajoutant la couche objet de C#. La combinaison de caractéristiques venant de langages éloignés forme un langage ayant des traits bien spécifiques.

Notre choix se porte sur OCaml (en fait un sous-ensemble du langage) parce qu'il s'agit d'un langage de haut-niveau, qui a déjà été utilisé dans le cadre d'un développement logiciel soumis aux contraintes de la DO-178B [67]. Il est également le langage d'implantation d'outils qui aident à améliorer la sûreté des programmes, comme par exemple Astrée [22] qui a permis de détecter le bug de la fusée Ariane V, Frama-C [24] qui est un analyseur statique pour les programmes développés en langage C, FoCaLiZe [37] qui est un environnement de développement pour programmes formellement prouvés qui génère du code Coq [81] pour les preuves et du code OCaml pour produire des exécutables. Coq est un assistant aux preuves également implanté en OCaml.

Cependant, les travaux présentés ne sont pas spécifiques ou restreints à OCaml ou à ML, en ce sens que les langages applicatifs de manière générale peuvent profiter des idées directrices de cette thèse.

Plan du chapitre

Nous proposons en premier lieu dans ce chapitre une présentation du langage mCaml, assez petit pour être spécifiable formellement et assez riche pour prétendre être proche d'un langage utili-

1. OCaml s'appelait « Objective Caml » jusqu'à son renommage décidé en 2011. `https://sympa-roc.inria.fr/wws/arc/caml-list/2011-07/msg00135.html`

sable dans la pratique. Cela diffère des habitudes qui consistent à prendre un noyau minimaliste pour les spécifications formelles et les raisonnements qui les accompagnent.

La présence d'une spécification formelle pour l'ensemble du langage est nécessaire pour ne laisser aucune place aux imprécisions sur le langage qui nous sert de support pendant tout le reste de la thèse.

Notre langage applicatif inclut des traits impératifs (structures de données mutables, gestion des exceptions), le filtrage par motifs [1] [49] et les types définis par l'utilisateur (type somme et type produit). mCaml est effectivement plus riche (en constructions) et plus expressif qu'un mini-ML « classique » (*e.g.*, celui de Clément *et al.* [15]).

Toutefois, mCaml n'apporte aucun trait nouveau par rapport à la littérature (Pierce [70], Pottier et Rémy [71]). De fait, nous omettrons les preuves sur les propriétés du langage. Par exemple, la propriété bien connue des langages ML « *Well-typed programs cannot "go wrong"* » (*i.e.*, les programmes bien typés ne peuvent pas avoir d'erreur de type à l'exécution) [59] n'est pas démontrée dans ce chapitre mais sera admise.

Les langages ML offrent généralement une grande liberté de choix de style de programmation au programmeur, qui doit définir lui-même ses règles de codage et les degrés d'abstraction qu'il veut pratiquer. Par exemple, OCaml permet la programmation impérative, fonctionnelle mais aussi par objets, avec un système de modules paramétrés. Contrairement à des langages qui visent le *motto* « *only one way to express a given task* » [2], en ML il existera en général de multiples styles pour implanter une spécification.

mCaml est un sous-ensemble du langage OCaml. De ce dernier, nous extrayons les constructions linguistiques les plus communément utilisées, ce qui donne un langage statiquement typé, fonctionnel et impératif, avec le filtrage par motifs qui est l'une des grandes forces de la famille des langages ML.

Ainsi, nous avons pour mCaml les définitions de types algébriques sommes et produits, les définitions globales et locales, les boucles itératives (**`while`** et **`for`**), la conditionnelle (**`if`**), les fermetures éventuellement récursives, la séquence, l'affectation, le filtrage par motifs, les nombres et les booléens. On a aussi la valeur `()` de type **`unit`**, qui est notamment utilisée comme valeur de retour par les boucles et la séquence [3]. Les opérateurs booléens séquentiels de conjonction et de disjonction sont également spécifiés et sont considérés comme des formes particulières du langage. Le lecteur qui connaît assez bien OCaml peut directement aller à la seconde partie du chapitre page 53.

En seconde partie de chapitre, nous présentons une définition formelle de la couverture des expressions pour le langage mCaml. Cela permettra de ne laisser aucune place à l'ambiguïté, encore une fois, car **ces définitions sont la base des chapitres suivants**. Nous verrons déjà qu'utiliser un langage d'expressions aussi riche (car mCaml est riche par rapport aux langages habituellement utilisés dans les domaines des logiciels critiques) introduit des problèmes spécifiques.

III.1 Le langage mCaml

Notre langage est entièrement défini formellement mais pour alléger sa présentation, celle-ci est découpée en deux. Cette section présente une grammaire simplifiée des expressions du langage et

2. James Gosling et Henry McGilton écrivent sur Java en 1996 : « *It's not to say that functions and procedures are inherently wrong. But given classes and methods, we're now down to* ***only one way to express a given task****. By eliminating functions, your job as a programmer is immensely simplified : you work only with classes and their methods.* » [32, section 2.2.4]

3. Dans le système de types mCaml, toute expression est associée à un type, et toute expression rend une valeur ou bien lève une exception. Les boucles étant des expressions, il leur faut bien rendre une valeur. C'est la valeur du type singleton **`unit`** qui est utilisée.

donne la sémantique opérationnelle des définitions globales et des expressions du langage.

La sémantique opérationnelle n'est définie que pour les programmes acceptés par le système de types qui est défini en annexe (section A.2, page 161).

En annexe, se trouve également la grammaire formelle et complète du langage en section A.1 (page 157).

Nous insistons sur la nécessité de la présence d'une définition formelle, d'autant plus que nous proposons un langage peu conventionnel notamment dans l'optique de l'utiliser dans un cadre de développement de logiciels critiques, comme pourraient le rappeler Halang et Zalewski [33].

III.1.1 Grammaire informelle des expressions du langage mCaml

Nous donnons ici une grammaire sous forme intuitive dans le but de permettre au lecteur de se faire rapidement une idée du langage que nous définissons.

La figure III.1 résume les expressions du langage mCaml, où `x` désigne une variable, `c` désigne une constante, `e` désigne une expression, `f` désigne un champ d'enregistrement, `C` désigne un constructeur constant (*i.e.*, valeur d'un type énuméré), `C(e)` est un constructeur ayant un argument (*i.e.*, valeur d'un type énuméré associée à une autre valeur) et `p` désigne un motif servant au filtrage. Les mots-clefs du langage sont en gras.

e ::=	`x`	variable
|	`c`	constant
|	`C`	constant constructor
|	`C(e)`	constructor with an argument
|	`e e`	application
|	**`fun`** `x -> e`	abstraction
|	**`if`** `e` **`then`** `e` **`else`** `e`	conditional
|	**`let`** `x = e` **`in`** `e`	let-in
|	**`let rec`** `x = fun x -> e` **`in`** `e`	rec-in
|	`e ; e`	sequence
|	**`{`** `f=e ; ... ; f=e` **`}`**	record
|	`e.f`	access
|	`e.f <- e`	modify
|	**`for`** `x = e` **`to`** `e` **`do`** `e` **`done`**	for
|	**`while`** `e` **`do`** `e` **`done`**	while
|	`e` **`&&`** `e`	sequential conjunction
|	`e || e`	sequential disjunction
|	**`not`** `e`	negation
|	`(e)`	parentheses
|	**`match`** `e` **`with`** `p -> e | ... | p -> e`	match
|	**`raise`** `e`	raise
|	**`try`** `e` **`with`** `x -> e`	try

Figure III.1 – Grammaire simplifiée des expressions du langage mCaml

La figure III.2 présente une extension syntaxique pour le langage mCaml, qui n'a que pour rôle d'alléger l'écriture des exemples de programmes dans les chapitres suivants. Par exemple, on pourra écrire

```
let compose f g x =
 f (g x)
```

plutôt que

```
let compose = fun f -> fun g -> fun x ->
 f (g x)
```

Cette extension montre le schéma de réécriture pour transformer les expressions utilisant l'extension de langage en expression du langage mCaml sans extension.

Construction ajoutée		*Équivalence en* mCaml
`fun` x_1 `...` x_n `->` e	$\rightsquigarrow$	`fun` x_1 `->` `...` `fun` x_n `->` e
`(` e_1 e_2 `...` e_n `)`	$\rightsquigarrow$	`((` e_1 e_2 `)` `...` e_n `)`
`let` x_1 x_2 `...` x_n `=` e	$\rightsquigarrow$	`let` x_1 `=` `fun` x_2 `...` x_n `->` e
`let rec` x_1 x_2 `...` x_n `=` e	$\rightsquigarrow$	`let rec` x_1 `=` `fun` x_2 `...` x_n `->` e

`try` e `with` `|` p_1 `->` e_1 `|` `...` `|` p_n `->` e_n	$\rightsquigarrow$	`try` e `with` x `->`
		`(` `match` x `with`
		`|` p_1 `->` e_1 `|` `...` `|` p_n `->` e_n
		`|` x `->` `raise` x `)`

Figure III.2 – Sucre syntaxique pour le langage mCaml

III.1.2 Sémantique opérationnelle du langage mCaml

Cette section décrit comment un programme écrit en mCaml s'évalue. Nous utilisons la sémantique opérationnelle à grands pas. mCaml, à l'instar d'OCaml, utilise la convention d'appels par valeurs, c'est-à-dire que lors d'une application, l'argument d'une fonction est évalué avant le corps de la fonction.

En OCaml, il n'est pas spécifié l'ordre d'évaluation des paramètres des fonctions. Ainsi, dans l'évaluation de l'expression `(foo (raise A) (raise B))`, il est difficile de savoir si c'est l'exception `A` ou l'exception `B` qui sera levée. Cette sémantique laisse la liberté au compilateur (ou à son concepteur) de décider qui évaluer en premier pour des questions d'optimisations. En mCaml, il n'est pas question d'optimisations du compilateur. La sémantique de mCaml ne laisse aucun choix ouvert.

III.1.2.1 Notations

Pour décrire formellement la sémantique opérationnelle du langage mCaml, nous utilisons les notations suivantes.

• **Les constantes** sont notées c, c_0, c_1, c_2, c_n, etc.

• **Les valeurs** sont notées v, v_0, v_1, v_2, v_n, etc.

• **Les nombres** (entiers relatifs) sont notés n, n_0, n_1, n_2, n_n, etc., quand ils sont inconnus, ou leur représentation décimale lorsque c'est possible. Ils sont de type **`int`**.

• **Les booléens** sont notés **`true`** ou bien **`false`**, ils sont de type **`bool`**.

• **Les variables** sont notées x, x_0, x_1, x_2, x_n, etc.

• **Les expressions du langage** sont notées e, e_0, e_1, e_2, e_n, etc.
• **Les champs d'enregistrements** sont notés f, f_0, f_1, f_2, f_n, etc.
• **Les constructeurs constants** sont notés C, C_0, C_1, C_2, C_n, etc.
• **Les constructeurs non constants** sont notés $C(e)$, $C_0(e)$, $C_1(e)$, $C_2(e)$, $C_n(e)$, etc. Ils pourront également être notés $C(v)$, $C_0(v)$, $C_1(v)$, $C_2(v)$, $C_n(v)$, etc. s'ils sont sous forme irréductible (c'est-à-dire que l'argument est valeur).
• **Fermetures** Les fermetures sont des valeurs et pourront être notées comme les autres valeurs (v, v_0, etc.) mais on aura besoin parfois de destructurer le contenu d'une fermeture.

On note $\left\langle x, e, E \right\rangle$ la fermeture contenant l'environnement E, attendant un argument x pour exécuter le corps e.

Environnements d'évaluation Les expressions sont évaluées dans deux environnements d'évaluation. Le premier est l'environnement lexical, il contient les associations (variable × valeur). Le second est la mémoire et sert à représenter les effets de bord, il contient les associations (adresse × valeur).
• **Les environnements lexicaux** associent des variables à des valeurs. Ils sont notés $\mathcal{E}$, $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{E}_n$, etc.
• **La mémoire** associe des adresses à des valeurs. Elle est notée $\mathcal{M}$, $\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_n$, etc.
• **Enrichissement d'un environnement et accès aux informations**. Si H est un environnement et $H' \triangleq (x,t) \lhd H$, alors H' est un environnement et $H'(x) = t$.

Lorsqu'une association est absente d'un environnement, on rend $\bot$.
• **L'environnement vide** est noté []. Ainsi, si $H = []$, alors $\forall w, H(w) = \bot$.
• **L'environnement lexical initial** contient les opérateurs arithmétiques de base.

$\mathcal{E}$ = [(+,<primitive>) ;(-,<primitive>) ;(*,<primitive>) ;(/,<primitive>) ;]

<primitive> indique qu'il s'agit d'une valeur fournie au langage. Ici les noms devraient être assez éloquents pour se suffire.
• **La mémoire initiale** est vide.

$\mathcal{M}$ = []
• **L'allocation dans la mémoire**. $\mathcal{M} \triangleq (x,v) \lhd \mathcal{M}'$ signifie qu'à l'adresse x de la mémoire $\mathcal{M}$ est associée la valeur v. Comme la mémoire est un environnement, si $\mathcal{M} \triangleq (x,v) \lhd \mathcal{M}'$ alors $\mathcal{M}(x) = v$. En revanche, le système de types nous garantit que $\nexists x. \mathcal{M}(x) = \bot$.
• **L'évaluation d'une expression** rend sa valeur et l'état mémoire après son évaluation, sous la forme $(v \Big/ \mathcal{M})$. Si une exception est levée à la place d'un rendu de valeur, l'exception est rendue à la place de la valeur, ce qui donne un résultat sous la forme (**`Exception`** $v \Big/ \mathcal{M}$).

III.1.2.2 Évaluation des définitions globales et d'un programme

L'évaluation d'un programme ou d'une définition rend un couple d'environnements : l'environnement lexical et la mémoire sous la forme $(\mathcal{E}, \mathcal{M})$. En cas de levée d'exception non rattrapée, `STOP` est rendu à la place du couple.

L'évaluation d'un programme consiste à évaluer une suite de définitions globales. Si $\mathcal{P}$ est un programme et $\mathcal{D}_i$ la i-ème définition globale du programme $\mathcal{P}$, alors nous pouvons énoncer la règle suivante :

$$\textsc{Program}\ \frac{\mathcal{E}, \mathcal{M} \vdash \mathcal{D}_0 \rightsquigarrow \mathcal{E}_0, \mathcal{M}_0 \qquad \ldots \qquad \mathcal{E}_{n-1}, \mathcal{M}_{n-1} \vdash \mathcal{D}_n \rightsquigarrow \mathcal{E}_n, \mathcal{M}_n}{\mathcal{E}, \mathcal{M} \vdash \mathcal{P} \rightsquigarrow \mathcal{E}_n, \mathcal{M}_n}$$

L'évaluation d'une définition globale consiste à évaluer son corps et à rendre l'environnement enrichi de la liaison entre la nouvelle variable et la valeur résultat de l'évaluation du corps de la définition, ainsi que la mémoire qui a pu être modifiée.

$$\textsc{Let}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{let}\ x = e) \rightsquigarrow (x, v) \triangleleft \mathcal{E}, \mathcal{M}_1}$$

La définition récursive globale étant limitée à la définition de fonctions (potentiellement récursives), la définition en elle-même n'entraîne pas de modification de l'état mémoire.

$$\textsc{Rec}\ \frac{\mathcal{E}' \triangleq (x_1, \left\langle x_2, e, \mathcal{E}' \right\rangle) \triangleleft \mathcal{E}}{\mathcal{E}, \mathcal{M} \vdash (\texttt{let rec}\ x_1 = \texttt{fun}\ x_2 \texttt{ -> } e) \rightsquigarrow \mathcal{E}', \mathcal{M}}$$

> • **Les levées exceptions** sont notées **`Exception`** v où v est la valeur de l'exception levée. Le mot **`Exception`** est alors un mot spécial désignant la levée d'exception. On évitera de l'utiliser comme nom d'exception pour éviter la confusion, et il est d'ailleurs écrit en gras.

Pour le cas où une définition s'évalue en une levée d'exception, le programme s'arrête. C'est la seule construction dans le langage qui peut mettre fin à l'exécution d'un programme avant que l'ensemble des définitions n'aient été évaluées. On représente l'arrêt d'un programme par `STOP`.

$$\textsc{Let-STOP}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{let}\ x = e) \rightsquigarrow \texttt{STOP}}$$

$$\textsc{Program-STOP}\ \frac{\mathcal{E}, \mathcal{M} \vdash \mathcal{D}_0 \rightsquigarrow \mathcal{E}_0, \mathcal{M}_0 \qquad \dots \qquad \mathcal{E}_i, \mathcal{M}_i \vdash \mathcal{D}_j \rightsquigarrow \texttt{STOP}}{\mathcal{E}, \mathcal{M} \vdash \mathcal{P} \rightsquigarrow \texttt{STOP}}$$

III.1.2.3 Évaluation des expressions

L'évaluation d'une expression e parenthésée est l'évaluation de l'expression e.

$$\textsc{Parentheses}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{(}e\texttt{)} \rightsquigarrow v \Big| \mathcal{M}_1}$$

L'évaluation d'une variable x rend sa valeur $\mathcal{E}(x)$ quand elle existe dans l'environnement d'évaluation $\mathcal{E}$. Si elle n'existe pas, elle ne peut pas être évaluée, mais le système de types garantit que si l'expression est bien typée, alors il ne peut pas y avoir de variable introuvable à l'évaluation du programme.

$$\textsc{Var}\ \mathcal{E}, \mathcal{M} \vdash x \rightsquigarrow \mathcal{E}(x) \Big| \mathcal{M}$$

Pour les évaluations des constantes, ni l'environnement ni l'état mémoire ne sont affectés.

$$\textsc{Constant}\ \mathcal{E}, \mathcal{M} \vdash c \rightsquigarrow c \Big| \mathcal{M}$$

L'évaluation de l'abstraction construit une fermeture contenant le corps de l'abstraction ainsi que l'environnement courant. Une optimisation des compilateurs consiste à ne garder que les variables libres du corps dans l'environnement puisque les autres éléments de l'environnement sont alors inutiles.

Abstraction
$$\mathcal{E}, \mathcal{M} \vdash (\texttt{fun}\ x\ \texttt{->}\ e) \rightsquigarrow \Big\langle x, e, \mathcal{E} \Big\rangle \Big/ \mathcal{M}$$

L'application entre deux expressions peut se faire lorsque la première s'évalue en une fermeture. L'application peut entraîner des mutations de l'état mémoire. C'est notamment le cas si au moins l'une des deux expressions e_1 ou e_2 en entraînent.

Application
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \Big\langle x, e_3, \mathcal{E}_1 \Big\rangle \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big/ \mathcal{M}_2 \qquad (x, v_2) \lhd \mathcal{E}_1, \mathcal{M}_2 \vdash e_3 \rightsquigarrow v \Big/ \mathcal{M}_3}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow v \Big/ \mathcal{M}_3}$$

La conditionnelle entraîne l'évaluation de deux expressions : la condition et la conséquence ou bien la condition et l'alternative. L'état mémoire peut être affecté par l'une des expressions. Le système de types garantit que le résultat de l'évaluation de la condition est une valeur booléenne.

Conditional$_1$
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{true} \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \rightsquigarrow v \Big/ \mathcal{M}_2}$$

Conditional$_2$
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{false} \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_3 \rightsquigarrow v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \rightsquigarrow v \Big/ \mathcal{M}_2}$$

La définition locale associe une valeur à une variable.

Let-in
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \qquad (x, v_1) \lhd \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) \rightsquigarrow v \Big/ \mathcal{M}_2}$$

La définition récursive locale est limitée à la définition de fonctions (éventuellement récursives) conformément à la grammaire, la définition en elle-même n'entraîne pas de modification de l'état mémoire. En revanche, l'expression qui l'utilise peut bien sûr affecter l'état mémoire.

Rec-in
$$\frac{\mathcal{E}' \triangleq (x_1, \Big\langle x_2, e_1, \mathcal{E}' \Big\rangle) \lhd \mathcal{E} \qquad \mathcal{E}', \mathcal{M}_0 \vdash e_2 \rightsquigarrow v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{let rec}\ x_1\ \texttt{=}\ \texttt{fun}\ x_2\ \texttt{->}\ e_1\ \texttt{in}\ e_2) \rightsquigarrow v \Big/ \mathcal{M}_1}$$

Dans le cas des conjonctions logiques, la seconde expression n'est évaluée que si la première a été évaluée à **`true`** ; sinon, la seconde expression n'est pas évaluée et le résultat de la conjonction est **`false`**.

Sequential-Conjunction$_1$
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{false} \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ \texttt{\&\&}\ e_2) \rightsquigarrow \texttt{false} \Big/ \mathcal{M}_1}$$

Sequential-Conjunction$_2$
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{true} \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ \texttt{\&\&}\ e_2) \rightsquigarrow v \Big/ \mathcal{M}_2}$$

Le cas de la disjonction est similaire.

$$\textsc{Sequential-Disjunction}_1\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{true} \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1 \texttt{ || } e_2) \rightsquigarrow \texttt{true} \Big/ \mathcal{M}_1} \qquad \textsc{Sequential-Disjunction}_2\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{false} \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1 \texttt{ || } e_2) \rightsquigarrow v \Big/ \mathcal{M}_2}$$

Nous donnons la sémantique opérationnelle de la négation car nous considérons que **not** est un mot-clef. Autrement, nous aurions pu tout à fait définir **not** comme une fonction normale ayant par exemple pour valeur `fun e -> if e then false else true`.

$$\textsc{Negation}_1\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \texttt{true} \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{not(}e\texttt{)} \rightsquigarrow \texttt{false} \Big/ \mathcal{M}_1} \qquad \textsc{Negation}_2\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \texttt{false} \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{not(}e\texttt{)} \rightsquigarrow \texttt{true} \Big/ \mathcal{M}_1}$$

Introduction des traits impératifs

L'écriture d'un champ d'enregistrement avec une nouvelle valeur est une construction à effet secondaire (*side effect* en anglais, ou effet de bord).

La séquence de deux expressions évalue la première expression, qui selon le système de types doit s'évaluer en `()`, puis la seconde expression et retourne la valeur de cette dernière.

$$\textsc{Sequence}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_n \rightsquigarrow v_n \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{(}e_1 \texttt{ ; } e_n\texttt{)} \rightsquigarrow v_n \Big/ \mathcal{M}_2}$$

La construction d'un enregistrement commence par évaluer chacune des expressions à mettre dans les champs, en partant de la gauche vers la droite, pour ensuite construire la valeur enregistrement. Remarque : l'exhaustivité des champs est garantie par le typage.

$$\textsc{Record}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \quad \dots \quad \mathcal{E}, \mathcal{M}_{n-1} \vdash e_n \rightsquigarrow v_n \Big/ \mathcal{M}_n}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{\{ } f_1 \texttt{ = } e_1 \texttt{ ; } \dots \texttt{ ; } f_n \texttt{ = } e_n \texttt{ \}} \rightsquigarrow \texttt{\{ } f_1 \texttt{ = } v_1 \texttt{ ; } \dots \texttt{ ; } f_n \texttt{ = } v_n \texttt{ \}} \Big/ \mathcal{M}_x \qquad \mathcal{M}_x \triangleq (f_1, v_1) \lhd \dots \lhd (f_n, v_n) \lhd \mathcal{M}_n}$$

La lecture d'un champ d'un enregistrement n'est possible que si l'expression à gauche s'évalue en un enregistrement et qu'il possède le bon champ. Le système de types empêche de toutes manières la lecture invalide d'un champ.

$$\textsc{Record-Access}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v' \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash v'\texttt{.}f \rightsquigarrow v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash e\texttt{.}f \rightsquigarrow v \Big/ \mathcal{M}_1}$$

L'écriture d'un champ d'un enregistrement est une affectation par effet de bord. Elle modifie une valeur dans la mémoire.

$$\textsc{Record-Modify}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\texttt{.}f \texttt{<-} e_2) \rightsquigarrow \texttt{()} \Big/ \mathcal{M}_3 \qquad \mathcal{M}_3 \triangleq (v_1\texttt{.}f, v_2) \lhd \mathcal{M}_2}$$

Comme toute constante, un constructeur constant s'évalue en lui-même. La mémoire ne peut pas être modifiée pendant ce temps.

$$\textsc{constant-Constructor}\quad \mathcal{E}, \mathcal{M} \vdash C \rightsquigarrow C \Big| \mathcal{M}$$

La construction d'une valeur d'un type variant commence par l'évaluation de l'argument, puis la construction en annotant avec le constructeur la valeur obtenue.

$$\textsc{Constructor}\quad \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash C\,(e) \rightsquigarrow C\,(v) \Big| \mathcal{M}_1}$$

La boucle bornée est une expression à trois composantes, qu'on peut nommer e_1, e_2 et e_3. Les deux expressions e_1 et e_2 sont évaluées l'une après l'autre (et peuvent chacune modifier l'état mémoire) respectivement en v_1 et v_2 (qui sont deux nombres entiers, d'après le système de types). Si $v_1 > v_2$, alors on ne fait rien d'autre que retourner la valeur `()`. Sinon, on évalue e_3, qui peut affecter l'état mémoire, puis on ré-évalue la boucle avec les nouvelles valeurs $v_1 + 1$ et v_2 respectivement à la place de e_1 et e_2. On peut remarquer qu'après le premier tour de boucle, l'état mémoire ne peut plus être modifié que par le corps de la boucle.

$$\textsc{For}_1\quad \frac{\begin{array}{c}\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big| \mathcal{M}_2 \qquad \mathcal{E}, \mathcal{M}_2 \vdash v_1 > v_2 \rightsquigarrow \texttt{false} \Big| \mathcal{M}_2 \\ (x, v_1) \lhd \mathcal{E}, \mathcal{M}_2 \vdash e3 \rightsquigarrow \texttt{()} \Big| \mathcal{M}_3 \qquad \mathcal{E}, \mathcal{M}_3 \vdash \texttt{for}\ x = v_1 + 1\ \texttt{to}\ v_2\ \texttt{do}\ e_3\ \texttt{done} \rightsquigarrow \texttt{()} \Big| \mathcal{M}_4\end{array}}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} \rightsquigarrow \texttt{()} \Big| \mathcal{M}_4}$$

$$\textsc{For}_2\quad \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big| \mathcal{M}_2 \qquad \mathcal{E}, \mathcal{M}_2 \vdash v_1 > v_2 \rightsquigarrow \texttt{true} \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} \rightsquigarrow \texttt{()} \Big| \mathcal{M}_2}$$

La boucle conditionnelle (ou non bornée) s'évalue tant que la condition de continuité s'évalue en `true` ; à chaque tour de boucle, la condition est réévaluée et peut potentiellement changer l'état mémoire.

$$\textsc{While}_1\quad \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{true} \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \texttt{()} \Big| \mathcal{M}_2 \qquad \mathcal{E}, \mathcal{M}_2 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{()} \Big| \mathcal{M}_3}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{()} \Big| \mathcal{M}_3}$$

$$\textsc{While}_2\quad \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{false} \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{()} \Big| \mathcal{M}_1}$$

La levée d'exception (`raise` e) n'est possible que si e s'évalue en une valeur v. `Exception` v représente alors la levée de l'exception de valeur v.

$$\textsc{Raise}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{raise}\ e) \rightsquigarrow (\texttt{Exception}\ v) \Big| \mathcal{M}_1}$$

Contrôle de la levée d'exception Pour l'expression (**try** e_1 **with** x **->** e_2), dans le cas où e_1 s'évalue en une valeur, on rend cette valeur.

$$\textsc{Try}_1\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{try}\ e_1\ \texttt{with}\ x\ \texttt{->}\ e_2) \rightsquigarrow v \Big| \mathcal{M}_1}$$

Dans le cas où e_1 lève une exception v, elle est rattrapée, nommée x et l'expression e_2 est évaluée.

$$\textsc{Try}_2\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_1 \qquad (x, v) \triangleleft \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{try}\ e_1\ \texttt{with}\ x\ \texttt{->}\ e_2) \rightsquigarrow v \Big| \mathcal{M}_2}$$

Les filtres du filtrage par motifs sont notés $PAT(V, M)$ où V est la valeur filtrée et M est le motif filtrant. Si le motif M correspond à la valeur V, alors l'environnement des définitions courant est rendu éventuellement enrichi.

Le motif variable est toujours satisfait et rend l'environnement enrichi de l'association de la valeur filtrée avec la variable donnée.

$$\textsc{pat-var}\ \frac{}{\mathcal{E} \vdash PAT(v, x) \rightsquigarrow (x, v) \triangleleft \mathcal{E}}$$

Le motif constante est satisfait si la valeur filtrée est égale à cette constante.

$$\textsc{pat-Constant}_1\ \frac{}{\mathcal{E} \vdash PAT(c, c) \rightsquigarrow \mathcal{E}}$$

Le motif constructeur paramétré est satisfait si le constructeur utilisé est le même et le motif en paramètre est également satisfait. L'environnement enrichi par le filtrage du paramètre est rendu.

$$\textsc{pat-Constructor}\ \frac{\mathcal{E} \vdash PAT(v, p) \rightsquigarrow \mathcal{E}'}{\mathcal{E} \vdash PAT(C(v), C(p)) \rightsquigarrow \mathcal{E}'}$$

Pour les enregistrements, on filtre les valeurs des champs. On n'impose pas à tous les champs d'apparaître dans le filtre. On sait par le système de types que tous filtres sont valides.

$$\textsc{pat-Record}\ \frac{\mathcal{E} \vdash PAT(v_1, p_1) \rightsquigarrow \mathcal{E}' \qquad \dots \qquad \mathcal{E}^m \vdash PAT(v_n, p_n) \rightsquigarrow \mathcal{E}^n}{\mathcal{E} \vdash PAT(\{f_1 = v_1\ ;\ \dots\ ;\ f_n = v_n\ ;\ \dots\ \}, \{f_1 = p_1\ ;\ \dots\ ;\ f_n = p_n\ \}) \rightsquigarrow \mathcal{E}^n}$$

Autres cas (*i.e.*, aucun filtre n'a été satisfait)

$$\textsc{pat-no-Match}\ \frac{\mathcal{E} \vdash v \text{ est incompatible avec } p}{\mathcal{E} \vdash PAT(v, p) \rightsquigarrow []}$$

Filtrage avec la première branche qui filtre

$$\textsc{Match}_1\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1 \qquad \dfrac{\mathcal{E} \vdash PAT(v, p_1) \rightsquigarrow \mathcal{E}' \qquad \mathcal{E}' \neq []}{\mathcal{E}', \mathcal{M}_1 \vdash e_1 \rightsquigarrow v_1 \Big| \mathcal{M}_2}}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) \rightsquigarrow v_1 \Big| \mathcal{M}_2}$$

Filtrage avec la première branche qui ne filtre pas et une qui filtre

$$\textsc{Match}_2\ \frac{\begin{array}{c}\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1 \qquad \mathcal{E} \vdash PAT(v, p_1) \rightsquigarrow [] \quad \ldots \quad \mathcal{E} \vdash PAT(v, p_{k-1}) \rightsquigarrow [] \\ \mathcal{E} \vdash PAT(v, p_k) \rightsquigarrow \mathcal{E}' \qquad \mathcal{E}' \neq [] \qquad \mathcal{E}', \mathcal{M}_1 \vdash e_k \rightsquigarrow v_k \qquad k \leq n\end{array}}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) \rightsquigarrow v_k \Big| \mathcal{M}_2}$$

Filtrage avec aucune des branches qui ne filtre

$$\textsc{Match}_3\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1 \qquad \mathcal{E} \vdash PAT(v, p_1) \rightsquigarrow [] \quad \ldots \quad \mathcal{E} \vdash PAT(v, p_n) \rightsquigarrow []}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) \rightsquigarrow \textbf{Exception}\ \texttt{Match\_failure} \Big| \mathcal{M}_1}$$

III.1.2.4 Levées d'exceptions

Jusqu'à présent, les règles d'évaluation des expressions ne traitaient pas le cas où au lieu de s'évaluer en une valeur, une expression s'évalue en une levée d'exception. Il nous reste donc ces cas à traiter.

Une expression s'évalue en une levée d'exception si elle est elle-même la levée d'exception ou bien si l'une de ses sous-expressions s'évalue en une levée d'exception. Les règles formelles associées reprennent l'ensemble des règles données ici et spécifient simplement chaque cas où une prémisse lève une exception.

On donne ci-après trois règles associées à l'évaluation d'une application déclenchant une exception pour donner un aperçu.

$$\textsc{Application-Exception}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}$$

$$\textsc{Application-Exception}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \Big\langle x, e_3, E_1 \Big\rangle \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}$$

$$\textsc{Application-Exception}\ \frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \Big\langle x, e_3, E_1 \Big\rangle \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big| \mathcal{M}_2 \qquad (x, v_2) \lhd \mathcal{E}, \mathcal{M}_2 \vdash e_3 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_3}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_3}$$

L'ensemble complet des règles est un parcours exhaustif des constructions du langage où une expression peut s'évaluer en une levée d'exception. Comme les règles sont simples mais très nombreuses, elles sont fournies en annexe, section A.3, page 171.

III.1.3 Arbre d'évaluation

Lorsqu'on applique l'ensemble des règles d'évaluation de notre sémantique opérationnelle à un programme, on obtient ce qu'on appelle « un arbre d'évaluation. »

On donne à titre d'exemple l'arbre d'évaluation pour le (très petit) programme suivant.

```
let rec odd = fun n ->
  if eq n 0 then
    false
  else if eq n 1 then
    true
  else
    odd (sub n 2)
let res = odd 3
```

Pour alléger les notations,
- soit $\mathcal{D}_1$ la première définition,
- soit $\mathcal{D}_2$ la seconde,
- soit X = `if n = 0 then false else if n = 1 then true else odd (n - 2)`.

On va avoir besoin de la définition de deux fonctions (*i.e.*, l'égalité sur les entiers et la soustraction sur les entiers) pour donner une sémantique correcte à notre petit programme.

$$\texttt{eq} : \textbf{int->int->bool}$$

$$\texttt{sub} : \textbf{int->int->int}$$

Sémantique opérationnelle de ces deux fonctions appliquées à des valeurs :

Equal-Int
$$\frac{\mathcal{E}, \mathcal{M} \vdash e_1 \rightsquigarrow n_1 \big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow n_2 \big/ \mathcal{M}_2 \qquad n_1 = n_2}{\mathcal{E}, \mathcal{M} \vdash \texttt{eq}\ e_1\ e_2 \rightsquigarrow \textbf{true} \big/ \mathcal{M}}$$

not-Equal-Int
$$\frac{\mathcal{E}, \mathcal{M} \vdash e_1 \rightsquigarrow n_1 \big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow n_2 \big/ \mathcal{M}_2 \qquad n_1 \neq n_2}{\mathcal{E}, \mathcal{M} \vdash \texttt{eq}\ e_1\ e_2 \rightsquigarrow \textbf{false} \big/ \mathcal{M}}$$

Substraction
$$\frac{\mathcal{E}, \mathcal{M} \vdash e_1 \rightsquigarrow n_1 \big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow n_2 \big/ \mathcal{M}_2 \qquad n_1 - n_2 = n_3}{\mathcal{E}, \mathcal{M} \vdash \texttt{sub}\ e_1\ e_2 \rightsquigarrow n_3 \big/ \mathcal{M}_2}$$

N.B. Si l'un des arguments lève une exception, on appliquera les règles de l'application générale. Cependant, on pourra remarquer que notre petit programme ne lève pas d'exception.

Voici l'arbre d'évaluation du programme en plusieurs morceaux du fait que sa largeur ne lui permet pas de tenir entièrement sur une page si on ne le découpe pas.

$$\mathrm{A} = \frac{(n,3)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash n \rightsquigarrow 3 \Big/ \mathcal{M}}{(n,3)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathsf{sub}\ n\ 2 \rightsquigarrow 1 \Big/ \mathcal{M}}$$

$$\mathrm{B} = \frac{\dfrac{(n,1)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathsf{eq}\ n\ 1 \rightsquigarrow \mathtt{true} \Big/ \mathcal{M} \qquad (n,1)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathtt{true} \rightsquigarrow \mathtt{true} \Big/ \mathcal{M}}{(n,1)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash X \rightsquigarrow \mathtt{true} \Big/ \mathcal{M}}}{(n,1)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathsf{eq}\ n\ 1 \rightsquigarrow \mathtt{true} \Big/ \mathcal{M}}$$

$$\mathrm{C} = \frac{\mathcal{E}_1, \mathcal{M} \vdash \mathsf{odd} \rightsquigarrow \left\langle n, X, \mathcal{E}_1 \right\rangle \Big/ \mathcal{M} \qquad A \qquad B}{(n,3)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathsf{odd}\ (\mathsf{sub}\ n\ 2) \rightsquigarrow \mathtt{true} \Big/ \mathcal{M}}$$

$$\mathrm{D} = \frac{(n,3)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathsf{eq}\ n\ 0 \rightsquigarrow \mathtt{false} \Big/ \mathcal{M} \qquad (n,3)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash \mathsf{eq}\ n\ 1 \rightsquigarrow \mathtt{false} \Big/ \mathcal{M} \qquad C}{(n,3)\triangleleft\mathcal{E}_1, \mathcal{M} \vdash X \rightsquigarrow \mathtt{true} \Big/ \mathcal{M}}$$

$$\frac{\dfrac{\mathcal{E}_1 = (\mathsf{odd}, \left\langle n, X, \mathcal{E}_1 \right\rangle)\triangleleft\mathcal{E} \Big/ \mathcal{M}}{\mathcal{E}, \mathcal{M} \vdash \mathcal{D}_1 \rightsquigarrow \mathcal{E}_1 \Big/ \mathcal{M}} \qquad \dfrac{\dfrac{\mathcal{E}_1, \mathcal{M} \vdash \mathsf{odd} \rightsquigarrow \left\langle n, X, \mathcal{E}_1 \right\rangle \Big/ \mathcal{M} \qquad \mathcal{E}_1, \mathcal{M} \vdash 3 \rightsquigarrow 3 \Big/ \mathcal{M} \qquad D}{\mathcal{E}_1, \mathcal{M} \vdash \mathsf{odd}\ 3 \rightsquigarrow \mathtt{true} \Big/ \mathcal{M}}}{\mathcal{E}_1, \mathcal{M} \vdash \mathcal{D}_2 \rightsquigarrow (\mathtt{res}, \mathtt{true})\triangleleft\mathcal{E}_1, \mathcal{M}}}{\mathcal{E}, \mathcal{M} \vdash \mathcal{D}_1 \ \ \mathcal{D}_2 \rightsquigarrow (\mathtt{res}, \mathtt{true})\triangleleft\mathcal{E}_1, \mathcal{M}}$$

Maintenant que nous avons présenté mCaml, la section suivante va pouvoir spécifier ce qu'est la couverture des expressions pour ce langage.

III.2 Spécification de la couverture des expressions pour mCaml

Dans cette section, nous définissons formellement le critère de couverture structurelle de code simple pour le langage mCaml.

Le principe est le suivant : une expression atomique (constantes et variables) est couverte si elle est évaluée ; pour toutes les autres expressions, qui comportent toutes une ou plusieurs sous-expressions, il faut pour être couverte que chacune des sous-expressions soit couverte, et qu'en plus, sauf impossibilité totale, elle soit évaluée en une valeur. Un exemple d'impossibilité totale d'évaluation en une valeur : `(raise e)`. En effet, `(raise e)` ne peut, par définition, s'évaluer en une valeur.

Intuitivement, nous souhaitons que le taux de couverture soit de 100% pour un programme qui s'évalue sans problème et dont on ne peut pas améliorer la couverture en ajoutant des tests ou en modifiant des branchements conditionnels. Par exemple,

```
let f = fun x -> if false then 42 else 23 in
let a = f false in
let b = f true in
a
```

possède du code qui ne peut pas être couvert, qui est `42`. Ce code peut-être « corrigé » en

```
let f = fun x -> if x then 42 else 23 in
let a = f false in
let b = f true in
a
```

pour que `42` soit couvert : on a modifié un branchement conditionnel.

III.2.1 Règles formelles de couverture des expressions pour mCaml

On établit la couverture de code à partir d'un ensemble d'arbres d'évaluation noté T_R et obtenu à partir d'un ensemble de jeux de tests. *Un arbre d'évaluation résulte de l'évaluation d'un programme d'après sa sémantique opérationnelle, à partir d'un jeu de tests, comme défini plus haut.*

Le prédicat $\mathcal{C}$ est défini ci-après inductivement. Il prend une expression en argument et affirme sa couverture. Les expressions non-couvertes ne satisfont donc pas les règles.

Une constante est couverte si, et seulement si, elle apparaît dans une feuille d'un des arbres d'évaluation. *Ici, « une constante » désigne précisément une constante apparaissant à un endroit précis dans le code du programme. Par exemple, si la constante `42` apparaît plusieurs fois dans un programme, chacune de ses occurrences aura sa propre mesure de couverture.*

$$\mathcal{C}\text{-Constant}\ \frac{\exists T \in T_R \,\middle|\, (\mathcal{E}, \mathcal{M} \vdash \mathtt{c} \rightsquigarrow v \middle/ \mathcal{M}) \in T}{T_R \vdash \mathcal{C}(\mathtt{c})}$$

> La notation $(\mathcal{E}, \mathcal{M} \vdash \mathtt{c} \rightsquigarrow v) \in T$ signifie que $(\mathcal{E}, \mathcal{M} \vdash \mathtt{c} \rightsquigarrow v)$ apparaît dans l'arbre d'évaluation T. Cela n'arrive que si la constante `c` (précisément celle dont il est question dans le code du programme) a été évaluée.

Une variable est couverte si, et seulement si, elle apparaît dans une feuille d'un des arbres d'évaluation. *Ici, « une variable » désigne précisément une variable apparaissant à un endroit précis dans le code du programme. Si une même variable apparaît à plusieurs endroits dans le code, chacune de ses occurrences aura sa propre mesure de couverture.*

$$\mathcal{C}\text{-Variable}\ \frac{\exists T \in T_R \,\middle|\, (\mathcal{E}, \mathcal{M} \vdash \mathtt{x} \rightsquigarrow \mathcal{E}(\mathtt{x}) \middle/ \mathcal{M}) \in T}{T_R \vdash \mathcal{C}(\mathtt{x})}$$

Une abstraction est couverte si, et seulement si, son corps est couvert. Remarque : l'abstraction est une valeur. C'est lors de l'application qu'il est important de connaître le critère de couverture de l'abstraction, puisque c'est à ce moment là que le corps de l'abstraction est évalué.

$$\mathcal{C}\text{-Abstraction}\ \frac{T_R \vdash \mathcal{C}(\mathtt{e})}{T_R \vdash \mathcal{C}(\texttt{fun x -> e})}$$

Une application est couverte si, et seulement si, les sous-expressions qui la composent sont couvertes et qu'elle a été évaluée en une valeur.

$$\mathcal{C}\text{-Application}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1) \qquad T_R \vdash \mathcal{C}(\texttt{e}_2) \qquad \exists T \in T_R \,\middle|\, (\mathcal{E}, \mathcal{M}_1 \vdash \texttt{e}_1\ \ \texttt{e}_2 \rightsquigarrow v \Big/ \mathcal{M}_2) \in T}{T_R \vdash \mathcal{C}(\texttt{e}_1\ \ \texttt{e}_2)}$$

Une conditionnelle est couverte si, et seulement si, chacune de ses trois sous-expressions est couverte. On choisit ici de ne pas forcer l'évaluation des conditionnelles en une valeur. Une conditionnelle peut donc être couverte sans s'évaluer en une valeur. Ce cas se présente si, et seulement si, à la fois la conséquence et l'alternative sont toutes deux couvertes et évaluées au moins une fois chacune et toujours en une levée d'exception par un appel direct à la primitive **`raise`**.

$$\mathcal{C}\text{-Conditional}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1) \qquad T_R \vdash \mathcal{C}(\texttt{e}_2) \qquad T_R \vdash \mathcal{C}(\texttt{e}_3)}{T_R \vdash \mathcal{C}(\texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3)}$$

Une définition locale est couverte si, et seulement si, chacune de ses sous-expressions est couverte.

$$\mathcal{C}\text{-Let-in}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1) \qquad T_R \vdash \mathcal{C}(\texttt{e}_2)}{T_R \vdash \mathcal{C}(\texttt{let x = e}_1\ \texttt{in e}_2)}$$

$$\mathcal{C}\text{-Rec-in}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1) \qquad T_R \vdash \mathcal{C}(\texttt{e}_2)}{T_R \vdash \mathcal{C}(\texttt{let rec x}_1\ \texttt{= fun x}_2\ \texttt{-> e}_1\ \texttt{in e}_2)}$$

Un constructeur non-constant est couvert si, et seulement si, son argument est couvert.

$$\mathcal{C}\text{-Constructor}\ \frac{T_R \vdash \mathcal{C}(\texttt{e})}{T_R \vdash \mathcal{C}(\texttt{C(e)})}$$

Une séquence est couverte si, et seulement si, chacune de ses sous-expressions est couverte.

$$\mathcal{C}\text{-Sequence}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1) \qquad T_R \vdash \mathcal{C}(\texttt{e}_2)}{T_R \vdash \mathcal{C}(\texttt{e}_1\ \texttt{;}\ \texttt{e}_2)}$$

Un enregistrement est couvert si, et seulement si, chacune de ses sous-expressions est couverte et qu'elle s'est évaluée en une valeur.

$$\mathcal{C}\text{-Records}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1)\ \ \ldots\ \ \mathcal{C}(\texttt{e}_n) \qquad \exists T \in T_R \,\middle|\, (\texttt{\{ f}_1\ \texttt{= e}_1\ \texttt{; ... ; f}_n\ \texttt{= e}_n\ \texttt{\}})}{T_R \vdash \mathcal{C}(\texttt{\{ f}_1\ \texttt{= e}_1\ \texttt{; ... ; f}_n\ \texttt{= e}_n\ \texttt{\}})}$$

Un accès à un champ d'enregistrement est couvert si, et seulement si, l'enregistrement est couvert.

$$\mathcal{C}\text{-Access}\ \frac{T_R \vdash \mathcal{C}(\texttt{e})}{T_R \vdash \mathcal{C}(\texttt{e . f})}$$

Une modification d'un champ d'enregistrement est couvert si, et seulement si, l'enregistrement et l'expression rendant la nouvelle valeur sont couvertes.

$$\mathcal{C}\text{-Modify}\ \frac{T_R \vdash \mathcal{C}(\texttt{e}_1) \qquad T_R \vdash \mathcal{C}(\texttt{e}_2)}{T_R \vdash \mathcal{C}(\texttt{e}_1\ \texttt{. f <- e}_2)}$$

Une boucle bornée est couverte si, et seulement si, elle s'est évaluée en une valeur (*i.e.*, elle a terminé) et que chacune de ses sous-expressions est couverte.

$$\mathcal{C}\text{-For}\ \frac{\exists T \in T_R \,\Big|\, (\mathcal{E}, \mathcal{M}_1 \vdash \texttt{for x = } e_1 \texttt{ to } e_2 \texttt{ do } e_3 \texttt{ done} \rightsquigarrow () \Big/ \mathcal{M}_2) \in T \qquad T_R \vdash \mathcal{C}(e_1) \qquad T_R \vdash \mathcal{C}(e_2) \qquad T_R \vdash \mathcal{C}(e_3)}{T_R \vdash \mathcal{C}(\texttt{for x = } e_1 \texttt{ to } e_2 \texttt{ do } e_3 \texttt{ done})}$$

Une boucle conditionnelle est couverte si, et seulement si, elle s'est évaluée en une valeur et que chacune des sous-expressions est couverte.

$$\mathcal{C}\text{-While}\ \frac{\exists T \in T_R \,\Big|\, (\mathcal{E}, \mathcal{M}_1 \vdash \texttt{while } e_1 \texttt{ do } e_2 \texttt{ done} \rightsquigarrow () \Big/ \mathcal{M}_2) \in T \qquad T_R \vdash \mathcal{C}(e_1) \qquad T_R \vdash \mathcal{C}(e_2)}{T_R \vdash \mathcal{C}(\texttt{while } e_1 \texttt{ do } e_2 \texttt{ done})}$$

Une conjonction est couverte si, et seulement si, chacune de ses sous-expressions est couverte.

$$\mathcal{C}\text{-Conjunction}\ \frac{T_R \vdash \mathcal{C}(e_1) \qquad T_R \vdash \mathcal{C}(e_2)}{T_R \vdash \mathcal{C}(e_1 \texttt{ \&\& } e_2)}$$

Une disjonction est couverte si, et seulement si, chacune de ses sous-expressions est couverte.

$$\mathcal{C}\text{-Disjunction}\ \frac{T_R \vdash \mathcal{C}(e_1) \qquad T_R \vdash \mathcal{C}(e_2)}{T_R \vdash \mathcal{C}(e_1 \texttt{ || } e_2)}$$

Une négation est couverte si, et seulement si, son argument est couvert.

$$\mathcal{C}\text{-Negation}\ \frac{T_R \vdash \mathcal{C}(e)}{T_R \vdash \mathcal{C}(\texttt{not } e)}$$

Les parenthèses n'entraînent pas de calcul supplémentaire.

$$\mathcal{C}\text{-Parentheses}\ \frac{T_R \vdash \mathcal{C}(e)}{T_R \vdash \mathcal{C}(\texttt{(}e\texttt{)})}$$

Le contrôle des exceptions est couvert si, et seulement si, ses sous-expressions sont couvertes. On remarque que dans l'expression `try` e_1 `with x ->` e_2, si e_1 ne lève jamais d'exception, alors e_2 ne peut pas être couvert.

$$\mathcal{C}\text{-Try}\ \frac{T_R \vdash \mathcal{C}(e_1) \qquad T_R \vdash \mathcal{C}(e_2)}{T_R \vdash \mathcal{C}(\texttt{try } e_1 \texttt{ with x -> } e_2)}$$

Un filtrage par motifs est couvert si, et seulement si, l'expression filtrée est couverte ainsi que chacune des branches du filtrage.

$$\mathcal{C}\text{-Match}\ \frac{T_R \vdash \mathcal{C}(e) \qquad T_R \vdash \mathcal{C}(e_1) \quad \ldots \quad T_R \vdash \mathcal{C}(e_n)}{T_R \vdash \mathcal{C}(\texttt{match } e \texttt{ with } p_1 \texttt{ -> } e_1 \texttt{ | ... | } p_n \texttt{ -> } e_n)}$$

Une levée d'exception est couverte si, et seulement si, son argument est couvert et qu'il s'est évalué en une valeur.

$$\mathcal{C}\text{-Raise}\ \frac{T_R \vdash \mathcal{C}(e) \qquad \exists T \in T_R \,\Big|\, (\mathcal{E}, \mathcal{M}_1 \vdash e \rightsquigarrow v \Big/ \mathcal{M}_2) \in T}{T_R \vdash \mathcal{C}(\texttt{raise } e)}$$

Remarque : pour `raise(raise e)`, l'application la plus extérieure ne peut jamais être couverte car `(raise e)` ne s'évaluera jamais en une valeur.

En résumé Pour qu'une expression soit couverte, il faut que toutes ses sous-expressions soient couvertes, et s'il s'agit du cas particulier de la levée d'exception (**`raise`**), il faut en plus que l'argument du **`raise`** se soit évalué en une valeur.

III.2.2 Taux de couverture

Le taux τ de couverture structurelle simple des expressions pour un programme P est alors le ratio entre le nombre d'expressions couvertes et le nombre total d'expressions.

$$\tau = \frac{\left\|\{e \in P \mid \mathcal{C}(e)\}\right\|}{\left\|\{e \in P\}\right\|}$$

où $\left\|\{e \in P\}\right\|$ est le cardinal de l'ensemble des expressions e qu'on peut distinguer dans le programme P et $\left\|\{e \in P \mid \mathcal{C}(e)\}\right\|$ est le cardinal de l'ensemble des expressions e qu'on peut distinguer dans le programme P qui satisfont le prédicat $\mathcal{C}$.

III.3 Conclusion du chapitre

Nous avons présenté le langage mCaml (en laissant sa définition formelle complète en annexe). Ce langage servira de base pour le reste de cette thèse. Nous avons également défini formellement la couverture des expressions pour ce même langage.

Le chapitre suivant se consacrera à des critères plus complexes de couverture de code structurelle, qui font intervenir les notions de condition et de décision.

CHAPITRE IV

Couverture des conditions et des décisions pour mCaml

*« **Condition** : A Boolean expression containing no Boolean operators.*
***Decision** : A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition.*
If a condition appears more than once in a decision, each occurrence is a distinct condition. »
– DO-178B [73]

Sommaire

IV.1 Critères de couverture pour les expressions booléennes 62
IV.1.1 Couverture des conditions (CC) 62
IV.1.2 Couverture des décisions (DC) 63
IV.1.3 Couverture des conditions/décisions (C/DC) 63
IV.1.4 Couverture des conditions multiples (MCC) 63
IV.1.5 Couverture des conditions/décisions modifiée (MC/DC) 64
IV.1.6 Autres critères de couverture pour les expressions booléennes 69
IV.2 Conditions et décisions pour mCaml : quatre sémantiques formelles . . . 71
IV.2.1 Notations 71
IV.2.2 Sémantique 1 : repérage syntaxique aux branchements conditionnels . 72
IV.2.3 Sémantique 2 : sémantique intermédiaire, généralisation aux opérateurs booléens 74
IV.2.4 Sémantique 3 : une extension de la sémantique 2, généralisation aux applications à retour booléen 77
IV.2.5 Sémantique 4 : généralisation aux expressions booléennes 78
IV.3 Choix de la sémantique 84
IV.4 Fonction de comptage des conditions d'une expression 84
IV.5 Conclusion du chapitre 85

Résumé du chapitre 4

Ce chapitre présente premièrement différentes définitions pour les notions de conditions et de décisions dans le cadre de la couverture structurelle de code mCaml, dont MC/DC. La richesse du langage mCaml entraîne une difficulté d'interprétation et d'application des notions classiquement appliquées à de petits sous-ensembles des langages Ada (*e.g.*, SPARK) ou C.

Deuxièmement, nous définissons formellement les notions de conditions et décisions appliquées seules puis appliquées combinées au sein de critères complexes de couverture, tels que MC/DC (couverture des conditions/décisions modifiées).

Au chapitre précédent, nous avons vu la couverture structurelle simple pour le langage mCaml. Dans ce présent chapitre, nous nous intéressons à la couverture structurelle des expressions booléennes.

Les branchements conditionnels sont déterminés à partir de l'évaluation d'expressions booléennes. Si la valeur de ces expressions ne change jamais, alors il y a potentiellement une erreur. En effet, un branchement conditionnel dont la condition ne varie jamais n'est pas vraiment « conditionnel ». Alors pourquoi utiliser un branchement conditionnel ? Serait-ce une condition erronée ou une mauvaise structure du code ? Il est alors utile de chercher la raison pour laquelle un tel branchement existe dans un code.

Par exemple,

```
if (x = 0) || (x <> 0) then something() else somethingElse()
```

Les branchements conditionnels inconditionnels ? Quelques fois, on veut justement vérifier qu'une condition reste toujours constante, et signaler une erreur si elle varie, il s'agit de ce qui est communément appelé « assertions. » Mais ces cas sont une minorité des utilisations des expressions conditionnelles. Lorsque ces cas sont rencontrés, la couverture ne pourra être satisfaite. Dans le cadre d'un développement sous le contrôle d'une autorité de certification, il faudra probablement justifier leurs présences.

Par exemple,

```
if x then () else somethingWrong()
```

Deux catégories d'expressions booléennes. Lorsque l'on parcourt la littérature, on trouve deux notions pour distinguer deux catégories d'expressions booléennes : les conditions et les décisions. Intuitivement, les décisions sont les expressions booléennes qui se situent avant les branchements, et les conditions sont les expressions booléennes qui composent les décisions. Une décision peut alors être composée d'une ou de plusieurs conditions. Par exemple, l'expression $(x_1 \texttt{ || } x_2)$ est une décision composée de deux conditions qui sont les variables x_1 et x_2.

Pour un langage tout à fait minimal, ces définitions intuitives sont suffisantes. Mais dès que le langage s'enrichit un peu, elles deviennes ambiguës car les contextes peuvent rapidement devenir compliqués. Notamment pour un langage comme mCaml, nous verrons qu'il faut faire des choix d'interprétation.

Dans la suite de ce chapitre, nous consacrons d'abord une large section aux définitions de différents critères de couverture de code pour les expressions booléennes, dont le critère MC/DC. Cela permettra de mieux comprendre les impacts des définitions des notions de « condition » et de « décision » pour un langage d'expressions riche.

Ensuite, une seconde large section sera consacrée à définir formellement ces deux notions pour notre langage mCaml. Nous verrons ainsi les problématiques de choix d'interprétation de définitions simples appliquées à un langage riche. Cette section fournira quatre définitions, graduellement enrichies, des notions de condition et de décision pour mCaml.

IV.1 Critères de couverture pour les expressions booléennes

Cette section porte sur les critères de couverture conditionnelle classiquement rencontrés dans la littérature. Nous gravirons une échelle de complexité en partant d'un critère simple, celui de la couverture des conditions, pour arriver au fameux critère MC/DC.

Les définitions proposées dans cette section ne portent pas particulièrement sur mCaml, mais plutôt sur les langages de programmation en général. Ces définitions serviront alors de base et lorsque ce sera nécessaire, elles seront raffinées. D'autre part, les notions de « condition » et de « décision » utilisées dans cette section sont « génériques », en ce sens que quelles que soient les variantes des définitions utilisées pour ces deux mots, les définitions proposées dans cette section ne devraient pas en être impactées.

On peut toutefois affirmer que dans tous les cas, une condition et une décision sont des expressions booléennes.

Les valeurs booléennes ne sont pas les mêmes dans tous les langages. En C, un mot (*e.g.*, un entier, un flottant, une adresse mémoire) dont tous les bits sont à 0 représente faux, et tous les autres mots représentent vrai. En Java, nous avons les valeurs primitives **`true`** et **`false`**, comme pour ML.

En même temps, on peut dire que le type des booléens est représenté et représentable par deux ensembles de valeurs (qui sont souvent chacun des singletons mais pas toujours).

Ici, on choisit effectivement de ne considérer que les deux valeurs **`true`** et **`false`**.

IV.1.1 Couverture des conditions (CC)

La couverture des conditions, ou *condition coverage (CC)* en anglais, mesure quelles conditions ont été évaluées à la fois à la valeur **`true`** et à la valeur **`false`**.

On pourra éventuellement indiquer à l'utilisateur quelles conditions n'ont été évaluées qu'en une seule des deux valeurs booléennes.

Définition 1 ~ Taux de couverture des conditions

Le taux de couverture τ_c des conditions d'un programme P, est le rapport entre « le nombre de conditions qui ont été évaluées à **`true`** et à **`false`** » et « le nombre de conditions au total dans le programme », lors d'un ensemble d'exécutions de ce programme selon un ensemble de jeux de tests.

Formellement, cela peut s'écrire :

$$\tau_c = \frac{\left\| \{\ c : condition \in P \mid C_c(c) \} \right\|}{\left\| \{\ c : condition \in P\ \} \right\|}$$

où $C_c(c) = (\exists\ T \in\ T_r \ \middle|\ (\mathcal{E}, \mathcal{M} \vdash c \rightsquigarrow \texttt{true}) \in T) \wedge (\exists\ T \in\ T_r \ \middle|\ (\mathcal{E}, \mathcal{M} \vdash c \rightsquigarrow \texttt{false}) \in T)$ et signifie que « dans l'ensemble T_r des exécutions du programme P, il existe une exécution T du programme où l'expression c a été évalué en **`true`** et il existe une exécution du programme où c a été évalué en **`false`**. » ;
et $\{\ c : condition \in P\ \}$ dénote l'ensemble des conditions qui apparaissent dans P.
N.B. : il n'est pas nécessaire que c soit couvert du point de vue de la couverture structurelle simple des expressions. En effet, considérons par exemple l'expression suivante :
`((if true then f () else g ()); x1) && x2`.

Si on considère que la partie gauche (`(if true then f () else g ()); x1`) est une condition, alors il suffit qu'elle ait été évaluée à **`true`** et à **`false`** pour satisfaire la couverture des conditions. On observe alors que sa valeur ne dépend que de x_1. Il suffit donc que x_1 satisfasse la couverture des conditions pour que l'expression entière la satisfasse également. Or, la partie `g ()` ne pourra jamais être couverte structurellement.

IV.1.2 Couverture des décisions (DC)

La couverture des décisions, ou *decision coverage (DC)* en anglais, mesure quelles décisions ont été évaluées à la fois à **`true`** et à **`false`**.

Définition 2 ~ Taux de couverture des décisions

Le taux de couverture τ_d des décisions d'un programme P est le rapport entre « le nombre de décisions qui ont été évaluées à **`true`** et à **`false`** » et « le nombre de décisions au total dans le programme. »

Formellement, cela peut s'écrire :

$$\tau_d = \frac{\left\| \{\ d : decision \in P \mid C_d(d) \} \right\|}{\left\| \{\ d : decision \in P\ \} \right\|}$$

où $C_d(d) = \left(\exists\ T \in\ T_r \;\middle|\; (\mathcal{E}, \mathcal{M} \vdash d \rightsquigarrow \texttt{true}) \in T \right) \wedge \left(\exists\ T \in\ T_r \;\middle|\; (\mathcal{E}, \mathcal{M} \vdash d \rightsquigarrow \texttt{false}) \in T \right)$

IV.1.3 Couverture des conditions/décisions (C/DC)

La mesure de couverture des conditions et des décisions rapporte à la fois la mesure de couverture des conditions et la couverture des décisions.

Définition 3 ~ Taux de couverture des conditions et décisions

Le taux de couverture $\tau_{c/d}$ des conditions et des décisions d'un programme P est le rapport entre « le nombre de conditions et de décisions qui ont été évaluées à **`true`** et à **`false`** » et « le nombre de conditions et de décisions au total dans le programme. »

Formellement, cela peut s'écrire :

$$\tau_{c/d} = \frac{\left\| \{\ c : condition \in P \mid C_c(c) \} \right\| + \left\| \{\ d : decision \in P \mid C_d(d) \} \right\|}{\left\| \{\ c : condition \in P\ \} \right\| + \left\| \{\ d : decision \in P\ \} \right\|}$$

Remarque : on pourrait également choisir d'utiliser la moyenne des deux mesures ($\tau_{c/d} = (\tau_c + \tau_d)/2$). Dans tous les cas, l'important est que la détection des conditions et des décisions non couvertes reste effective.

IV.1.4 Couverture des conditions multiples (MCC)

La couverture des conditions multiples demande en plus du critère C/DC aux conditions d'une décision d'avoir pris toutes les configurations possibles. Donc pour une décision à n conditions, l'ensemble des combinaisons possibles des conditions donne 2^n configurations des conditions.

Remarque : Pour les opérateurs booléens séquentiels, on ne peut pas obtenir 2^n configurations différentes pour les conditions. Par exemple, pour e_1 `&&` e_2, si e_1 vaut **`false`** dans un test, alors e_2 ne sera pas du tout évalué. Donc le test contenant **`false`** pour e_1 sera le même pour e_2 à **`true`** et **`false`** (puisqu'on ne saura pas attribuer de valeur à e_2).

Généralement, le critère MCC n'est pas demandé car il serait très coûteux à atteindre dans les conditions de tests imposées par les processus de développement d'applications critiques.

IV.1.5 Couverture des conditions/décisions modifiée (MC/DC)

Intuitivement, le critère MC/DC demande, en plus de C/DC, à ce que les conditions d'une décision donnée puissent chacune avoir un effet sur la décision. Ce critère sert à donner une preuve que toutes les conditions servent. Les conditions qui ne satisfont pas ce critère sont suspectes ; le cas le plus simple où cela arrive est une décision où la même condition apparaît plusieurs fois dans une décision (*e.g.*, (`x && x`)). Il faut alors soit justifier la présence des conditions qui ne satisfont pas le critère ou bien mener une nouvelle campagne de tests.

Dans le document DO-178B [73], on trouve la définition suivante pour MC/DC.

> *Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to* ***independently*** *affect that decision's outcome. A condition is shown to* ***independently*** *affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.*

Ce critère est sujet à de nombreuses interprétations, certaines plus efficaces que d'autres pour la détection des erreurs, certaines plus coûteuses que d'autres. Les différentes interprétations seront souvent un compromis entre efficacité et coût.

Hayhurst *et al.* proposent un tutoriel sur ce critère de couverture de code [38].

Les membres de *Certification Authorities Software Team* (CAST) ont publié leur point de vue [13] sur l'interprétation de la notion de « décision ».

Chilenski étudie l'efficacité de trois interprétations de ce critère dans [14].

Comme ce critère est complexe et important notamment pour les projets soumis aux obligations de la DO-178B, on va développer davantage les motivations.

IV.1.5.1 Pourquoi MC/DC ?

Comprendre le critère MC/DC n'est pas toujours ce qu'il y a de plus facile et naturel pour un programmeur. On peut se voir rejeter cette notion tant qu'on ne l'a pas comprise, ou l'accepter en tant que contrainte mais sans en comprendre la réelle utilité.

On va essayer de comprendre un peu mieux les motivations de cette mesure.

Il faut une motivation assez forte pour suivre les obligations de mesures lorsqu'on veut satisfaire le critère MC/DC du fait du coût de cette mesure.

MC/DC sert entre autres à mettre en évidence une certaine catégorie d'erreurs lorsque le critère n'est pas satisfait, tout comme le typage à la ML permet entre autres d'empêcher une certaine catégorie d'erreurs. Les deux approches sont des compromis entre la quantité d'erreurs détectées et la quantité de programmes bons qui sont rejetés ou remis en question.

Pour satisfaire le critère MC/DC, il faut que chacune des conditions d'une décision ait été évaluée aux deux valeurs booléennes (**`true`** et **`false`**), et il faut montrer qu'elle a pu avoir un effet décisif sur

la décision. Le dernier point veut dire qu'il faut que chaque condition ait pu être évaluée à **true** et à **false** en faisant varier la valeur de la décision sans que les autres conditions aient changé de valeur entre ces deux tests. C'est le critère d'**indépendance des conditions**.

Par exemple, une expression telle que `(x1 > 42) && (x1 > 0)` est rarement écrite intentionnellement car si x_1 est plus grand que 42, il est forcément plus grand que 0, ce qui fait qu'il y a une forte dépendance entre les deux conditions `(x1 > 42)` et `(x1 > 0)`, donc une potentielle erreur. *N.B. Si c'est intentionnel, par exemple si le programme est concurrent et que le but est de détecter les variations de valeurs ou si on remplace x_1 par une expression qui va lire une valeur sur un périphérique, dans le pire cas on n'aura que fait une détection inutile dans un cas non général, ce qui n'est pas vraiment gênant.*

L'indépendance entre les conditions n'a pas non plus à être absolue. Il faut montrer au moins deux configurations des conditions et décisions où chaque condition a montré qu'elle a pu affecter la décision indépendamment des autres conditions, mais elle a le droit de ne pas être complètement indépendante.

Par exemple, une expression telle que `(x1 mod 2 = 0) && (x1 > 0)` peut satisfaire MC/DC avec un bon jeu de tests alors que les deux conditions sont liées puisqu'elles dépendent toutes deux de la valeur de la variable x_1. Si on considère les quatre tests suivants (où `(true)` signifie que si l'opérateur est séquentiel alors l'expression n'est pas évaluée, sinon elle est évaluée en **true**) :

#	entrée	valeurs des conditions		valeur de la décision
	x_1	x_1 mod 2 = 0	$x_1 > 0$	(x_1 mod 2 = 0) && ($x_1 > 0$)
1	0	**true**	**false**	**false**
2	1	**false**	(true)	**false**
3	2	**true**	**true**	**true**
4	-1	**false**	(false)	**false**

alors on constate que les tests 2 et 4 font (potentiellement) varier la valeur prise par la seconde condition sans faire varier la décision, parce qu'il y a effectivement une dépendance. Mais on constate également que les tests 1 et 3 montrent que **la seconde condition est capable de faire varier le résultat de la décision**, donc **sa présence n'est pas remise en question par MC/DC** avec ces quatre tests, et on peut remarquer que le critère MC/DC est satisfait avec seulement les trois premiers tests.

IV.1.5.2 Définition formelle de la mesure MC/DC

Remarque préliminaire à la définition suivante : *Quelque soit la variante de la définition du mot « décision », il s'agit toujours d'un ensemble non vide de « conditions » reliées entre elles d'une manière ou d'une autre, par exemple par des opérateurs booléens. Une décision peut ne contenir qu'une condition.*

Définition 4 ~ Critère d'indépendance des conditions d'une décision

La couverture des conditions/décisions modifiée combine les couvertures des conditions et des décisions tout en mesurant le critère $\mathcal{I}(d)$ d'indépendance des conditions appartenant à la décision d.

La décision d satisfait les critères d'indépendance de ses conditions si pour chacune de ses conditions c_i, il existe deux « configurations de ses conditions » qui ne diffèrent que d'une et une seule valeur qui est celle de la condition c_i tout en changeant la valeur de la décision.

Formellement, le critère $\mathcal{I}$ peut s'écrire :

$\mathcal{I}(d) = \forall\ c_i \in d,\ \exists\ (v_j,\ v_k)\ \Big|\ \Big(v_j[i] \neq v_k[i]\ \wedge\ \forall l \neq i,\ v_j[l] = v_k[l]\ \wedge\ d|v_j \rightsquigarrow \texttt{true} \wedge d|v_k \rightsquigarrow \texttt{false}\Big)$
où
- $v[i]$ dénote la i-ème valeur de la configuration v (qui correspond à la valeur prise par la i-ème condition de d) ;
- $d|v \rightsquigarrow$ **true** dénote la décision d s'est évaluée en **true** lorsque ses conditions ont pris la configuration v.

Définition 5 ~ Taux de couverture des conditions/décisions modifiée

En plus du taux de couverture des conditions et des décisions $\tau_{c/d}$, nous ajoutons la satisfaction du critère d'indépendance $\mathcal{I}$, pour obtenir le taux de couverture des conditions/décisions modifiée $\tau_{mc/d}$.

Nous pouvons alors exprimer $\tau_{mc/d}$ formellement :

$$\tau_{mc/d} = \frac{\Big\|\{\ d \in P \mid C_d(d) \wedge \mathcal{I}(d)\}\Big\|}{\Big\|\{\ d \in P\ \}\Big\|}$$

Une autre définition du taux de satisfaction du critère MC/DC consiste à donner une mesure du taux de satisfaction du critère d'indépendance appliqué aux conditions plutôt qu'aux décisions. On nomme ce nouveau critère $\mathcal{I}'$. C'est d'ailleurs cette définition que MLcov [1] adopte.

$\mathcal{I}'(c)$ = « Soit d la décision contenant c. Il existe deux arbres d'évaluation T_1 et T_2 appartenant à l'ensemble des arbres d'évaluation du programme P tels que
- c s'est évaluée en v_c dans T_1
- et en $\neg v_c$ dans T_2,
- d s'est évaluée en v_d dans $T1$
- et en $\neg v_d$ dans T_2,
- et pour toute condition x de la décision d différente de c, x s'est évaluée en la même valeur dans T_1 et T_2. »

Définition 6 ~ Taux de couverture des conditions/décisions modifiée - bis

Avec le critère d'indépendance $\mathcal{I}'$, nous pouvons donner la définition formelle suivante

$$\tau_{mc/d} = \frac{\Big\|\{\ c \in P \mid \mathcal{I}'(c)\}\Big\|}{\Big\|\{\ c \in P\ \}\Big\|}$$

Proposition 1 ~ Équivalence de $\mathcal{I}$ et $\mathcal{I}'$ pour la couverture complète

Toutes les conditions c d'un programme P satisfont le critère $\mathcal{I}'$ si, et seulement si, toutes les décisions d du même programme P satisfont le critère $\mathcal{I}$.

$$(\forall c : condition \in P, \mathcal{I}'(c)) \Longleftrightarrow (\forall d : decision \in P, \mathcal{I}(d))$$

1. MLcov est un outil de couverture de code pour les programmes développés en OCaml.

Preuve 1 ~ Équivalence de $\mathcal{I}$ et $\mathcal{I}'$ pour la couverture complète

Pour satisfaire le critère $\mathcal{I}$, une décision doit avoir chacune de ses conditions évaluées **indépendamment des autres conditions** à **`true`** et à **`false`** (ce qui sous-entend que la décision a également été évaluée à ces deux valeurs).

Pour satisfaire le critère $\mathcal{I}$', une condition doit avoir été évaluée à **`true`** et à **`false`** en ayant fait varier la décision (qui a donc été a fortiori évaluée aux deux valeurs) **indépendamment des autres conditions**.

Pour $(\forall c : condition \in P, \mathcal{I}'(c)) \Leftarrow (\forall d : decision \in P, \mathcal{I}(d))$:
Si toutes les décisions ont satisfait le critère $\mathcal{I}$, alors toutes les conditions de ces décisions ont influencé les résultats de leurs décisions indépendamment des autres conditions, ce qui revient à ce que toutes les conditions satisfassent le critère $\mathcal{I}'$.

Pour $(\forall c : condition \in P, \mathcal{I}'(c)) \Rightarrow (\forall d : decision \in P, \mathcal{I}(d))$:
Si toutes les conditions ont satisfait le critère $\mathcal{I}'$, alors toutes les conditions ont influencé les résultats de leurs décisions indépendamment des autres conditions, ce qui revient à ce que toutes les décisions satisfassent le critère $\mathcal{I}$. □

Proposition 2 ~ Non-équivalence de $\mathcal{I}$ et $\mathcal{I}'$ pour les couvertures incomplètes

Le calcul des taux est tout à fait différent lorsqu'il n'est pas de 100%.

Preuve 2 ~ Non-équivalence de $\mathcal{I}$ et $\mathcal{I}'$ pour les couvertures incomplètes

Pour un programme qui ne contient qu'une seule décision à deux conditions, si une seule des deux conditions satisfait le critère $\mathcal{I}'$, le taux de satisfaction de ce critère est de 50%, mais le taux de satisfaction du critère $\mathcal{I}$ est de 0%.

Exemple d'un tel programme : $\mathtt{x}_1$ **`&&`** $\mathtt{x}_2$ où $\mathtt{x}_2$ vaut toujours **`true`**. Dans ce programme, pour que $\mathtt{x}_1$ satisfasse le critère $\mathcal{I}'$, il suffit de le faire évaluer à **`true`** et à **`false`**, mais puisque $\mathtt{x}_2$ vaut toujours la même valeur, $\mathtt{x}_1$ **`&&`** $\mathtt{x}_2$ ne satisfera jamais le critère $\mathcal{I}$. □

$\mathcal{I}$ versus $\mathcal{I}'$ L'adoption du critère $\mathcal{I}'$ à la place du critère $\mathcal{I}$ permet une mesure plus fine lorsque la satisfaction du critère est partielle. Par exemple, pour une décision à trois conditions, avoir une condition qui n'est pas couverte implique ne pas satisfaire le critère $\mathcal{I}$ pour la décision entière, ce qui ne donne pas d'information sur les conditions qui posent problème, En revanche, si on utilise le critère $\mathcal{I}'$, on sait précisément quelles conditions satisfont le critère.

Opérateurs coupe-circuit Si on veut appliquer le critère MC/DC à des constructions linguistiques « coupe-circuit », comme par exemple les opérateurs booléens séquentiels, on doit relâcher un peu les critères. Pour $\mathcal{I}'$, cela donne :

$\mathcal{I}'(c)$ = « Soit d la décision contenant c. Il existe deux arbres d'évaluation T_1 et T_2 appartenant à l'ensemble des arbres d'évaluation du programme P tels que
- c s'est évaluée en v_c dans T_1
- et en $\neg v_c$ dans T_2,
- d s'est évaluée en v_d dans $T1$
- et en $\neg v_d$ dans T_2,
- et pour toute condition x de la décision d différente de c, x s'est évaluée en v_x ou ne s'est pas évaluée dans T_1, x s'est évaluée en v_x ou ne s'est pas évaluée dans T_2. »

IV.1.5.3 Au moins « $n+1$ tests pour le critère MC/DC »

Proposition 3 ~ MC/DC : $n+1$ tests au minimum pour une décision à n conditions

Pour satisfaire le critère $\mathcal{I}$, une décision d à n conditions doit être soumise à un jeu de tests de taille égale ou supérieure à $n+1$.

La proposition ne garantit pas qu'il existe un jeu de tests permettant de satisfaire le critère $\mathcal{I}$ pour une décision quelconque. Par exemple, il est impossible de satisfaire $\mathcal{I}$ pour la décision `(x && x)` *puisque varier la première condition varie également la seconde condition.*

Preuve 3 ~ MC/DC : $n+1$ tests au minimum pour une décision à n conditions

Remarque : Une décision ne peut pas contenir zéro condition.

Pour $n = 1$, les deux valeurs possibles de la condition sont les deux valeurs possibles de la décision. Cela fait exactement $2 = n+1$ tests.

Pour $n > 1$, on choisit une configuration des conditions. Ensuite, pour chacune des conditions, il faut faire varier la configuration d'une valeur (celle qui correspond à la condition). Comme il y a n conditions, cela fait n configurations de tests en plus du premier. On a bien $n+1$ tests pour la décision. (D'une certaine manière, on peut dire qu'*« on n'a pas la place d'en faire moins. »*) □

IV.1.5.4 Au plus « $2n$ tests pour le critère MC/DC »

Quand on rattache des conditions à une décision, usuellement il s'agit d'utiliser des opérateurs booléens de conjonction et de disjonction, qu'ils soient séquentiels ou non. On peut bien sûr rencontrer d'autres opérateurs logiques tels que la disjonction exclusive par exemple.

Plus loin dans ce chapitre (sections IV.2.4, page 77 et IV.2.5, page 78), nous proposerons de rattacher les conditions à une décision selon des moyens plus généraux que les opérateurs logiques. Par exemple, on peut se poser la question de la différence fondamentale entre un opérateur logique de conjonction (**&&**) et la construction conditionnelle (**if**), puisque le premier peut être remplacé par le second : e_1 `&&` e_2 $\rightsquigarrow$ `if` e_1 `then` e_2 `else false`.

Avec les opérateurs logiques séquentiels, certaines conditions sont inatteignables dans certaines configurations (*e.g.*, (`e1 && e2`) où `e1` s'évalue en **false**).

De manière plus générale, si pour un ensemble de conditions d'une décision, une seule condition peut être évaluée (c'est le minimum puisque sinon la décision ne prend pas de valeur) à la fois, le nombre de tests nécessaires devient $2n$ pour une décision à n conditions. En effet, pour que chaque condition soit évaluée aux deux valeurs booléennes, d'une part il faut deux tests par condition et d'autre part chaque condition affecte forcément la décision indépendamment des autres puisque les valeurs des autres ne sont pas calculées.

Cela augmente alors le nombre de tests nécessaires (*i.e.*, $2n$ au lieu de $n+1$), tout en simplifiant l'ensemble des tests puisque cela revient à satisfaire le critère CC (couverture des conditions).

En revanche, la combinaison de constructions demandant $2n$ tests et de constructions demandant $n+1$ tests peut engendrer un jeu de tests moins simple.

En somme, lorsqu'il est possible pour une décision et ses n conditions d'obtenir la satisfaction du critère MC/DC, il faut entre $n+1$ et $2n$ tests selon la nature de l'expression.

IV.1.6 Autres critères de couverture pour les expressions booléennes

Les différents critères de couverture de code présentés précédemment ne sont pas toujours ceux qui sont utilisés. D'autres critères de couverture de code peuvent être proposés.

Le critère MC/DC fait débat entre ceux qui l'adoptent et ceux qui le rejettent, ainsi que ceux qui ne le comprennent pas. Et il n'y a pas vraiment de consensus au sein de ceux qui l'adoptent, puisqu'il en existe plusieurs versions.

Néanmoins, nous pouvons voir le critère MC/DC comme un compromis permettant d'une part de ne pas faire les tests exhaustifs (critère MCC), et d'autre part de ne pas se contenter de faire le minimum de tests (par exemple deux tests, où toutes les conditions sont à **`true`** pour l'un, et **`false`** pour l'autre).

Yu *et al.* [89] proposent une comparaison de différents critères de couverture de décisions.

Xanthakis *et al.* [88] présentent et comparent différents critères de couverture de conditions et décisions.

IV.1.6.1 Couverture faible des conditions/décisions modifiée (Weak MC/DC)

La couverture des conditions/décisions modifiée affaiblie consiste à autoriser une condition à apparaître plusieurs fois dans une décision tout en permettant la satisfaction du critère MC/DC. Ainsi, une condition apparaissant plusieurs fois dans une décision n'est considérée qu'une seule fois.

Prenons par exemple un cas dégénéré : la décision `(x && x)` ne peut pas satisfaire le critère MC/DC puisqu'on ne pourra jamais faire varier la seconde branche du **`&&`** tout en maintenant inchangée la première branche. Le critère MC/DC faible ne demande pas de montrer l'indépendance de la seconde branche car `x` apparaît ailleurs, endroit où son indépendance par rapport aux autres conditions (si elles existaient) serait montrée.

Dans le cas où plusieurs conditions sont apparemment les mêmes mais produisent des effets de bord, cela se complique. Par exemple dans `e && e`, `e` pourrait être une lecture sur un flux de données. Dans ce cas, la raison d'utiliser un critère relâché n'est plus vraiment pertinent.

IV.1.6.2 Couverture des conditions/décisions renforcée (RC/DC)

RC/DC est un critère proposé par Vilkomir et Bowen [83]. Ce critère se veut renforcer le critère MC/DC en ajoutant des contraintes de tests « dans la mesure du possible ».

Voici comment ils formulent la définition du critère RC/DC, avec en gras italique les parties qui diffèrent de la définition de MC/DC :

> « Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, each condition in a

> decision has been shown to independently affect the decision's outcome, ***and each condition in a decision has been shown to independently keep the decision's outcome***. A condition is shown to independently affect ***and keep*** a decision's outcome by varying just that condition while holding fixed ***(if it is possible)*** all other conditions. »

Avec MC/DC, si deux ensembles v_1 et v_2 de n conditions montrent l'indépendance de la condition i, alors $(\forall 0 \leq j < n, (j \neq i \Rightarrow v_1[j] = v_2[j])) \wedge (v_1[i] \neq v_2[i])$ et si d_1 et d_2 sont les décisions correspondant respectivement aux conditions v_1 et v_2, alors $d_1 \neq d_2$.

Pour RC/DC, **dans la mesure du possible**, on veut qu'il y ait une mesure **supplémentaire par rapport à MC/DC** montrant qu'avec les ensembles v_1 et v_2 on obtient $d_1 = d_2$.

IV.1.6.3 Couverture des branches du code machine (OBC)

Attention, cette section utilise une définition restreinte (néanmoins assez classique) des notions de « condition » et « décision ». Nous avons donc un contraste avec les sections précédentes. La section IV.2 discutera et apportera différentes interprétations pour ces deux appellations en fournissant les sémantiques formelles.

Le critère « *Object Branch Coverage* » demande de couvrir uniquement les décisions d'un programme, sans regarder les conditions, mais dans sa forme compilée, donc au niveau du code objet (aussi appelé code machine). Les décisions dans un code objet sont les valeurs booléennes qui influent sur l'exécution des instructions de branchement.

Ce critère a été étudié [23] dans le cadre du projet « Couverture » [10] coordonné par la société AdaCore pour déterminer dans quelle mesure couvrir toutes les instructions d'un code machine suffisait à satisfaire le critère MC/DC.

L'avantage de ne couvrir que les branches au niveau du code machine est que le travail à fournir à l'exécution du programme est réduit. En quelque sorte, il suffit que chacun des tests de branchement ait été évalué à la fois à vrai et à faux.

Cependant, comme Beizer [5] l'indique et comme le rappellent Hayhurst *et al.* [38], la couverture du code objet ne peut pas satisfaire d'une manière générale la couverture du code source et vice versa.

L'idée de prendre cette mesure vient du fait que dans certains cas particuliers, obtenir une couverture à 100% selon le critère OBC équivaut à une couverture à 100% selon le critère MC/DC. En effet, si on se place dans un monde où les opérateurs booléens sont exclusivement séquentiels et qu'ils sont les seuls à composer les expressions booléennes, alors satisfaire OBC sur des expressions comme `((a && b) || (c && d))` équivaut à satisfaire MC/DC (car dans ce cas de figure, activer toutes les conditions à **`true`** et à **`false`** va impliquer la satisfaction du critère MC/DC). Cependant, satisfaire OBC pour une expression telle que `(((a && b) || c) && d)` avec un minimum de tests n'équivaut pas à satisfaire MC/DC. En effet, pour `(((a && b) || c) && d)`, les trois tests suivants suffisent à satisfaire le critère OBC pour cette décision à quatre conditions :
[**`true false true false`**] [**`false`** `true` **`false`** `true`] [**`true true`** `true` **`true`**]
(*a fortiori* MC/DC n'est alors pas satisfait ; les trois `true` qui ne sont pas en gras auraient pu être remplacés par `false` sans changer le résultat).

On constate effectivement que les opérateurs parenthésés à gauche posent problème dans l'équivalence entre OBC et MC/DC.

IV.2 Conditions et décisions pour mCaml : quatre sémantiques formelles

Les notions de « condition » et de « décision » sont ambiguës lorsque le langage de programmation est riche.

L'article « What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC) ? » [13] de Certification Authorities Software Team essaie d'apporter des réponses.

Mais quand il s'agit d'un langage applicatif tel que notre langage mCaml, cela se complique davantage. Cette problématique fait l'objet de cette section.

Ainsi, cette section donne différentes définitions formelles des notions de conditions et de décisions. Nous commençons par une sémantique très restrictive sur le style de programmation, qui réduit l'usage des expressions booléennes aux branchements pouvant exister dans un automate. Nous terminerons par une sémantique plus proche de l'habitude de programmation (à tendance très « libérale ») des utilisateurs des langages de la famille ML. Entre temps, nous aurons vu deux interprétations intermédiaires.

Remarque sur la construction `not`. Il est important de comprendre pourquoi on considérera toujours l'opérateur **`not`** comme neutre ou invisible dans la discrimination des conditions et décisions. Cet opérateur prend un booléen et en rend toujours un autre, il n'échoue jamais et est complètement déterministe. En particulier, on peut énoncer :

$$C_c(e) \Leftrightarrow C_c(\texttt{not}\ e)$$

Si on considérait son argument et son résultat comme deux conditions distinctes, on aurait la garantie de ne jamais pouvoir montrer l'indépendance de ces deux conditions, puisqu'elles sont au contraire directement et indéniablement dépendantes. Ceci reviendrait à interdire l'utilisation de cet opérateur, ce qui serait difficilement justifiable.

Cela aura pour impact dans ce chapitre ainsi que dans les deux chapitres suivants de traiter la construction **`not`** comme un parenthésage.

D'ailleurs, la DO-178C considère aussi que l'opérateur unaire de négation est neutre dans le comptage des conditions, comme le montre la définition donnée pour les **conditions**, avec en gras la partie qui n'apparaît pas dans celle donnée par la DO-178B :

> « ***Condition** : A Boolean expression containing no Boolean operators **except for the unary operator, that is, NOT**.* » [74]

IV.2.1 Notations

Voici la définition des notations utilisées dans la suite de cette section pour les quatre sémantiques $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ et $\mathcal{C}_4$.

- $\mathcal{C}_n$ est la fonction définie inductivement, en parcourant les expressions en profondeurs pour déterminer lesquelles sont des conditions et lesquelles sont des décisions, sous forme d'un environnement, pour la sémantique n. $\mathcal{C}_n$ prend en arguments un indicateur et une expression. L'indicateur permet de savoir si on se trouve dans une décision (d) ou pas ($\emptyset$).
  - $\mathcal{C}_n(\emptyset,$ `e : bool`) définit un environnement contenant les conditions et décisions apparaissant dans `e`, si, et seulement si, `e` est de type **`bool`**.

  - $\mathcal{C}_n(\emptyset, \texttt{e})$ définit un environnement contenant les conditions et décisions apparaissant dans e, quand e n'est pas de type **bool**.
  - $\mathcal{C}_n(d, \texttt{e : bool})$ définit un environnement contenant les conditions et décisions apparaissant dans l'expression e de type **bool** et appartenant à la décision d.
  - $\mathcal{C}_n(d, \texttt{e})$ la même chose que $\mathcal{C}_n(d, \texttt{e : bool})$ mais quand e n'est pas de type **bool**.
- $\texttt{e}, \texttt{e}_0, \texttt{e}_1, ..., \texttt{e}_n$ sont des expressions ;
- $\texttt{x}, \texttt{x}_0, \texttt{x}_1, ..., \texttt{x}_n$ sont des variables ;
- $\texttt{c}, \texttt{c}_0, \texttt{c}_1, ..., \texttt{c}_n$ sont des constantes.
- $\{(\ulcorner \texttt{e} \urcorner, d \rightarrow c)\}$ est le singleton qui indique que e est une condition de la décision d.
- $\{(\ulcorner \texttt{e} \urcorner, d)\}$ est le singleton qui indique que e est une décision et on la nomme d.
- L'union des informations définies par $\mathcal{C}_n(\emptyset, \texttt{e}_1)$ et $\mathcal{C}_n(\emptyset, \texttt{e}_2)$ est noté $\mathcal{C}_n(\emptyset, \texttt{e}_1) \cup \mathcal{C}_n(\emptyset, \texttt{e}_2)$.
- L'ajout d'une information I à un ensemble d'informations E est noté $I \lhd E$.

IV.2.2 Sémantique 1 : repérage syntaxique aux branchements conditionnels

IV.2.2.1 Particularités

L'application de règles de codage contraignantes qui refusent certaines formes d'expressions sont nécessaires pour satisfaire la définition donnée en début de chapitre [2], en effet une décision est une expression booléenne qui est repérée syntaxiquement. Elle se trouve avant un branchement, il s'agit donc de la condition de la conditionnelle (construction **if**) ou bien de la condition de la boucle conditionnelle (construction **while**).

Une décision peut être composée d'une ou de plusieurs conditions, qui sont des expressions booléennes composées par un ou plusieurs opérateurs booléens choisis parmi **&&** et **||** uniquement.

À l'exception des constantes **true** et **false** que cette sémantique autorise à apparaître à n'importe quel endroit valide, toutes les expressions booléennes doivent appartenir à une décision. Par exemple, $\texttt{x}_1$ **&&** $\texttt{x}_2$ ne peut pas apparaître ailleurs que dans la condition d'un **if** ou d'un **while**.

Cette première sémantique correspond aux habitudes des pratiques pour les projets DO-178B. En effet, l'utilisation d'une règle de codage très contraignante mais très simple rend cette sémantique relativement facile à présenter dans un tel projet.

IV.2.2.2 Sémantique formelle (1)

$\mathcal{C}_1$ est la fonction qui parcourt les expressions du langage mCaml pour déterminer lesquelles sont des conditions et lesquelles sont des décisions. Voici sa définition.

• Pour les expressions qui n'appartiennent pas à une décision et qui sont de type bool.
Toutes les expressions booléennes rencontrées en dehors d'une décision sont rejetées, à l'exception des constantes.
$\mathcal{C}_1(\emptyset, \texttt{true : bool}) = []$
$\mathcal{C}_1(\emptyset, \texttt{false : bool}) = []$
$\mathcal{C}_1(\emptyset, \texttt{e : bool})$ = coding-rule/error : not-in-a-decision

2. Rappel : « ***Condition*** *: A Boolean expression containing no Boolean operators.* ***Decision*** *: A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.* » [73]

• Pour les expressions qui n'appartiennent pas à une décision et qui ne sont pas de type `bool`. Nous appliquons là la fonction à toutes les sous-expressions des expressions qui ne sont pas de type **`bool`**.

Les deux seules constructions qui engendrent des décisions :

$\mathcal{C}_1(\emptyset, \texttt{while } e_1 \texttt{ do } e_2 \texttt{ done}) = \mathcal{C}_1(d, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = \mathcal{C}_1(d, e_1) \cup \mathcal{C}_1(\emptyset, e_2) \cup \mathcal{C}_1(\emptyset, e_3)$

> où d est une variable fraîche qui représente le repérage d'une nouvelle décision.
> **Cette notation est utilisée dans les 3 autres sémantiques**.

Les autres constructions non-booléennes :

$\mathcal{C}_1(\emptyset, c) = []$

$\mathcal{C}_1(\emptyset, x) = []$

$\mathcal{C}_1(\emptyset, e_1\ e_2) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, \texttt{fun x -> } e) = \mathcal{C}_1(\emptyset, e)$

$\mathcal{C}_1(\emptyset, \texttt{let rec } x_1 \texttt{ = fun } x_2 \texttt{ -> } e_1 \texttt{ in } e_2) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, \texttt{let x = } e_1 \texttt{ in } e_2) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, e_1 \texttt{ ; } e_2) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, \texttt{C(}e\texttt{)}) = \mathcal{C}_1(\emptyset, e)$

$\mathcal{C}_1(\emptyset, \texttt{\{ } f_1 \texttt{ = } e_1 \texttt{ ; ... ; } f_n \texttt{ = } e_n \texttt{ \}}) = \mathcal{C}_1(\emptyset, e_1) \cup \ldots \cup \mathcal{C}_1(\emptyset, e_n)$

$\mathcal{C}_1(\emptyset, e \texttt{ . } f) = \mathcal{C}_1(\emptyset, e)$

$\mathcal{C}_1(\emptyset, e_1 \texttt{ . } f \texttt{ <- } e_2) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, \texttt{for x = } e_1 \texttt{ to } e_2 \texttt{ do } e_3 \texttt{ done}) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2) \cup \mathcal{C}_1(\emptyset, e_3)$

$\mathcal{C}_1(\emptyset, \texttt{match } e \texttt{ with } p_1 \texttt{ -> } e_1 \texttt{ | ... | } p_n \texttt{ -> } e_n) = \mathcal{C}_1(\emptyset, e) \cup \mathcal{C}_1(\emptyset, e_1) \cup \ldots \cup \mathcal{C}_1(\emptyset, e_n)$

$\mathcal{C}_1(\emptyset, \texttt{raise } e) = \mathcal{C}_1(\emptyset, e)$

$\mathcal{C}_1(\emptyset, \texttt{try } e_1 \texttt{ with x -> } e_2) = \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(\emptyset, \texttt{(}e\texttt{)}) = \mathcal{C}_1(\emptyset, e)$

• Pour les expressions qui appartiennent à une décision. *Pour la présente sémantique, pour appartenir à une décision, il faut obligatoirement être de type **`bool`**.*

$\mathcal{C}_1(d, \texttt{true : bool}) = []$

$\mathcal{C}_1(d, \texttt{false : bool}) = []$

$\mathcal{C}_1(d, x \texttt{ : bool}) = \{(\ulcorner x \urcorner, d \rightarrow c)\}$

$\mathcal{C}_1(d, e_1\ e_2 \texttt{ : bool}) = \{(\ulcorner e_1\ e_2 \urcorner, d \rightarrow c)\} \cup \mathcal{C}_1(\emptyset, e_1) \cup \mathcal{C}_1(\emptyset, e_2)$

$\mathcal{C}_1(d, e \texttt{ . } f \texttt{ : bool}) = \{(\ulcorner e \texttt{ . } f \urcorner, d \rightarrow c)\} \cup \mathcal{C}_1(\emptyset, e)$

$\mathcal{C}_1(d, e_1 \texttt{ \&\& } e_2 \texttt{ : bool}) = \mathcal{C}_1(d, e_1) \cup \mathcal{C}_1(d, e_2)$

$\mathcal{C}_1(d, e_1 \texttt{ || } e_2 \texttt{ : bool}) = \mathcal{C}_1(d, e_1) \cup \mathcal{C}_1(d, e_2)$

$\mathcal{C}_1(d, \texttt{not } e \texttt{ : bool}) = \mathcal{C}_1(d, e)$

$\mathcal{C}_1(d, \texttt{(}e\texttt{) : bool}) = \mathcal{C}_1(d, e)$

*Voici les expressions complexes (dont le type est compatible avec le type **`bool`**) qui ne sont pas autorisées à apparaître dans les décisions.*

$\mathcal{C}_1(d, e_1 \texttt{ ; } e_2 \texttt{ : bool}) =$ coding-rule/error

$\mathcal{C}_1(d, \texttt{let rec } x_1 \texttt{ = fun } x_2 \texttt{ -> } e_1 \texttt{ in } e_2 \texttt{ : bool}) =$ coding-rule/error

$\mathcal{C}_1(d, \texttt{let x = } e_1 \texttt{ in } e_2 \texttt{ : bool}) =$ coding-rule/error

$\mathcal{C}_1(d, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ : bool}) =$ coding-rule/error

$\mathcal{C}_1(d, \texttt{match e with } p_1 \texttt{ -> } e_1 \texttt{ | ... | } p_n \texttt{ -> } e_n \texttt{ : bool})$ = coding-rule/error
$\mathcal{C}_1(d, \texttt{raise e : bool})$ = coding-rule/error
$\mathcal{C}_1(d, \texttt{try } e_1 \texttt{ with x -> } e_2 \texttt{ : bool})$ = coding-rule/error

Remarque. Une expression telle que $(x_1\ x_2$ **&&** $x_3\ x_4)$ est donc rejetée si x_2 ou x_4 est de type **bool**, mais également si l'ensemble de cette expression se trouve en dehors d'une construction **if** ou **while**.

Dans la sémantique suivante, on n'obligera plus une telle expression à appartenir à une construction **while** ou **if**.

Nous avons terminé la définition de notre première fonction de détermination des conditions et des décisions. Les règles de codage qui rejettent certaines expressions paraissent plutôt réalistes dans le cadre d'un développement de logiciel selon les exigences DO-178B.

IV.2.3 Sémantique 2 : sémantique intermédiaire, généralisation aux opérateurs booléens

Cette section autorise les expressions booléennes à apparaître ailleurs que devant les branchements des constructions **if** et **while**.

IV.2.3.1 Particularités

Ici, nous proposons d'autoriser les valeurs booléennes à apparaître n'importe où (si le système de types l'autorise). Nous pouvons observer plusieurs points importants :

- une décision est repérée syntaxiquement mais aussi par les types : soit il s'agit d'une expression booléenne sans opérateur booléen (parmi **&&** et **||**), soit il s'agit de plusieurs conditions composées par les opérateurs booléens **&&** et **||**,
- n'importe quelle expression de type **bool** est désormais valide (*i.e.*, conforme aux règles de codage) sauf si elle appartient à une décision auquel cas elle a des contraintes supplémentaires ;
- ainsi l'expression $(x_1$ **&& if** x_2 **then** x_3 **else** $x_4)$ est rejetée mais l'expression **(if** x_1 **&&** x_2 **then** x_3 **||** x_4 **else** $x_5)$ est acceptée.

Remarques. Cette version autorisant les décisions à apparaître ailleurs que devant les branchements des constructions **if** et **while** peut déjà surprendre dans le cadre d'un développement DO-178B. À cela nous pouvons voir deux raisons : tout d'abord, il faut admettre que c'est un peu étrange de garder l'appellation « décision » pour une expression qui n'est plus obligatoirement en position de décision à proprement parler. La seconde raison est que le milieu de la certification des logiciels critiques est plutôt conservateur.

Ce qui déplaît au premier regard dans l'approche autorisant les expressions booléennes à apparaître en dehors des constructions syntaxiques de branchements conditionnels vient surtout de la crainte des relâchements de contraintes sur la couverture de ces expressions. Pour compenser ce relâchement, nous appliquons les mêmes contraintes sur les « décisions » qui se trouvent hors des constructions syntaxiques **if** et **while** pour pallier les inquiétudes éventuelles.

La procédure consistant à faire en sorte que les décisions qui ne se situent pas avant les branchements subissent les mêmes procédures de tests que les décisions qui se situent juste avant les branchements n'est cependant pas nouvelle. Hayhurst et al. mentionnent déjà que les décisions ne doivent pas obligatoirement être avant un branchement dans [38]. D'autre part, elle a pu être jugée recevable dans le cadre de

la qualification de KCG[3] *dont les rapports de mesures de couverture structurelles étaient produits par MLcov*[4].

IV.2.3.2 Sémantique formelle (2)

Voici la définition de la fonction d'exploration des expressions déterminant les conditions et décisions d'une expression, avec règles de codage simples, qui autorise cette fois les expressions booléennes à apparaître en dehors d'une construction **`if`** ou **`while`**.

• Application aux expressions booléennes.

Toutes les expressions booléennes n'appartenant pas à une décision sont des décisions. Les constructions **`if`** et **`while`** ne subissent plus de traitement particulier.

$\mathcal{C}_2(\emptyset, \texttt{true : bool}) = []$
$\mathcal{C}_2(\emptyset, \texttt{false : bool}) = []$
$\mathcal{C}_2(\emptyset, \texttt{x : bool}) = \{(\ulcorner \texttt{x} \urcorner, d)\}$
$\mathcal{C}_2(\emptyset, \texttt{e . f : bool}) = \{(\ulcorner \texttt{e . f} \urcorner, d)\} \cup \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{e}_1\ \texttt{e}_2 \texttt{ : bool}) = \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d)\} \cup \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{e}_1 \texttt{ \&\& } \texttt{e}_2 \texttt{ : bool}) = \{(\ulcorner \texttt{e}_1 \texttt{ \&\& } \texttt{e}_2 \urcorner, d)\} \cup \mathcal{C}_2(d, \texttt{e}_1) \cup \mathcal{C}_2(d, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{e}_1 \texttt{ || } \texttt{e}_2 \texttt{ : bool}) = \{(\ulcorner \texttt{e}_1 \texttt{ || } \texttt{e}_2 \urcorner, d)\} \cup \mathcal{C}_2(d, \texttt{e}_1) \cup \mathcal{C}_2(d, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{not e : bool}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{(e) : bool}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{e}_1 \texttt{ ; } \texttt{e}_2 \texttt{ : bool}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{let rec x}_1 \texttt{ = fun x}_2 \texttt{ -> e}_1 \texttt{ in e}_2 \texttt{ : bool}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{let x = e}_1 \texttt{ in e}_2 \texttt{ : bool}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 \texttt{ : bool}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2) \cup \mathcal{C}_2(\emptyset, \texttt{e}_3)$
$\mathcal{C}_2(\emptyset, \texttt{match e with p}_1 \texttt{ -> e}_1 \texttt{ | ... | p}_n \texttt{ -> e}_n \texttt{ : bool}) =$
$\quad \mathcal{C}_2(\emptyset, \texttt{e}) \cup \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_2(\emptyset, \texttt{e}_n)$
$\mathcal{C}_2(\emptyset, \texttt{raise e : bool}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{try e}_1 \texttt{ with x -> e}_2 \texttt{ : bool}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$

• Application aux expressions non booléennes.

$\mathcal{C}_2(\emptyset, \texttt{c}) = []$
$\mathcal{C}_2(\emptyset, \texttt{x}) = []$
$\mathcal{C}_2(\emptyset, \texttt{e . f}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{e}_1 \texttt{ . f <- e}_2) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{e}_1\ \texttt{e}_2) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{(e)}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{C(e)}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{\{ f}_1 \texttt{ = e}_1 \texttt{ ; ... ; f}_n \texttt{ = e}_n \texttt{ \}}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_2(\emptyset, \texttt{e}_n)$
$\mathcal{C}_2(\emptyset, \texttt{e}_1 \texttt{ ; e}_2) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{let rec x}_1 \texttt{ = fun x}_2 \texttt{ -> e}_1 \texttt{ in e}_2) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{let x = e}_1 \texttt{ in e}_2) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{fun x -> e}) = \mathcal{C}_2(\emptyset, \texttt{e})$

3. On rappelle que KCG est le générateur de code de SCADE 6.
4. On rappelle que MLcov est un outil de couverture structurelle de code pour les programmes écrits en OCaml.

$\mathcal{C}_2(\emptyset, \texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2) \cup \mathcal{C}_2(\emptyset, \texttt{e}_3)$
$\mathcal{C}_2(\emptyset, \texttt{match e with p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n) =$
$\quad \mathcal{C}_2(\emptyset, \texttt{e}) \cup \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_2(\emptyset, \texttt{e}_n)$
$\mathcal{C}_2(\emptyset, \texttt{raise e}) = \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(\emptyset, \texttt{try e}_1\ \texttt{with x ->}\ \texttt{e}_2) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{while e}_1\ \texttt{do e}_2\ \texttt{done}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(\emptyset, \texttt{for x = e}_1\ \texttt{to e}_2\ \texttt{do e}_3\ \texttt{done}) = \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2) \cup \mathcal{C}_2(\emptyset, \texttt{e}_3)$

• Application aux expressions booléennes appartenant à des décisions. Ici, on interdit l'utilisation d'expressions « complexes » dans les décisions par des règles de codage :
$\mathcal{C}_2(d, \texttt{true : bool}) = []$
$\mathcal{C}_2(d, \texttt{false : bool}) = []$
$\mathcal{C}_2(d, \texttt{x : bool}) = \{(\ulcorner \texttt{x} \urcorner, d \rightarrow c)\} \cup []$
$\mathcal{C}_2(d, \texttt{e . f : bool}) = \{(\ulcorner \texttt{e . f} \urcorner, d \rightarrow c)\} \cup \mathcal{C}_2(\emptyset, \texttt{e})$
$\mathcal{C}_2(d, \texttt{e}_1\ \texttt{e}_2\ \texttt{: bool}) = \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d \rightarrow c)\} \cup \mathcal{C}_2(\emptyset, \texttt{e}_1) \cup \mathcal{C}_2(\emptyset, \texttt{e}_2)$
$\mathcal{C}_2(d, \texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2\ \texttt{: bool}) = \mathcal{C}_2(d, \texttt{e}_1) \cup \mathcal{C}_2(d, \texttt{e}_2)$
$\mathcal{C}_2(d, \texttt{e}_1\ \texttt{||}\ \texttt{e}_2\ \texttt{: bool}) = \mathcal{C}_2(d, \texttt{e}_1) \cup \mathcal{C}_2(d, \texttt{e}_2)$
$\mathcal{C}_2(d, \texttt{not e : bool}) = \mathcal{C}_2(d, \texttt{e})$
$\mathcal{C}_2(d, \texttt{(e) : bool}) = \mathcal{C}_2(d, \texttt{e})$
$\mathcal{C}_2(d, \texttt{e}_1\ \texttt{;}\ \texttt{e}_2\ \texttt{: bool}) =$ coding-rule/error
$\mathcal{C}_2(d, \texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3\ \texttt{: bool}) =$ coding-rule/error
$\mathcal{C}_2(d, \texttt{let rec x}_1\ \texttt{= fun x}_2\ \texttt{->}\ \texttt{e}_1\ \texttt{in e}_2\ \texttt{: bool}) =$ coding-rule/error
$\mathcal{C}_2(d, \texttt{let x = e}_1\ \texttt{in e}_2\ \texttt{: bool}) =$ coding-rule/error
$\mathcal{C}_2(d, \texttt{match e with p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n\ \texttt{: bool}) =$ coding-rule/error
$\mathcal{C}_2(d, \texttt{raise e : bool}) =$ coding-rule/error
$\mathcal{C}_2(d, \texttt{try e}_1\ \texttt{with x ->}\ \texttt{e}_2\ \texttt{: bool}) =$ coding-rule/error

• Les expressions non-booléennes qui appartiennent à une décision sont impossibles car, dans cette sémantique, les seules expressions qui appartiennent à des décisions sont booléennes par construction syntaxique directe. En effet, il faut être soi-même une expression booléenne, ou bien être directement à gauche ou à droite d'un opérateur booléen, et dans les autres cas soit l'expression est rejetée (*e.g.*, la séquence : $\texttt{(e}_1\ \texttt{;}\ \texttt{e}_2\texttt{)}$ est rejetée si $\texttt{(e}_1\ \texttt{;}\ \texttt{e}_2\texttt{)}\ \texttt{\&\&}\ \texttt{e}_3$), soit le lien généalogique avec la décision n'est pas transmis (*e.g.*, l'application : $\texttt{e}_2$ n'est pas liée à une décision si $\texttt{(e}_1\ \texttt{e}_2\texttt{)}\ \texttt{\&\&}\ \texttt{e}_3$).

Ces règles laissent une certaine liberté au programmeur. On interdit seulement d'écrire des expressions complexes lorsqu'on se trouve dans une décision.

Néanmoins, nous faisons toujours un traitement particulier pour les opérateurs logiques séquentiels **&&** et **||**. Ces opérateurs doivent-ils être les seuls à engendrer des liens entre différentes expressions booléennes, alors appelées conditions, pour former des décisions ? Qu'en est-il de l'application générale, à retour booléen ? Ici, nous l'avons traité comme une expression quelconque, où les arguments ne sont pas traités comme des conditions liées au résultat de l'application même quand ils sont booléens.

En effet, on remarque qu'il est possible avec la présente sémantique de définir une abstraction sur les opérateurs booléens (*e.g.*, `let conj = fun a b -> a && b`). Cela est un problème dans la mesure où, pour le critère MC/DC, il faut montrer l'indépendance de $\texttt{e}_1$ et $\texttt{e}_2$ dans $\texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2$ mais pas dans l'application $\texttt{(conj e}_1\ \texttt{e2)}$, ce qui est une potentielle porte ouverte aux abus.

IV.2.4 Sémantique 3 : une extension de la sémantique 2, généralisation aux applications à retour booléen

Dans cette section, nous proposons une sémantique basée sur la précédente pour que les applications soient toutes en mesure de lier leurs arguments booléens entre eux. Dans cette nouvelle sémantique, toutes les applications à résultat booléen sont des décisions et les paramètres booléens éventuels en sont les conditions.

Dans la sémantique précédente, les arguments booléens des applications ne sont pas liés entre eux. Par exemple, pour l'expression $\texttt{x}_0$ `(`$\texttt{x}_1$ `&&` $\texttt{x}_2$`)` `(`$\texttt{x}_3$ `&&` $\texttt{x}_4$`)`, $\texttt{x}_1$ et $\texttt{x}_2$ appartiennent à la même décision, mais $\texttt{x}_1$ et $\texttt{x}_3$ n'appartiennent pas à la même décision. Cela n'est pas forcément grave, mais si $\texttt{x}_0$ est en fait l'opération de conjonction (par exemple défini ainsi : `let` $\texttt{x}_0$ `= fun a b -> a && b`) et qu'on prend le critère MC/DC, alors on a évité de devoir montrer l'indépendance entre $\texttt{x}_1$ et $\texttt{x}_3$. On peut facilement imaginer des règles de codage pour éviter ces abus ; la plus drastique serait d'interdire les fonctions à paramètres booléens. Mais l'idée directrice est que nous allons plutôt choisir l'autre démarche, qui consiste à ajouter la liaison des paramètres des applications entre eux lorsqu'ils sont ou contiennent des booléens.

IV.2.4.1 Sémantique formelle (3)

La définition précédente $\mathcal{C}_2$ est reprise dans son ensemble, mais nous lui ajoutons le cas des applications à retours booléens présentés ci-après.

Application générale

Dans le cas où l'application est à retour booléen, il y a deux scénarios.

Le premier soutient que l'application n'appartient pas à une décision, auquel cas nous la reconnaissons en tant que nouvelle décision et ses arguments pourront contenir des conditions de cette décision.

Le second indique que l'expression est dans une décision, auquel cas nous la reconnaissons en tant que condition de la décision à laquelle elle appartient, et d'autre part indépendamment nous considérons l'expression comme une décision et ses arguments comme des éventuelles conditions de cette nouvelle décision.

> On peut faire l'analogie avec les opérateurs séquentiels de compositions de booléens.
> Par exemple, dans la décision `(`$\texttt{x}_1$ `||` $\texttt{x}_2$`)` `||` $\texttt{x}_3$, on ne veut pas montrer l'indépendance du résultat de `(`$\texttt{x}_1$ `||` $\texttt{x}_2$`)` avec $\texttt{x}_1$ et $\texttt{x}_2$ car cela serait impossible et n'aurait pas de sens.

$\mathcal{C}_3(\emptyset, \texttt{e}_1\ \texttt{e}_2\ \texttt{: bool}) = \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d)\} \cup \mathcal{C}_3(d, \texttt{e}_1) \cup \mathcal{C}_3(d, \texttt{e}_2)$
$\mathcal{C}_3(d, \texttt{e}_1\ \texttt{e}_2\ \texttt{: bool}) = \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d \to c)\} \cup \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d_2)\} \cup \mathcal{C}_3(d_2, \texttt{e}_1) \cup \mathcal{C}_3(d_2, \texttt{e}_2)$

Dans le cas où l'application n'est pas à retour booléen, soit elle n'est pas dans une décision et on garde la règle de la sémantique précédente, soit elle est dans une décision et on rattache les arguments booléens à la décision.

$\mathcal{C}_3(d, \texttt{e}_1\ \texttt{e}_2\ \texttt{: t}_{\texttt{t} \neq \texttt{bool}}) = \mathcal{C}_3(d, \texttt{e}_1) \cup \mathcal{C}_3(d, \texttt{e}_2)$

C'est cette règle qui donne un sens au rattachement de $\texttt{e}_1$ à une décision lorsque $\texttt{e}_1$ est une valeur fonctionnelle donc non booléenne. Elle sert à propager le lien généalogique de la décision.

Remarque Lorsqu'on rencontre une application à retour booléen dans une décision on ne pourrait pas simplement proposer que seuls les arguments soient rattachés à la décision, parce que rien ne garantit que les arguments comportent des booléens. Suivre cette voie rendrait la sémantique très complexe, donc plus difficile à appliquer. D'autre part, si on voulait faire un traitement différent pour les retours booléens d'applications contenant et ne contenant pas d'arguments booléens, on aurait des difficultés pédagogiques à justifier la démarche.

IV.2.5 Sémantique 4 : généralisation aux expressions booléennes

Cette section propose de généraliser davantage la démarche et autoriser toutes les expressions du langage. Ainsi, en cas d'implantation d'outil, les règles de codage seront complètement laissées libres aux jugements des programmeurs et de leurs autorités. Néanmoins, les nouvelles contraintes posées pourront éventuellement être vues comme des règles de codage implicites. En effet, avoir à satisfaire certaines obligations de mesures pourrait être dissuasif, ce qui revient à des règles de codage implicites.

IV.2.5.1 Motivations pour une sémantique généraliste

Les règles de codage peuvent gêner certains programmeurs qui veulent avoir la liberté d'exprimer leurs programmes de la manière dont ils le souhaitent. Dans cette section, nous proposons une définition des conditions et décisions permettant au programmeur de décider de la manière dont il programme.

Dans le cadre d'un développement de logiciel critique, il se peut fortement que les autorités de certification n'acceptent pas une telle liberté, du fait de la complexité des expressions pouvant en résulter. Mais il suffira probablement d'éditer et de respecter un guide de codage complémentaire pour construire un ensemble acceptable. « Qui peut le plus, peut le moins. »

L'idée générale que nous suivons ici est que chaque expression booléenne n'appartenant pas à une décision est une décision. Chaque expression booléenne qui est une sous-expression d'une décision est une condition de la décision, sauf pour quelques cas particuliers.

Si x_1 et x_2 sont des variables, alors en langage C, l'expression x_1 `||` x_2 pourrait être exprimée ainsi (`(`x_1 `!=` x_2 `? 1 :` x_1 `!= 0)`) sans changer la sémantique. En gardant l'exclusivité des rattachements des conditions aux décisions selon les opérateurs booléens de conjonction et de disjonction, et éventuellement en considérant la généralisation aux applications, on se retrouve quand même face à une situation où les conditions de la seconde version ne sont rattachées à aucune décision.

Pour se convaincre qu'en mCaml nous avons le même problème, écrivons-la :
`if` x_1 `<>` x_2 **`then true else`** x_2.
Nous avons une grande différence qui est que mCaml ne partage pas le domaine des booléens et le domaine des entiers. En conséquence, x_1 et x_2 sont repérées comme étant des variables de type **`bool`** (car dans une branche du **`if`** on rend **`true`** et dans l'autre x_2, et on compare x_1 à x_2, donc **`true`**, x_1 et x_2 sont du même type). Néanmoins en gardant la pratique habituelle, on ne relie pas le résultat de l'expression globale à une décision du fait de l'absence d'opérateur de composition booléenne. Enfin, en mCaml, on écrirait plutôt **`if`** x_1 **`then true else`** x_2 du fait de ne pas pouvoir « cacher » le fait que x_1 et x_2 sont des booléens (alors qu'en C, dans l'expression (`(`x_1 `!=` x_2 `? 1 :` x_1 `!= 0)`, x_1 et x_2 ne sont que des entiers comme les autres).

Faciliter artificiellement la satisfaction du critère MC/DC

Dans la présente sémantique, comme nous relions plus largement les conditions entre elles, nous empêchons par la mesure le genre de réécriture présentée dans le programme ci-après, où les fonctions `f` et `g` calculent la même chose, de deux manières différentes. *Dans les sémantiques précédentes, c'étaient les règles de codage qui jouaient ce rôle.*

```
let et = fun a b -> if a then b else false
let ou = fun a b -> if a then true else b

let f = fun a b c d ->
  (((a && b) || c) && d)

let g = fun a b c d ->
  (et (ou (et a b) c) d)
```

Avec une mesure qui ne relie pas les conditions `a`, `b`, `c` et `d` entre elles pour le corps de la fonction `g`, trois tests suffisent pour obtenir la satisfaction du critère MC/DC, alors que pour la fonction `f` il en faudra cinq. Les trois tests sont par exemple les suivants :

```
1 true  false true  false
2 false true  false true
3 true  true  true  true
```

IV.2.5.2 Sémantique formelle (4)

• Parcours des expressions booléennes qui n'appartiennent pas à une décision

$\mathcal{C}_4(\emptyset, \texttt{x : bool}) = \{(\ulcorner \texttt{x} \urcorner, d)\} \cup []$
$\mathcal{C}_4(\emptyset, \texttt{c : bool}) = []$
$\mathcal{C}_4(\emptyset, \texttt{e}_1\ \texttt{e}_2\ \texttt{: bool}) = \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d)\} \cup \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 \texttt{ : bool}) =$
$\quad \{(\ulcorner \texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 \urcorner, d)\} \cup \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2) \cup \mathcal{C}_4(d, \texttt{e}_3)$
$\mathcal{C}_4(\emptyset, \texttt{let x = e}_1 \texttt{ in e}_2 \texttt{ : bool}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{let rec x}_1 \texttt{ = fun x}_2 \texttt{ -> e}_1 \texttt{ in e}_2 \texttt{ : bool}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{e}_1 \texttt{ ; e}_2 \texttt{ : bool}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{e . f : bool}) = \{(\ulcorner \texttt{e . f} \urcorner, d)\} \cup \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(\emptyset, \texttt{e}_1 \texttt{ \&\& e}_2 \texttt{ : bool}) = \{(\ulcorner \texttt{e}_1 \texttt{ \&\& e}_2 \urcorner, d)\} \cup \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{e}_1 \texttt{ || e}_2 \texttt{ : bool}) = \{(\ulcorner \texttt{e}_1 \texttt{ || e}_2 \urcorner, d)\} \cup \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{not e : bool}) = \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(\emptyset, \texttt{(e) : bool}) = \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(\emptyset, \texttt{match e with p}_1 \texttt{ -> e}_1 \texttt{ | ... | p}_n \texttt{ -> e}_n \texttt{ : bool}) = \mathcal{C}_4(\emptyset, \texttt{e}) \cup \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_4(\emptyset, \texttt{e}_n)$
$\mathcal{C}_4(\emptyset, \texttt{raise e : bool}) = \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(\emptyset, \texttt{try e}_1 \texttt{ with x -> e}_2 \texttt{ : bool}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$

À propos de la conditionnelle et du filtrage par motifs
Pour la conditionnelle (*i.e.*, construction **`if`**), on propose de rattacher les sous-expressions de la même manière que pour les opérateurs **`&&`** et `||`. On remarque cependant que pour les deux branches de la conditionnelle, montrer leur « indépendance » est facilitée du fait que jamais elles ne sont évaluées en même temps.
On rappelle qu'on utilise un critère d'indépendance autorisant les autres conditions à n'avoir pas été évaluées, puisque autrement il ne serait pas possible d'utiliser les opérateurs coupe circuits (`&&` et `||`) tout en espérant obtenir la satisfaction du critère MC/DC.
Pour le filtrage par motifs (*i.e.*, construction **`match`**), on propose de ne pas rattacher les branches entre elles afin de simplifier la formule surtout vis-à-vis du reste de la thèse. Néanmoins, en adoptant les règles proposées ci-dessus, on remarque que les contraintes posées sur la conditionnelle pourraient inciter à utiliser le filtrage à la place. En effet, les deux constructions sont équivalentes dans les deux cas suivants :

1. **`if`** $\mathtt{e}_1$ **`then`** $\mathtt{e}_2$ **`else`** $\mathtt{e}_3$
2. **`match`** $\mathtt{e}_1$ **`with | true ->`** $\mathtt{e}_2$ **`| false ->`** $\mathtt{e}_3$

Pour pallier ce problème précis, on pourrait ajouter la règle suivante pour les expressions dont la valeur filtrée est booléenne :
$\mathcal{C}_4(\emptyset, \mathbf{match}\ \mathtt{e}\ \mathbf{:\ bool\ with}\ \mathtt{p}_1\ \mathtt{->}\ \mathtt{e}_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ \mathtt{p}_n\ \mathtt{->}\ \mathtt{e}_n\ \mathbf{:\ bool}) =$
$\quad \{(\ulcorner \mathbf{match}\ \mathtt{e}\ \mathbf{with}\ \mathtt{p}_1\ \mathtt{->}\ \mathtt{e}_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ \mathtt{p}_n\ \mathtt{->}\ \mathtt{e}_n \urcorner, d)\} \cup \mathcal{C}_4(d, \mathtt{e}) \cup \mathcal{C}_4(d, \mathtt{e}_1) \cup \ldots \cup \mathcal{C}_4(d, \mathtt{e}_n)$

Bien sûr, on pourrait directement utiliser une règle encore plus générale, sans contrainte de type sur la valeur filtrée :
$\mathcal{C}_4(\emptyset, \mathbf{match}\ \mathtt{e}\ \mathbf{with}\ \mathtt{p}_1\ \mathtt{->}\ \mathtt{e}_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ \mathtt{p}_n\ \mathtt{->}\ \mathtt{e}_n\ \mathbf{:\ bool}) =$
$\quad \{(\ulcorner \mathbf{match}\ \mathtt{e}\ \mathbf{with}\ \mathtt{p}_1\ \mathtt{->}\ \mathtt{e}_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ \mathtt{p}_n\ \mathtt{->}\ \mathtt{e}_n \urcorner, d)\} \cup \mathcal{C}_4(d, \mathtt{e}) \cup \mathcal{C}_4(d, \mathtt{e}_1) \cup \ldots \cup \mathcal{C}_4(d, \mathtt{e}_n)$

Cependant, cette obligation de mesure ne semble pas à première vue apporter un intérêt : les différentes branches seront toujours indépendantes entre elles puisqu'une seule d'entre elles peut être évaluée à la fois, donc il ne s'agirait que de montrer l'indépendance de chacune des conditions des branches avec les conditions contenues dans l'expression filtrée : cela donne un critère compliqué à expliquer. Il ne faut pas oublier que l'expression ne se trouve pas au sein d'une décision. Il faudrait faire une étude d'efficacité pour voir s'il est intéressant de retenir cette mesure.

• Parcours des expressions non booléennes qui n'appartiennent pas à une décision
$\mathcal{C}_4(\emptyset, \mathtt{x}) = []$
$\mathcal{C}_4(\emptyset, \mathtt{c}) = []$
$\mathcal{C}_4(\emptyset, \mathtt{e}_1\ \ \mathtt{e}_2) = \mathcal{C}_4(\emptyset, \mathtt{e}_1) \cup \mathcal{C}_4(\emptyset, \mathtt{e}_2)$
$\mathcal{C}_4(\emptyset, \mathbf{fun}\ \mathtt{x}\ \mathtt{->}\ \mathtt{e}) = \mathcal{C}_4(\emptyset, \mathtt{e})$
$\mathcal{C}_4(\emptyset, \mathbf{if}\ \mathtt{e}_1\ \mathbf{then}\ \mathtt{e}_2\ \mathbf{else}\ \mathtt{e}_3) = \mathcal{C}_4(\emptyset, \mathtt{e}_1) \cup \mathcal{C}_4(\emptyset, \mathtt{e}_2) \cup \mathcal{C}_4(\emptyset, \mathtt{e}_3)$
$\mathcal{C}_4(\emptyset, \mathbf{let}\ \mathtt{x}\ \mathtt{=}\ \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2) = \mathcal{C}_4(\emptyset, \mathtt{e}_1) \cup \mathcal{C}_4(\emptyset, \mathtt{e}_2)$
$\mathcal{C}_4(\emptyset, \mathbf{let\ rec}\ \mathtt{x}_1\ \mathtt{=}\ \mathbf{fun}\ \mathtt{x}_2\ \mathtt{->}\ \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2) = \mathcal{C}_4(\emptyset, \mathtt{e}_1) \cup \mathcal{C}_4(\emptyset, \mathtt{e}_2)$
$\mathcal{C}_4(\emptyset, \mathtt{C(e)}) = \mathcal{C}_4(\emptyset, \mathtt{e})$
$\mathcal{C}_4(\emptyset, \mathtt{e}_1\ \mathtt{;}\ \mathtt{e}_2) = \mathcal{C}_4(\emptyset, \mathtt{e}_1) \cup \mathcal{C}_4(\emptyset, \mathtt{e}_2)$
$\mathcal{C}_4(\emptyset, \mathtt{\{}\ \mathtt{f}_1\ \mathtt{=}\ \mathtt{e}_1\ \mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{f}_n\ \mathtt{=}\ \mathtt{e}_n\ \mathtt{\}}) = \mathcal{C}_4(\emptyset, \mathtt{e}_1) \cup \ldots \cup \mathcal{C}_4(\emptyset, \mathtt{e}_n)$
$\mathcal{C}_4(\emptyset, \mathtt{e}\ \mathtt{.}\ \mathtt{f}) = \mathcal{C}_4(\emptyset, \mathtt{e})$

$\mathcal{C}_4(\emptyset, \texttt{e}_1\ \texttt{.}\ \texttt{f}\ \texttt{<-}\ \texttt{e}_2) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \textbf{for}\ \texttt{x}\ \texttt{=}\ \texttt{e}_1\ \textbf{to}\ \texttt{e}_2\ \textbf{do}\ \texttt{e}_3\ \textbf{done}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2) \cup \mathcal{C}_4(\emptyset, \texttt{e}_3)$
$\mathcal{C}_4(\emptyset, \textbf{while}\ \texttt{e}_1\ \textbf{do}\ \texttt{e}_2\ \textbf{done}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$
$\mathcal{C}_4(\emptyset, \texttt{(e)}) = \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(\emptyset, \textbf{match}\ \texttt{e}\ \textbf{with}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n) = \mathcal{C}_4(\emptyset, \texttt{e}) \cup \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_4(\emptyset, \texttt{e}_n)$
$\mathcal{C}_4(\emptyset, \textbf{raise}\ \texttt{e}) = \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(\emptyset, \textbf{try}\ \texttt{e}_1\ \textbf{with}\ \texttt{x}\ \texttt{->}\ \texttt{e}_2) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$

• Parcours des expressions de type **bool** appartenant à une décision
$\mathcal{C}_4(d, \texttt{x}\ \texttt{:}\ \textbf{bool}) = \{(\ulcorner \texttt{x} \urcorner, d \to c)\} \cup []$
$\mathcal{C}_4(d, \texttt{c}\ \texttt{:}\ \textbf{bool}) = []$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d \to c)\} \cup \{(\ulcorner \texttt{e}_1\ \texttt{e}_2 \urcorner, d_2)\} \cup \mathcal{C}_4(d_2, \texttt{e}_1) \cup \mathcal{C}_4(d_2, \texttt{e}_2)$
$\mathcal{C}_4(d, \textbf{if}\ \texttt{e}_1\ \textbf{then}\ \texttt{e}_2\ \textbf{else}\ \texttt{e}_3\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2) \cup \mathcal{C}_4(d, \texttt{e}_3)$
$\mathcal{C}_4(d, \textbf{let}\ \texttt{x}\ \texttt{=}\ \texttt{e}_1\ \textbf{in}\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \textbf{let rec}\ \texttt{x}_1\ \texttt{=}\ \textbf{fun}\ \texttt{x}_2\ \texttt{->}\ \texttt{e}_1\ \textbf{in}\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{;}\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \texttt{e}\ \texttt{.}\ \texttt{f}\ \texttt{:}\ \textbf{bool}) = \{(\ulcorner \texttt{e}\ \texttt{.}\ \texttt{f} \urcorner, d \to c)\} \cup \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{||}\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \textbf{not}\ \texttt{e}\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \texttt{(e)}\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \textbf{match}\ \texttt{e}\ \textbf{with}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}) \cup \mathcal{C}_4(d, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_4(d, \texttt{e}_n)$
$\mathcal{C}_4(d, \textbf{raise}\ \texttt{e}\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \textbf{try}\ \texttt{e}_1\ \textbf{with}\ \texttt{x}\ \texttt{->}\ \texttt{e}_2\ \texttt{:}\ \textbf{bool}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$

• Parcours des expressions non booléennes qui appartiennent à une décision
$\mathcal{C}_4(d, \texttt{x}) = []$
$\mathcal{C}_4(d, \texttt{c}) = []$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{e}_2) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \textbf{fun}\ \texttt{x}\ \texttt{->}\ \texttt{e}) = \mathcal{C}_4(\emptyset, \texttt{e})$
$\mathcal{C}_4(d, \textbf{if}\ \texttt{e}_1\ \textbf{then}\ \texttt{e}_2\ \textbf{else}\ \texttt{e}_3) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2) \cup \mathcal{C}_4(d, \texttt{e}_3)$
$\mathcal{C}_4(d, \textbf{let}\ \texttt{x}\ \texttt{=}\ \texttt{e}_1\ \textbf{in}\ \texttt{e}_2) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \textbf{let rec}\ \texttt{x}_1\ \texttt{=}\ \textbf{fun}\ \texttt{x}_2\ \texttt{->}\ \texttt{e}_1\ \textbf{in}\ \texttt{e}_2) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \texttt{C(e)}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{;}\ \texttt{e}_2) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \texttt{\{}\ \texttt{f}_1\ \texttt{=}\ \texttt{e}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \texttt{f}_n\ \texttt{=}\ \texttt{e}_n\ \texttt{\}}) = \mathcal{C}_4(d, \texttt{e}_1) \cup \ldots \cup \mathcal{C}_4(d, \texttt{e}_n)$
$\mathcal{C}_4(d, \texttt{e}\ \texttt{.}\ \texttt{f}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \texttt{e}_1\ \texttt{.}\ \texttt{f}\ \texttt{<-}\ \texttt{e}_2) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$
$\mathcal{C}_4(d, \textbf{for}\ \texttt{x}\ \texttt{=}\ \texttt{e}_1\ \textbf{to}\ \texttt{e}_2\ \textbf{do}\ \texttt{e}_3\ \textbf{done}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2) \cup \mathcal{C}_4(\emptyset, \texttt{e}_3)$
$\mathcal{C}_4(d, \textbf{while}\ \texttt{e}_1\ \textbf{do}\ \texttt{e}_2\ \textbf{done}) = \mathcal{C}_4(\emptyset, \texttt{e}_1) \cup \mathcal{C}_4(\emptyset, \texttt{e}_2)$
$\mathcal{C}_4(d, \texttt{(e)}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \textbf{match}\ \texttt{e}\ \textbf{with}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n) = \mathcal{C}_4(d, \texttt{e}) \cup \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_n)$
$\mathcal{C}_4(d, \textbf{raise}\ \texttt{e}) = \mathcal{C}_4(d, \texttt{e})$
$\mathcal{C}_4(d, \textbf{try}\ \texttt{e}_1\ \textbf{with}\ \texttt{x}\ \texttt{->}\ \texttt{e}_2) = \mathcal{C}_4(d, \texttt{e}_1) \cup \mathcal{C}_4(d, \texttt{e}_2)$

À propos des expressions qui ne sont pas de type `bool` mais appartenant à une décision
S'il n'y a pas de rejet dû à des règles de codage, il y a tout de même des contraintes assez importantes qui peuvent décourager l'écriture d'expressions booléennes complexes comme par exemple la suivante :
`(`e_1 `;` e_2 `;` e_3`)` **`&&`** `(`e_4 e_5`)`
En effet, si e_1 et e_2 contiennent des sous-expressions booléennes, l'ensemble devient très compliqué à tester si on fait la mesure MC/DC car montrer l'indépendance des expressions booléennes contenues dans cette expression peut facilement s'avérer impossible.

À propos des conditions non liées par une décision
Lorsque plusieurs conditions sont des sous-expressions d'une expression en commun, si elles ne sont pas reliées par une décision commune, alors nous acceptons l'indépendance innée de ces conditions. Par exemple, pour le critère MC/DC, il n'y aura pas besoin de montrer l'indépendance de ces conditions via les procédures de tests, puisqu'il n'y a pas de décision.
Voici un exemple d'expression où au moins deux conditions (e_1 et e_4) ne sont pas liées par une décision :
`(fun x -> (if` e_1 **`then`** e_2 **`else`** e_3 **`; if`** e_4 **`then`** e_5 **`else ()))`**
Voici un autre exemple, où e_1 et e_2 sont de type **`bool`** :
`(`x_1 e_1 `;` e_2`)`

À propos des applications à types polymorphes
Les applications à types polymorphes peuvent être classées dans les quatre catégories suivantes.
La première représente les opérations où la valeur booléenne n'a pas d'influence sur le calcul, comme lorsqu'on met une donnée dans une structure de données ou lorsqu'on l'en extrait. Elle n'a donc pas d'effet sur les prises de décision. Il ne devrait pas y avoir de danger à ne pas appliquer les obligations de mesure MC/DC sur de telles applications. Voici un exemple où le site d'appel à `set` dans la définition de `f` peut être polymorphe et où si `n` est instancié en booléen, on n'effectue pas de mesure.

```
let set = fun x -> fun v ->
   x.contents <- v
let f = fun m -> fun n -> (* f : 'a ref -> 'a -> unit *)
   e; (* some actions *)
   set m n
```

La seconde catégorie est l'utilisation des valeurs fonctionnelles polymorphes. Dans ce cas, le calcul est délégué. Là il peut y avoir un calcul dépendant des paramètres booléens, selon ce qui est passé en paramètre. Par exemple, dans le corps de la fonction `apply2` définie ci-après, le site d'appel `(f a b)` n'est jamais vu comme un site contenant des valeurs booléennes.

```
let apply2 = fun f -> fun a -> fun b ->
   (f a b)
```

La troisième est les primitives et les appels externes à des fonctions qui ne peuvent pas être définies autrement que par des appels externes. Par exemple, les fonctions de comparaison polymorphes ne peuvent pas être définies directement dans le langage.
La quatrième catégorie est celle qui mélange les trois autres.

L'utilisation de valeurs fonctionnelles peut potentiellement servir à réduire les obligations de mesures. Cependant, une telle utilisation serait de la même classe que l'utilisation de liaisons pour séparer des conditions. Dans l'exemple suivant, si on applique les obligations de mesure MC/DC à la définition de `exemple1`, on s'affranchit de montrer l'indépendance de `x` et de `y` avec `z` et `t`, par rapport à `exemple2` qui calcule pourtant exactement la même chose.

```
let exemple1 =
  let a = x && y in
  let b = z && t in
     a && b

let exemple2 =
 (x && y) && (z && t)
```

Il y a alors deux voies possibles :

1. soit on considère que c'est à l'outil de mettre en évidence les cas potentiellement malicieux, et donc de les signaler ou de les interdire ;
2. soit on considère que c'est au développeur de prendre la responsabilité de ses actes, c'est-à-dire qu'il n'use pas de « ruses » pour faciliter la satisfaction des exigences de tests, et ce sera aux auditeurs de vérifier qu'il n'y a pas de mauvaises pratiques.

IV.3 Choix de la sémantique

Nous avons vu quatre interprétations des notions de condition et de décision. Selon le degré de liberté qu'on est prêt à offrir au programmeur, l'une ou l'autre des interprétations conviendra le mieux. Il est aussi bien entendu possible de donner des interprétations supplémentaires. Cependant, nous avons donné un large aperçu de ce que les interprétations pouvaient être.

Dans un cadre de développement très contraint et très surveillé, les deux premières interprétations peuvent bien convenir. La troisième interprétation apporte une procédure de tests plus large que la seconde, en ce sens que le travail de tests exigé est potentiellement plus grand tout en limitant les potentiels « masquages » de contraintes [5]. La quatrième interprétation est une suggestion sur la voie à emprunter si on veut laisser entier le langage, c'est-à-dire sans restriction.

L'idéal est probablement de laisser la liberté au programmeur tout en lui proposant un ensemble de règles à respecter pour ne pas rendre le programme impossible à couvrir.

Dans le reste du manuscrit, nous considérons par défaut seulement la quatrième interprétation puisque fondamentalement, les trois autres sont des sortes de restrictions de celle-ci.

IV.4 Fonction de comptage des conditions d'une expression

Nous aurons besoin dans les deux chapitres suivants d'une fonction de comptage des conditions qui se trouvent au sein d'une décision. Nous définissons ainsi cette fonction, notée $\#_d$ et conforme à la quatrième sémantique (fonction $\mathcal{C}_4$).

La fonction $\#_d$ n'est utilisée que pour compter les conditions qui appartiennent à la décision courante. Notamment, il ne s'agit pas de compter le nombre total des conditions dans l'absolu.

Cette fonction servira à savoir combien de cases il faut allouer pour les conditions contenues dans les sous-expressions lorsqu'on se trouve au sein d'une décision. Les expressions étant parcourues en profondeur par les algorithmes qui font appel à $\#_d$, ce sont ces algorithmes qui se chargeront de compter l'ensemble des conditions d'un programme.

Pour ces raisons, le parcours des expressions qui n'appartiennent pas à une décision n'est pas effectué par $\#_d$.

Voici sa définition :

• Parcours des expressions de type **bool** appartenant à une décision

$\#_d$(`x : bool`) = 1

$\#_d$(`c : bool`) = 0

$\#_d$(`e1 e2 : bool`) = 1

$\#_d$(`if e1 then e2 else e3 : bool`) = $\#_d$(`e1`) + $\#_d$(`e2`) + $\#_d$(`e3`)

$\#_d$(`let x = e1 in e2 : bool`) = $\#_d$(`e1`) + $\#_d$(`e2`)

$\#_d$(`let rec x1 = fun x2 -> e1 in e2 : bool`) = $\#_d$(`e2`)

$\#_d$(`e1 ; e2 : bool`) = $\#_d$(`e1`) + $\#_d$(`e2`)

$\#_d$(`e . f : bool`) = 1 + $\#_d$(`e`)

$\#_d$(`e1 && e2 : bool`) = $\#_d$(`e1`) + $\#_d$(`e2`)

$\#_d$(`e1 || e2 : bool`) = $\#_d$(`e1`) + $\#_d$(`e2`)

$\#_d$(`not e : bool`) = $\#_d$(`e`)

5. Par exemple, la sémantique 3 empêche de masquer les opérateurs booléens **&&** et **||**, c'est-à-dire en définissant des fonctions comme `let opAND a b = a && b`, ce qui pourrait potentiellement être fait avec la sémantique 2.

$\#_d$(`(e) : bool`) = $\#_d$(`e`)
$\#_d$(**match** `e` **with** p_1 `->` e_1 `|` `...` `|` p_n `->` e_n `: bool`) = $\#_d$(`e`) + $\#_d$(e_1) + ... + $\#_d$(e_n)
$\#_d$(**raise** `e : bool`) = $\#_d$(`e`)
$\#_d$(**try** e_1 **with** `x ->` e_2 `: bool`) = $\#_d$(e_1) + $\#_d$(e_2)

• Parcours des expressions qui ne sont pas de type **bool** mais appartenant à une décision
$\#_d$(`x : t` $_{t\neq\texttt{bool}}$) = 0
$\#_d$(`c : t` $_{t\neq\texttt{bool}}$) = 0
$\#_d$(e_1 e_2 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + $\#_d$(e_2)
$\#_d$(**fun** `x ->` `e : t` $_{t\neq\texttt{bool}}$) = 0
$\#_d$(**if** e_1 **then** e_2 **else** e_3 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + $\#_d$(e_2) + $\#_d$(e_3)
$\#_d$(**let** `x =` e_1 **in** e_2 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + $\#_d$(e_2)
$\#_d$(**let rec** x_1 `=` **fun** x_2 `->` e_1 **in** e_2 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_2)
$\#_d$(`C(e) : t` $_{t\neq\texttt{bool}}$) = $\#_d$(`e`)
$\#_d$(e_1 `;` e_2 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + $\#_d$(e_2)
$\#_d$(`{` f_1 `=` e_1 `;` `...` `;` f_n `=` e_n `}` `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + ... + $\#_d$(e_n)
$\#_d$(`e . f : t` $_{t\neq\texttt{bool}}$) = $\#_d$(`e`)
$\#_d$(e_1 `. f <-` e_2 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + $\#_d$(e_2)
$\#_d$(**for** `x =` e_1 **to** e_2 **do** e_3 **done** `: t` $_{t\neq\texttt{bool}}$) = 0
$\#_d$(**while** e_1 **do** e_2 **done** `: t` $_{t\neq\texttt{bool}}$) = 0
$\#_d$(`(e) : t` $_{t\neq\texttt{bool}}$) = $\#_d$(`e`)
$\#_d$(**match** `e` **with** p_1 `->` e_1 `|` `...` `|` p_n `->` e_n `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(`e`) + $\#_d$(e_1) + $\#_d$(e_n)
$\#_d$(**raise** `e : t` $_{t\neq\texttt{bool}}$) = $\#_d$(`e`)
$\#_d$(**try** e_1 **with** `x ->` e_2 `: t` $_{t\neq\texttt{bool}}$) = $\#_d$(e_1) + $\#_d$(e_n)

IV.5 Conclusion du chapitre

Ce chapitre a montré la problématique du choix de l'interprétation des définitions pour un langage de haut niveau qui sort du cadre des langages classiquement utilisés dans ces domaines.

Nous avons choisi l'interprétation la plus générale qui permet au programmeur d'exprimer ses algorithmes comme il le souhaite. Les autres interprétations ne sont que des restrictions sur le langage, faciles à appliquer en partant du cas le plus général. D'autres interprétations peuvent bien sûr être proposées, y compris des interprétations plus générales ou plus fines.

Nous pouvons maintenant passer à l'étape de la mise en œuvre, que traitent de deux manières différentes les deux chapitres suivants.

CHAPITRE V

Couverture de code structurelle intrusive pour mCaml

« *OCaml c'est du tonnerre.* »
– Emmanuel Chailloux

Sommaire

V.1 Instrumentation de code mCaml pour la couverture des expressions . . . 90
V.1.1 Instrumentation la plus générale 90
V.1.2 Instrumentation simple 91
V.1.3 Instrumentation avancée pour la couverture structurelle 92
V.1.4 Sémantique opérationnelle préservée 99
V.2 Instrumentation de code mCaml pour la mesure de satisfaction du critère MC/DC 99
V.3 Conclusion du chapitre 104

Résumé du chapitre 5

Ce chapitre présente une technique « intrusive » pour la couverture structurelle de code mCaml, incluant la mesure MC/DC. On associe « l'intrusion » au fait de changer le code source pour parvenir à obtenir les mesures voulues. La couverture structurelle de code intrusive est alors une manière d'obtenir la mesure voulue en réécrivant le code source afin de lui faire produire des traces d'exécution lors de l'utilisation du logiciel. Les traces produites à l'exécution sont ensuite utilisées pour générer des rapports de couverture structurelle de code. L'avantage principal de cette approche est sa relative simplicité de mise en œuvre. Ce chapitre peut être vu comme un développement formel des articles de Pagano *et al.* [68] et [66]

La manière la plus simple pour instrumenter un programme est de le réécrire en ajoutant des instructions d'émission de traces d'exécution correspondant aux données qu'on veut recueillir sur la façon dont le programme s'est exécuté. Par exemple, on peut imaginer une instrumentation simple pour implanter un traceur d'applications de fonctions : il suffit de transformer tous les sites d'appels, de la forme
`(`e_1 e_2`)`
où e_1 et e_2 sont n'importe quelles expressions valides, par
`(print_endline "Je vais invoquer (`e_1 e_2`)." ; `e_1 e_2`)`
pour que toutes les applications de fonctions soient indiquées tour après tour par le programme sur la sortie standard.

Si ensuite nous souhaitons aussi savoir quelles applications ont rendu un résultat, on peut instrumenter le programme autrement. Par exemple,
`(`e_1 e_2`)`
pourrait être réécrit en

```
(print_endline "Je vais invoquer (e1 e2)." ;
 let r = e1 e2 in
 print_endline "Une invocation de (e1 e2) a rendu un résultat." ;
 r)
```

Pour le profilage, l'instrumentation du code source par réécriture est utilisé, par exemple, par les outils `ocamlcp` et `ocamlprof` [52, chapitre 17] mais aussi par des outils plus largement diffusés comme `gcov` [1]. Le premier se charge d'instrumenter le programme et le second se charge de générer un rapport de profilage du code source du programme.

Ce qui nous intéresse est un peu plus complexe que le fait de savoir si une application de fonction a bien rendu un résultat. Les deux chapitres précédents ont présenté formellement les informations que nous souhaitons obtenir. Le chapitre III a montré formellement en section III.2.1 les informations élémentaires que nous voulons obtenir sur un programme. Il s'agit en quelque sorte de la généralisation à toutes les expressions du fait de savoir si une application a été invoquée et a rendu un résultat. Puis le chapitre IV a montré des critères plus complexes sur une catégorie spéciale de valeurs que sont les booléens.

Nous aurons une approche de progression de la « simplicité » vers la « nécessité. » La simplicité posera les bases du raisonnement de la technique d'instrumentation. La nécessité abordera les problèmes de performance technique qui viennent s'opposer à la simplicité.

Dans ce chapitre, nous allons d'abord voir comment on peut instrumenter un programme par réécriture pour obtenir les informations de couverture structurelle. Le choix des sites de code à instrumenter repose sur l'analyse sutructurelle d'un code applicatif tel qu'il est exprimé par le prédicat de couverture $\mathcal{C}$ défini au chapitre III. Ensuite, nous nous intéresserons aux critères de couverture des expressions conditionnelles présentés dans le chapitre IV, notamment le critère MC/DC qui est la partie la plus complexe en ce qui concerne l'instrumentation.

1. `gcov` est un outil de couverture fourni avec GCC. Sa documentation en ligne se trouve à l'URL `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

V.1 Instrumentation de code mCaml pour la couverture des expressions

La manière la plus simple, et certes naïve, d'instrumenter un programme mCaml pour lui faire émettre des traces pour la couverture des expressions est de réécrire chaque expression de la même façon. C'est-à-dire qu'une expression `(e)` devient alors
`(let v = e in emission_traces(); v)`
de même pour ses sous-expressions. *On remarque que le nom de variable* `v` *peut être utilisé indépendamment du contenu de l'expression* `e`. *Quant à* `emission_traces()`, *on considère qu'il s'agit d'une expression à effets de bord qui se charge d'émettre les traces correspondant au succès de l'évaluation de* `e`.

Si on veut tracer les exécutions ratées, on peut instrumenter `(e)` pour avoir davantage d'informations, et on obtient ainsi
`(emission_traces_pre() ; let v = e in emission_traces_post(); v)`
ce qui permet de savoir si l'évaluation de `(e)` a pu échouer (par exemple en levant une exception).

Si on veut également savoir quelle exception `(e)` a levée, `(e)` peut être réécrit en

```
(emission_traces_pre() ;
 let v = try e with x -> emission_traces_exception(x); raise x in
 emission_traces_post(); v)
```

où `emission_traces_exception(x)` trace la levée de l'exception `x` puis la lève à son tour pour ne pas changer le cours attendu de l'évaluation de `(e)`.

Dans la suite, on présente trois instrumentations :

1. la première est la plus simple possible (systématique) ;
2. la seconde reste simple mais essaie de limiter les marques manifestement inutiles ;
3. la troisième offre un compromis entre une instrumentation minimaliste et la simplicité.

V.1.1 Instrumentation la plus générale

On note ${}^{i}[\![\ ]\!]$ la fonction qui instrumente une expression, et $\{\!|\ |\!\}$ la fonction qui marque une expression. Le schéma de réécriture suivant montre une instrumentation systématique simple à exprimer.

$$
\begin{aligned}
&{}^{i}[\![\ \texttt{x}\ ]\!] \rightsquigarrow \{\!|\ \texttt{x}\ |\!\}\\
&{}^{i}[\![\ \texttt{c}\ ]\!] \rightsquigarrow \{\!|\ \texttt{c}\ |\!\}\\
&{}^{i}[\![\ \texttt{e}_1\ \texttt{e}_2\ ]\!] \rightsquigarrow \{\!|\ {}^{i}[\![\ \texttt{e}_1\ ]\!]\ {}^{i}[\![\ \texttt{e}_2\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{fun x -> e}\ ]\!] \rightsquigarrow \{\!|\ \texttt{fun x -> } {}^{i}[\![\ \texttt{e}\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3\ ]\!] \rightsquigarrow \{\!|\ \texttt{if } {}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{then } {}^{i}[\![\ \texttt{e}_2\ ]\!]\ \texttt{else } {}^{i}[\![\ \texttt{e}_3\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{let x = e}_1\ \texttt{in e}_2\ ]\!] \rightsquigarrow \{\!|\ \texttt{let x = } {}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{in } {}^{i}[\![\ \texttt{e}_2\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{let rec x}_1\ \texttt{= fun x}_2\ \texttt{-> e}_1\ \texttt{in e}_2\ ]\!] \rightsquigarrow \{\!|\ \texttt{let rec x}_1\ \texttt{= fun x}_2\ \texttt{-> } {}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{in } {}^{i}[\![\ \texttt{e}_2\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{C(e)}\ ]\!] \rightsquigarrow \{\!|\ \texttt{C(}\ {}^{i}[\![\ \texttt{e}\ ]\!]\ \texttt{)}\ |\!\}\\
&{}^{i}[\![\ \texttt{e}_1\ \texttt{;}\ \texttt{e}_2\ ]\!] \rightsquigarrow \{\!|\ {}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{;}\ {}^{i}[\![\ \texttt{e}_2\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{\{ f}_1\ \texttt{= e}_1\ \texttt{; ... ; f}_n\ \texttt{= e}_n\ \texttt{\}}\ ]\!] \rightsquigarrow \{\!|\ \texttt{\{ f}_1\ \texttt{= }{}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{; ... ; f}_n\ \texttt{= }{}^{i}[\![\ \texttt{e}_n\ ]\!]\ \texttt{\}}\ |\!\}\\
&{}^{i}[\![\ \texttt{e . f}\ ]\!] \rightsquigarrow \{\!|\ {}^{i}[\![\ \texttt{e}\ ]\!]\ \texttt{. f}\ |\!\}\\
&{}^{i}[\![\ \texttt{e}_1\ \texttt{. f <- e}_2\ ]\!] \rightsquigarrow \{\!|\ {}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{. f <- }{}^{i}[\![\ \texttt{e}_2\ ]\!]\ |\!\}\\
&{}^{i}[\![\ \texttt{for x = e}_1\ \texttt{to e}_2\ \texttt{do e}_3\ \texttt{done}\ ]\!] \rightsquigarrow \{\!|\ \texttt{for x = }{}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{to }{}^{i}[\![\ \texttt{e}_2\ ]\!]\ \texttt{do }{}^{i}[\![\ \texttt{e}_3\ ]\!]\ \texttt{done}\ |\!\}\\
&{}^{i}[\![\ \texttt{while e}_1\ \texttt{do e}_2\ \texttt{done}\ ]\!] \rightsquigarrow \{\!|\ \texttt{while }{}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{do }{}^{i}[\![\ \texttt{e}_2\ ]\!]\ \texttt{done}\ |\!\}\\
&{}^{i}[\![\ \texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2\ ]\!] \rightsquigarrow \{\!|\ {}^{i}[\![\ \texttt{e}_1\ ]\!]\ \texttt{\&\&}\ {}^{i}[\![\ \texttt{e}_2\ ]\!]\ |\!\}
\end{aligned}
$$

$^{i}[\![$ `e1 || e2` $]\!] \rightsquigarrow \{\!|\ ^{i}[\![$ `e1` $]\!]$ `||` $^{i}[\![$ `e2` $]\!]\ |\!\}$
$^{i}[\![$ **`not`** `e` $]\!] \rightsquigarrow$ **`not`** $^{i}[\![$ `e` $]\!]$
$^{i}[\![$ `(e)` $]\!] \rightsquigarrow$ `(` $^{i}[\![$ `e` $]\!]$ `)`
$^{i}[\![$ **`match`** `e` **`with`** p_1 `->` e_1 `|` `...` `|` p_n `->` e_n $]\!] \rightsquigarrow$
$\quad \{\!|$ **`match`** $^{i}[\![$ `e` $]\!]$ **`with`** p_1 `->` $^{i}[\![$ e_1 $]\!]$ `|` `...` `|` p_n `->` $^{i}[\![$ e_n $]\!]\ |\!\}$
$^{i}[\![$ **`raise`** `e` $]\!] \rightsquigarrow \{\!|$**`raise`** $^{i}[\![$ `e` $]\!]\ |\!\}$
$^{i}[\![$ **`try`** e_1 **`with`** `x ->` e_2 $]\!] \rightsquigarrow \{\!|$**`try`** $^{i}[\![$ e_1 $]\!]$ **`with`** `x ->` $^{i}[\![$ e_2 $]\!]\ |\!\}$

où $\{\!|$`e`$|\!\}$ peut avoir diverses définitions au choix :

- Le cas où l'instrumentation ne trace que la réussite d'une évaluation :
  $\{\!|$ `e` $|\!\} \rightsquigarrow$ `(`**`let`** `v =` `e` **`in`** `emission_traces_post(); v)`

- Le cas où l'instrumentation trace l'avant et l'après évaluation :
  $\{\!|$ `e` $|\!\} \rightsquigarrow$ `(emission_traces_pre()` **`;`** **`let`** `v =` `e` **`in`** `emission_traces_post(); v)`

- Et enfin le cas où on veut en plus savoir si une exception est levée pendant l'évaluation :
  $\{\!|$ `e` $|\!\} \rightsquigarrow$ `(emission_traces_pre()` **`;`**
  `let` `v =` **`try`** `e` **`with`** `x ->` `emission_traces_exception(x)` **`in`**
  `emission_traces_post(); v)`

En conséquence, toutes les expressions sont tracées de la même manière, ce qui permet un raisonnement formel simple et général.

V.1.2 Instrumentation simple

Par rapport à l'instrumentation simpliste, ici nous allons faire attention à ne pas faire d'instrumentation manifestement inutile.

Notamment :

- l'évaluation des constantes et des variables ne lève jamais d'exception et termine toujours, il n'y a donc pas besoin de double marqueur pour ces expressions ;
- le cas est similaire pour la création des fermetures, on ne placera donc pas non plus de double marqueur, mais on pensera bien à instrumenter sa sous-expression selon sa nature ;
- la réussite de la construction d'une valeur d'un type somme ne dépend que de son argument, donc l'instrumentation de ce dernier suffit ;
- la réussite de l'évaluation d'une séquence revient à la réussite de l'évaluation de ses sous-expressions ;
- l'affectation, si elle se produit, termine toujours avec la valeur `()`, ses sous-expressions doivent cependant être instrumentées ;
- etc.

En fait, pour toutes les expressions non atomiques, sauf l'application, la réussite de l'évaluation ne dépend que de la réussite des évaluations des sous-expressions. On peut alors obtenir une instrumentation qui reste simple mais qui fait moins de « calculs inutiles ».

Nous proposons ci-après une nouvelle définition de la fonction $^{i}[\![\ ]\!]$, tout en gardant les propositions précédentes de définitions de la fonction $\{\!|\ |\!\}$.

Dans la suite, l'émission de trace sera notée $\star$.

Remarques

- Les $\star$ sont posées pour les valeurs immédiates (*i.e.*, pas de calcul) atomiques (constantes et variables) et fonctionnelles (construction **fun**), ainsi que pour les retours des boucles.
- $\{\!|\ |\!\}$ n'est utilisée que pour les applications, permettant ainsi de savoir si elles ont été évaluées en des valeurs.
- Le reste du temps, on parcourt simplement l'expression en profondeur avec la fonction ${}^{i}[\![\ ]\!]$.

```
i[[ x ]] ⇝  ( ★ ; x )
i[[ c ]] ⇝  ( ★ ; c )
i[[ e1 e2 ]] ⇝ {| i[[ e1 ]] i[[ e2 ]] |}
i[[ fun x -> e ]] ⇝  ( ★ ; fun x -> i[[ e ]] )
i[[ if e1 then e2 else e3 ]] ⇝  if i[[ e1 ]] then i[[ e2 ]] else i[[ e3 ]]
i[[ let x = e1 in e2 ]] ⇝  let x = i[[ e1 ]] in i[[ e2 ]]
i[[ let rec x1 = fun x2 -> e1 in e2 ]] ⇝ let rec x1 = fun x2 -> i[[ e1 ]] in i[[ e2 ]]
i[[ C(e) ]] ⇝ C( i[[ e ]] )
i[[ e1 ; e2 ]] ⇝ i[[ e1 ]] ; i[[ e2 ]]
i[[ { f1 = e1 ; ... ; fn = en } ]] ⇝  { f1 = i[[ e1 ]] ; ... ; fn = i[[ en ]] }
i[[ e . f ]] ⇝  i[[ e ]] . f
i[[ e1 . f <- e2 ]] ⇝  i[[ e1 ]] . f <- i[[ e2 ]]
i[[ for x = e1 to e2 do e3 done ]] ⇝ for x = i[[ e1 ]] to i[[ e2 ]] do i[[ e3 ]] done  ; ★
i[[ while e1 do e2 done ]] ⇝ while i[[ e1 ]] do i[[ e2 ]] done  ; ★
i[[ e1 && e2 ]] ⇝  i[[ e1 ]] && i[[ e2 ]]
i[[ e1 || e2 ]] ⇝  i[[ e1 ]] || i[[ e2 ]]
i[[ not e ]] ⇝ not i[[ e ]]
i[[ (e) ]] ⇝ ( i[[ e ]] )
i[[ match e with p1 -> e1 | ... | pn -> en ]] ⇝
   match i[[ e ]] with p1 -> i[[ e1 ]] | ... | pn -> i[[ en ]]
i[[ raise e ]] ⇝ {| raise i[[ e ]] |}
i[[ try e1 with x -> e2 ]] ⇝ try i[[ e1 ]] with x -> i[[ e2 ]]
```

V.1.3 Instrumentation avancée pour la couverture structurelle

Dans cette section, on présente une instrumentation qui servira à montrer de manière un peu plus binaire la satisfaction de la couverture structurelle. En ce sens, si la couverture est parfaite, donc à un taux de 100%, alors la mesure sera équivalente aux mesures plus riches (*i.e.*, les mesures qui émettent davantage de traces présentées dans V.1.1 et V.1.2). En revanche, si la couverture n'est pas de 100%, alors cela sera mis en évidence mais avec moins de précision qu'avec les autres mesures. Le but premier ici est d'instrumenter le moins possible afin limiter le ralentissement de l'exécution de programmes instrumentés.

L'algorithme proposé ici veut minimiser l'utilisation des instructions de productions de traces sans être « excessivement » compliqué. Le but de cette minimisation est d'éviter une instrumentation trop coûteuse sur le plan des performances. En effet, contrôler la bonne évaluation et les levées d'exceptions pour une simple variable engendre une perte de performance très grande : au lieu d'un accès en une petite opération, on ajoute la pose d'un contrôle d'exception, l'enregistrement de traces pré et post exécution, alors qu'on sait d'avance que l'évaluation d'une variable n'engendrera jamais de levée

d'exception et qu'elle se passera toujours bien du début à la fin.

En fait, l'algorithme présenté dans cette section est basé sur celui présenté en 2008 dans l'article de Pagano *et al.* [66].

Notations. On propose deux fonctions d'exploration de l'arbre de syntaxe. La première, notée ${}^{i}[\![\ ]\!]$, instrumente les expressions en les marquant, tandis que la seconde, notée $\{\!|\ |\!\}$, instrumente les sous-expressions des expressions déjà marquées. L'instrumentation d'une sous-expression sera déléguée selon le contexte soit à la première fonction, soit à la seconde, selon qu'on voudra imposer la présence d'un marqueur ou non et selon le contexte.

L'idée générale est de reprendre la sémantique de la couverture des expressions ainsi que la sémantique du langage présentées dans le chapitre précédent et de bien distinguer le critère « est couvert » de la propriété « s'est évalué en une valeur. » Ainsi, on évitera de placer une marque sur une expression lorsqu'il existe forcément une marque ailleurs permettant de confirmer qu'elle s'est bien évaluée comme attendu.

Les expressions atomiques, que sont les variables et les constantes, sont toujours couvertes si elles sont atteintes. Un marqueur placé juste avant leur évaluation permet de montrer leur couverture. Si on sait déduire qu'elles ont été évaluées, on s'abstient de poser de marqueurs. C'est le cas par exemple si on met une expression atomique en tête d'une séquence : si la dernière expression de la séquence est couverte, alors l'atome qui se situe en début est forcément couvert car on n'a pas pu l'ignorer.

${}^{i}[\![$ `atom` $]\!] \rightsquigarrow$ `( ★ ; atom )`

$\{\!|$ `atom` $|\!\} \rightsquigarrow$ `atom`

Un exemple plus parlant est celui de la conditionnelle, que nous traitons dans le paragraphe suivant.

La conditionnelle est une séquence particulière avec un branchement. La condition est forcément évaluée avant d'évaluer l'une des branches. Par exemple, dans l'expression
`if x then` $\mathtt{e}_1$ **`else`** $\mathtt{e}_2$, si $\mathtt{e}_1$ ou $\mathtt{e}_2$ est couvert, alors `x` est forcément couvert et n'a donc pas besoin de marqueur pour montrer la couverture de `x`. Si ni $\mathtt{e}_1$ ni $\mathtt{e}_2$ ne sont couverts, alors `x` n'est pas non plus couvert. Mais si on remplace l'atome `x` par une expression générale `e`, alors on peut simplement manquer d'informations selon la nature de `e`. Ainsi, on va déléguer l'instrumentation de la condition à la seconde fonction $\{\!|\ |\!\}$. Les conséquences et les alternatives doivent toujours être instrumentées via la première fonction ${}^{i}[\![\ ]\!]$ car elles pourraient sinon comporter du code mort marqué comme étant vivant, à cause du branchement. Un exemple très simple est lorsque nous avons une expression comme
`(` **`if`** `(` **`if`** $\mathtt{x}_1$ **`then`** $\mathtt{x}_2$ **`else`** $\mathtt{x}_3$ `)` **`then`** $\mathtt{x}_4$ **`else`** $\mathtt{x}_5$ `)`
où les $\mathtt{x}_1$ `...` $\mathtt{x}_5$ sont des variables : $\mathtt{x}_2$ et $\mathtt{x}_3$ doivent avoir chacun des marqueurs pour ne pas avoir de faux-positifs.

${}^{i}[\![$ **`if`** $\mathtt{e}_1$ **`then`** $\mathtt{e}_2$ **`else`** $\mathtt{e}_3$ $]\!] \rightsquigarrow$ **`if`** $\{\!|\ \mathtt{e}_1\ |\!\}$ **`then`** ${}^{i}[\![\ \mathtt{e}_2\ ]\!]$ **`else`** ${}^{i}[\![\ \mathtt{e}_3\ ]\!]$

$\{\!|$ **`if`** $\mathtt{e}_1$ **`then`** $\mathtt{e}_2$ **`else`** $\mathtt{e}_3$ $|\!\} \rightsquigarrow$ **`if`** $\{\!|\ \mathtt{e}_1\ |\!\}$ **`then`** ${}^{i}[\![\ \mathtt{e}_2\ ]\!]$ **`else`** ${}^{i}[\![\ \mathtt{e}_3\ ]\!]$

L'application existe sous deux formes :

- L'application « diverge obligatoirement », elle est repérable par son type car son type de retour est une variable quantifiée universellement qui n'apparaît pas à gauche de la flèche. D'après les propriétés du système de types, on sait que soit elle lève une exception, soit elle ne termine pas [2].
- L'application rend peut-être une valeur. On peut distinguer deux catégories de valeurs :
  1. les valeurs de type **unit**
  2. et les valeurs d'autres types.

  Dans le premier cas, on peut placer un marqueur après l'application sans changer le type de la valeur de l'application. Dans le second, il faut stocker la valeur de l'application dans une variable, puis invoquer le marqueur avant de rendre la valeur affectée à la variable.

$e_1\ e_2$ est de type **unit** :
${}^{i}[\![\ e_1\ e_2\ ]\!] \rightsquigarrow (\ \lbag e_1 \rbag \lbag e_2 \rbag\ ;\ \star\ )$
$e_1\ e_2$ ne rendra pas de valeur :
${}^{i}[\![\ e_1\ e_2\ ]\!] \rightsquigarrow (\ \star\ ;\ \lbag e_1 \rbag \lbag e_2 \rbag\ )$
Cas particulier du raise :
${}^{i}[\![\ \texttt{raise}\ e\ ]\!] \rightsquigarrow (\ \star\ ;\ \texttt{raise}\ \lbag e \rbag\ )$
$\lbag\ \texttt{raise}\ e\ \rbag \rightsquigarrow (\ \texttt{raise}\ \lbag e \rbag\ )$
Autres cas :
${}^{i}[\![\ e_1\ e_2\ ]\!] \rightsquigarrow (\ \texttt{let r =}\ \lbag e_1 \rbag \lbag e_2 \rbag\ \texttt{in}\ \star\ ;\ r\ )$

La fonction $\lbag\ \rbag$ est définie de la même manière pour toutes les applications :
$\lbag\ e_1\ e_2\ \rbag \rightsquigarrow (\ \lbag e_1 \rbag \lbag e_2 \rbag\ )$

Remarque : Lorsqu'un appel est la dernière action effectuée dans un corps de fonction, on dit qu'il est en position terminale. Le résultat rendu par un appel terminal est le résultat que rendra la fonction qui contient l'appel. Une optimisation pour ce genre d'appels consiste à réutiliser immédiatement l'espace de la pile qui a été occupé par la fonction appelante puisqu'elle n'en aura plus besoin. L'instrumentation que nous proposons pour les appels en général fait que les appels terminaux ne sont plus en position terminale (puisqu'on doit revenir pour faire une action). Ainsi, un programme qui fonctionne bien en version normale peut ne plus fonctionner en version instrumentée du fait de sa consommation potentiellement beaucoup plus importante de la pile.

Les fonctions sont un cas assez particulier du fait qu'à leur création, leur corps est « gelé » (*i.e.*, non évalué) jusqu'à l'application. On va plutôt avoir tendance à définir leur couverture comme le fait que leur corps est bien couvert. (L'évaluation de la création de la fermeture ne donne aucune information sur la couverture de son corps, alors que la couverture de son corps implique la couverture

2. Remarque : on ne prend pas en compte ici les fonctions dont le type n'est pas calculé par l'algorithme de typage mais imposé par le programmeur, comme c'est le cas pour les fonctions externes. En OCaml, ces fonctions sont par exemple la fonction de désérialisation `Marshal.from_channel : in_channel -> 'a` ou la fonction de « dé-typage » `Obj.magic : 'a -> 'b`. Cependant, rien n'empêche d'interdire l'usage de telles fonctions si le contexte est sensible. On peut éventuellement utiliser des fonctions de désérialisation monomorphes si cela s'avère nécessaire ou bien utiliser un mécanisme de désérialisation sûre [40, 39].

de la création de la fermeture.) Dans tous les cas, le corps de la fermeture doit posséder au moins un marqueur, sinon on ne peut pas savoir s'il est couvert.
$^{i}[\![$ **fun** x **->** e $]\!] \rightsquigarrow$ **fun** x **->** $^{i}[\![$ e $]\!]$
$\lbag$ **fun** x **->** e $\rbag \rightsquigarrow$ **fun** x **->** $^{i}[\![$ e $]\!]$

Remarque : on ne gêne pas les optimisations de décurryfication.

La séquence nécessitant un marqueur verra son marqueur à la fin, puisque la couverture de la dernière expression implique la couverture des expressions précédentes. Lorsqu'on ne nécessite pas de marqueur, la séquence est une construction qui n'en imposera pas (contrairement à la conditionnelle par exemple).
$^{i}[\![\ e_1$ **;** $e_2\ ]\!] \rightsquigarrow \lbag e_1 \rbag$ **;** $^{i}[\![e_2]\!]$
$\lbag e_1$ **;** $e_2 \rbag \rightsquigarrow \lbag e_1 \rbag$ **;** $\lbag e_2 \rbag$

Les enregistrements sont des valeurs un peu particulières : si on a bien spécifié l'ordre d'évaluation pour le langage mCaml, la plupart des langages laissent aux créateurs de compilateur la liberté de choisir l'ordre d'évaluation. On va donc prendre en compte cette propriété malgré tout, donc en supposant que l'ordre d'évaluation des différentes composantes des enregistrements n'est pas spécifié.

Avec l'ordre d'évaluation spécifié, de gauche à droite, on peut simplifier en donnant une version analogue à la séquence, où la couverture du dernier suffit à indiquer que les expressions précédentes ont été évaluées en des valeurs :
$^{i}[\![$ **{** f_1 **=** e_1 **;** ... **;** f_m **=** e_m **;** f_n **=** e_n **}** $]\!] \rightsquigarrow$
{ f_1 **=** $\lbag e_1 \rbag$ **;** ... **;** f_m **=** $\lbag e_m \rbag$ **;** f_n **=** $^{i}[\![e_n]\!]$ **}**

Avec l'ordre d'évaluation non spécifié, on place le marqueur à la fin de l'évaluation de l'expression, ce qui affecte davantage le visuel du code mais donne finalement bien la sémantique que nous voulons.
$^{i}[\![$ **{** f_1 **=** e_1 **;** ... **;** f_n **=** e_n **}** $]\!] \rightsquigarrow$
(**let** r **=** **{** f_1 **=** $\lbag e_1 \rbag$ **;** ... **;** f_n **=** $\lbag e_n \rbag$ **}** **in** $\star$ **;** r **)**
$\lbag$ **{** f_1 **=** e_1 **;** ... **;** f_n **=** e_n **}** $\rbag \rightsquigarrow$ **{** f_1 **=** $\lbag e_1 \rbag$ **;** ... **;** f_n **=** $\lbag e_n \rbag$ **}**

La boucle conditionnelle est analogue à la conditionnelle (**if**). La condition est forcément évaluée si on termine la boucle. L'évaluation du corps implique l'évaluation de la condition mais la non-évaluation du corps n'implique rien sur la condition (à part qu'elle a été évaluée à fausse ou bien n'a pas du tout été évaluée).
$^{i}[\![$ **while** e_1 **do** e_2 **done** $]\!] \rightsquigarrow$ **(** **while** $\lbag e_1 \rbag$ **do** $^{i}[\![e_2]\!]$ **done ;** $\star$ **)**
$\lbag$ **while** e_1 **do** e_2 **done** $\rbag \rightsquigarrow$ **while** $\lbag e_1 \rbag$ **do** $^{i}[\![e_2]\!]$ **done**

La boucle bornée est assez analogue à la boucle conditionnelle, sauf qu'elle ne peut pas être la cause directe d'une évaluation infinie.

$^{i}[\![$ **for** x = e_1 **to** e_2 **do** e_3 **done** $]\!] \rightsquigarrow$ (**for** x = $\lbag e_1 \rbag$ **to** $\lbag e_2 \rbag$ **do** $^{i}[\![e_3]\!]$ **done** ; $\star$)

$\lbag$ **for** x = e_1 **to** e_2 **do** e_3 **done** $\rbag \rightsquigarrow$ **for** x = $\lbag e_1 \rbag$ **to** $\lbag e_2 \rbag$ **do** $^{i}[\![e_3]\!]$ **done**

Les déclarations locales sont analogues aux séquences. La différence se situe dans l'affectation d'une valeur à une variable.

$^{i}[\![$ **let** x = e_1 **in** e_2 $]\!] \rightsquigarrow$ **let** x = $\lbag e_1 \rbag$ **in** $^{i}[\![e_2]\!]$

$\lbag$ **let** x = e_1 **in** e_2 $\rbag \rightsquigarrow$ **let** x = $\lbag e_1 \rbag$ **in** $\lbag e_2 \rbag$

Si la déclaration locale est récursive, on combine deux définitions précédentes : celle de la déclaration locale générale et celle de la fonction. On obtient ainsi :

$^{i}[\![$ **let** x_1 = **fun** x_2 -> e_1 **in** e_2 $]\!] \rightsquigarrow$ **let** x_1 = **fun** x_2 -> $^{i}[\![e_1]\!]$ **in** $^{i}[\![e_2]\!]$

$\lbag$ **let** x_1 = **fun** x_2 -> e_1 **in** e_2 $\rbag \rightsquigarrow$ **let** x_1 = **fun** x_2 -> $^{i}[\![e_1]\!]$ **in** $\lbag e_2 \rbag$

Ce qui s'écrit comme suit pour les déclarations récursives.

$^{i}[\![$ **let rec** x_1 x_2 = e_1 **in** e_2 $]\!] \rightsquigarrow$ **let rec** x_1 x_2 = $^{i}[\![e_1]\!]$ **in** $^{i}[\![e_2]\!]$

$\lbag$ **let rec** x_1 x_2 = e_1 **in** e_2 $\rbag \rightsquigarrow$ **let rec** x_1 x_2 = $^{i}[\![e_1]\!]$ **in** $\lbag e_2 \rbag$

Le contrôle des exceptions ressemble à la conditionnelle, en ce sens qu'il s'agit d'une expression à deux branches.

$^{i}[\![$ **try** e_1 **with** x -> e_2 $]\!] \rightsquigarrow$ **try** $^{i}[\![e_1]\!]$ **with** x -> $^{i}[\![e_2]\!]$

$\lbag$ **try** e_1 **with** x -> e_2 $\rbag \rightsquigarrow$ **try** $^{i}[\![e_1]\!]$ **with** x -> $^{i}[\![e_2]\!]$

Le filtrage par motifs ressemble à une conditionnelle dont on a généralisé le nombre de branches en passant de 2 à $n > 0$.

$^{i}[\![$ **match** e **with** p_1 -> e_1 | ... | p_n -> e_n $]\!] \rightsquigarrow$
match $\lbag$ e $\rbag$ **with** p_1 -> $^{i}[\![e_1]\!]$ | ... | p_n -> $^{i}[\![e_n]\!]$

$\lbag$ **match** e **with** p_1 -> e_1 | ... | p_n -> e_n $\rbag \rightsquigarrow$
match $\lbag$ e $\rbag$ **with** p_1 -> $^{i}[\![e_1]\!]$ | ... | p_n -> $^{i}[\![e_n]\!]$

L'accès en lecture d'un champ d'enregistrement est couvert si l'expression rendant la valeur de l'enregistrement est couverte.

$^{i}[\![$ e . f $]\!] \rightsquigarrow$ $^{i}[\![$ e $]\!]$. f

$\lbag$ e . f $\rbag \rightsquigarrow \lbag$ e $\rbag$. f

L'accès en écriture d'un champ d'enregistrement est couverte si l'expression rendant la valeur de l'enregistrement est couverte et l'expression rendant la valeur à affecter est couverte.

Si l'évaluation se fait bien de gauche à droite.

$^{i}[\![$ `e1 . f <- e2` $]\!] \rightsquigarrow \{\!|$ `e1` $|\!\}$ `. f <-` $^{i}[\![$ `e2` $]\!]$
$\{\!|$ `e1 . f <- e2` $|\!\} \rightsquigarrow \{\!|$ `e1` $|\!\}$ `. f <-` $\{\!|$ `e2` $|\!\}$

Quand l'ordre d'évaluation est inconnu, il suffit de placer un marqueur à la fin lorsqu'il y en a besoin, sans forcer leur présence dans les sous-expressions.
$^{i}[\![$ `e1 . f <- e2` $]\!] \rightsquigarrow \{\!|$ `e1` $|\!\}$ `. f <-` $\{\!|$ `e2` $|\!\}$ `; ⋆`
$\{\!|$ `e1 . f <- e2` $|\!\} \rightsquigarrow \{\!|$ `e1` $|\!\}$ `. f <-` $\{\!|$ `e2` $|\!\}$

La construction d'une valeur non-constante d'un type somme n'échoue jamais si son argument s'évalue bien. On délègue donc sans ajout ni suppression.
$^{i}[\![$ `C(e)` $]\!] \rightsquigarrow$ `C(` $^{i}[\![$ `e` $]\!]$ `)`
$\{\!|$ `C(e)` $|\!\} \rightsquigarrow$ `C(` $\{\!|$ `e` $|\!\}$ `)`

V.1.3.1 Récapitulatif

La figure V.1 (page 98) reprend l'ensemble des règles pour l'instrumentation qui vient d'être présentée.

Les cas particuliers de cette instrumentation au niveau des applications : intervention du typage

- Lors d'une application `(e1 e2)`, si le type de `e1` est de la forme `∀'a.(t -> 'a)` et `'a∉t`, alors c'est que l'application ne va pas retourner de valeur (elle va soit boucler indéfiniment, soit lever une exception). Dans ce cas, il est ici considéré inutile de surveiller le retour de cette application. *Néanmoins, il pourrait éventuellement être utile de savoir quelle exception a été levée, mais cette information n'est pas considérée comme utile ici.*
- Lorsque l'application a un type de retour égal à **`unit`**, alors on procède au marquage sans passer par le stockage de la valeur de retour avant, puisque la valeur de retour serait l'unique valeur de type **`unit`** et que le type de retour du marquage est également **`unit`**.

$^{i}[\![$ atom $]\!] \rightsquigarrow$ ($\star$; atom)
$\lfloor$ atom $\rfloor \rightsquigarrow$ atom
$^{i}[\![$ **if** e_1 **then** e_2 **else** e_3 $]\!] \rightsquigarrow$ **if** $\lfloor e_1 \rfloor$ **then** $^{i}[\![e_2]\!]$ **else** $^{i}[\![e_3]\!]$
$\lfloor$ **if** e_1 **then** e_2 **else** e_3 $\rfloor \rightsquigarrow$ **if** $\lfloor e_1 \rfloor$ **then** $^{i}[\![e_2]\!]$ **else** $^{i}[\![e_3]\!]$
e_1 e_2 est de type **unit** :
$^{i}[\![e_1\ e_2]\!] \rightsquigarrow$ ($\lfloor e_1 \rfloor \lfloor e_2 \rfloor$; $\star$)
e_1 e_2 ne rendra pas de valeur :
$^{i}[\![e_1\ e_2]\!] \rightsquigarrow$ ($\star$; $\lfloor e_1 \rfloor \lfloor e_2 \rfloor$)
Cas particulier du raise :
$^{i}[\![$ **raise** e $]\!] \rightsquigarrow$ ($\star$; **raise** $\lfloor$ e $\rfloor$)
$\lfloor$ **raise** e $\rfloor \rightsquigarrow$ (**raise** $\lfloor$ e $\rfloor$)
Autres cas :
$^{i}[\![e_1\ e_2]\!] \rightsquigarrow$ (**let** r = $\lfloor e_1 \rfloor \lfloor e_2 \rfloor$ **in** $\star$; r)
$\lfloor e_1\ e_2 \rfloor \rightsquigarrow$ ($\lfloor e_1 \rfloor \lfloor e_2 \rfloor$)
$^{i}[\![$ **fun** x -> e $]\!] \rightsquigarrow$ **fun** x -> $^{i}[\![$ e $]\!]$
$\lfloor$ **fun** x -> e $\rfloor \rightsquigarrow$ **fun** x -> $^{i}[\![$ e $]\!]$
$^{i}[\![e_1$; $e_2]\!] \rightsquigarrow \lfloor e_1 \rfloor$; $^{i}[\![e_2]\!]$
$\lfloor e_1$; $e_2 \rfloor \rightsquigarrow \lfloor e_1 \rfloor$; $\lfloor e_2 \rfloor$
$^{i}[\![$ { f_1 = e_1 ; ... ; f_m = e_m ; f_n = e_n } $]\!] \rightsquigarrow$
{ f_1 = $\lfloor e_1 \rfloor$; ... ; f_m = $\lfloor e_m \rfloor$; f_n = $^{i}[\![e_n]\!]$ }
$^{i}[\![$ **while** e_1 **do** e_2 **done** $]\!] \rightsquigarrow$ (**while** $\lfloor e_1 \rfloor$ **do** $^{i}[\![e_2]\!]$ **done** ; $\star$)
$\lfloor$ **while** e_1 **do** e_2 **done** $\rfloor \rightsquigarrow$ **while** $\lfloor e_1 \rfloor$ **do** $^{i}[\![e_2]\!]$ **done**
$^{i}[\![$ **for** x = e_1 **to** e_2 **do** e_3 **done** $]\!] \rightsquigarrow$ (**for** x = $\lfloor e_1 \rfloor$ **to** $\lfloor e_2 \rfloor$ **do** $^{i}[\![e_3]\!]$ **done** ; $\star$)
$\lfloor$ **for** x = e_1 **to** e_2 **do** e_3 **done** $\rfloor \rightsquigarrow$ **for** x = $\lfloor e_1 \rfloor$ **to** $\lfloor e_2 \rfloor$ **do** $^{i}[\![e_3]\!]$ **done**
$^{i}[\![$ **let** x = e_1 **in** e_2 $]\!] \rightsquigarrow$ **let** x = $\lfloor e_1 \rfloor$ **in** $^{i}[\![e_2]\!]$
$\lfloor$ **let** x = e_1 **in** e_2 $\rfloor \rightsquigarrow$ **let** x = $\lfloor e_1 \rfloor$ **in** $\lfloor e_2 \rfloor$
$^{i}[\![$ **let rec** x_1 x_2 = e_1 **in** e_2 $]\!] \rightsquigarrow$ **let rec** x_1 x_2 = $^{i}[\![e_1]\!]$ **in** $^{i}[\![e_2]\!]$
$\lfloor$ **let rec** x_1 x_2 = e_1 **in** e_2 $\rfloor \rightsquigarrow$ **let rec** x_1 x_2 = $^{i}[\![e_1]\!]$ **in** $\lfloor e_2 \rfloor$
$^{i}[\![$ **try** e_1 **with** x -> e_2 $]\!] \rightsquigarrow$ **try** $^{i}[\![e_1]\!]$ **with** x -> $^{i}[\![e_2]\!]$
$\lfloor$ **try** e_1 **with** x -> e_2 $\rfloor \rightsquigarrow$ **try** $^{i}[\![e_1]\!]$ **with** x -> $^{i}[\![e_2]\!]$
$^{i}[\![$ **match** e **with** p_1 -> e_1 | ... | p_n -> e_n $]\!] \rightsquigarrow$
match $\lfloor$ e $\rfloor$ **with** p_1 -> $^{i}[\![e_1]\!]$ | ... | p_n -> $^{i}[\![e_n]\!]$
$\lfloor$ **match** e **with** p_1 -> e_1 | ... | p_n -> e_n $\rfloor \rightsquigarrow$
match $\lfloor$ e $\rfloor$ **with** p_1 -> $^{i}[\![e_1]\!]$ | ... | p_n -> $^{i}[\![e_n]\!]$
$^{i}[\![$ e . f $]\!] \rightsquigarrow {}^{i}[\![$ e $]\!]$. f
$\lfloor$ e . f $\rfloor \rightsquigarrow \lfloor$ e $\rfloor$. f
$^{i}[\![e_1$. f <- $e_2]\!] \rightsquigarrow \lfloor e_1 \rfloor$. f <- $^{i}[\![e_2]\!]$
$\lfloor e_1$. f <- $e_2 \rfloor \rightsquigarrow \lfloor e_1 \rfloor$. f <- $\lfloor e_2 \rfloor$
$^{i}[\![$ C(e) $]\!] \rightsquigarrow$ C($^{i}[\![$ e $]\!]$)
$\lfloor$ C(e) $\rfloor \rightsquigarrow$ C($\lfloor$ e $\rfloor$)

Figure V.1 – (récapitulatif) Instrumentation avancée pour la couverture des expressions

V.1.4 Sémantique opérationnelle préservée

L'appel $\star$ doit se faire indépendamment du reste du programme, c'est-à-dire que les effets de bords effectués ne doivent pas impacter l'exécution du reste du programme.

Voici la sémantique opérationnelle généralisée du marquage : dans un environnement $\mathcal{E}$ et avec une mémoire $\mathcal{M}$, l'évaluation de $\star$ se fait toujours bien et retourne la valeur `()`. La mémoire est changée par effet de bord.

$$\mathcal{E}, \mathcal{M}_1 \vdash \star \rightsquigarrow () \,\big/ \mathcal{M}_2$$

En intégrant cette règle d'évaluation dans le système mCaml, c'est-à-dire en prenant en compte les environnements et mémoire de mCaml, on peut exprimer ainsi la règle pour les expressions atomiques :

$$\frac{\mathcal{E}, \mathcal{M}_1 \vdash \star \ \rightsquigarrow () \,\big/ \mathcal{M}_2 \qquad \mathcal{E}, \mathcal{M}_2 \vdash \texttt{atom} \rightsquigarrow v \big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_1 \vdash \star \ \texttt{;} \ \texttt{atom} \ \rightsquigarrow v \big/ \mathcal{M}_2}$$

En exprimant l'indépendance mémoire du marquage, on obtient ceci :

$$\frac{\mathcal{E}, \mathcal{M}, \mathcal{M}_1 \vdash \star \ \rightsquigarrow () \,\big/ \mathcal{M}, \mathcal{M}_2 \qquad \mathcal{E}, \mathcal{M}, \mathcal{M}_2 \vdash \texttt{atom} \rightsquigarrow v \big/ \mathcal{M}, \mathcal{M}_2}{\mathcal{E}, \mathcal{M}, \mathcal{M}_1 \vdash \star \ \texttt{;} \ \texttt{atom} \ \rightsquigarrow v \big/ \mathcal{M}, \mathcal{M}_2}$$

où $\mathcal{M}$ est la mémoire de l'environnement d'exécution du programme ; $\mathcal{M}_1$ et $\mathcal{M}_2$ sont les états de la mémoire dédiée au marquage. On considère que $\star$ utilise des variables qui ne sont pas utilisées par le programmeur mCaml.

Pour montrer que la sémantique opérationnelle est suffisamment préservée pour l'ensemble du langage d'expressions, il suffit de lui appliquer ce raisonnement par induction.

Par exemple, pour la conditionnelle (**if**) dont la condition s'évalue à **true** :

$$\frac{\mathcal{E}, \mathcal{M}_1, \mathcal{M}_2 \vdash \langle\!| \ \texttt{e}_1 \ |\!\rangle \rightsquigarrow \texttt{true} \big/ \mathcal{M}_3, \mathcal{M}_4 \qquad \mathcal{E}, \mathcal{M}_3, \mathcal{M}_4 \vdash {}^{i}[\![\ \texttt{e}_2 \ ]\!] \rightsquigarrow v \big/ \mathcal{M}_5, \mathcal{M}_6}{\mathcal{E}, \mathcal{M}_1, \mathcal{M}_2 \vdash {}^{i}[\![\ \texttt{if e}_1 \ \texttt{then e}_2 \ \texttt{else e}_3 \ ]\!] \rightsquigarrow v \big/ \mathcal{M}_5, \mathcal{M}_6}$$

V.2 Instrumentation de code mCaml pour la mesure de satisfaction du critère MC/DC

Le critère de mesure MC/DC demande à chacune des conditions d'une décision d'avoir été capable d'influencer le résultat de la décision en gardant les valeurs des autres conditions inchangées. Nous avons donc besoin, pour effectuer cette mesure, de savoir quelles ont été les valeurs prises par les conditions d'une décision ainsi que la valeur prise par la décision. Pour cela, nous avons besoin de statiquement déterminer quelles expressions sont des conditions et quelles expressions sont des décisions, ce qui a été présenté dans le chapitre IV.

On rappelle qu'on choisit la sémantique 4 définie par la fonction $\mathcal{C}_4$.

Dans cette section, nous définissons un schéma de réécriture noté ${}^{i}[\![\ ]\!]$ sur le langage mCaml selon les quatre cas de la combinatoire :

1. expression booléenne hors d'une décision : ${}^{i}[\![\ \texttt{e : bool , } \emptyset \ ]\!]$;

2. expression booléenne dans une décision : ${}^{i}[\![$ `e : bool ,` δ $]\!]$;
3. expression non booléenne dans une décision : ${}^{i}[\![$ `e ,` δ $]\!]$;
4. expression non booléenne hors d'une décision : ${}^{i}[\![$ `e ,` $\emptyset$ $]\!]$,

où δ représente un vecteur de valeurs pour les conditions et $\emptyset$ est un vecteur vide.

${}^{i}[\![\ ]\!]$ décrit donc un algorithme d'implantation de la sémantique $\mathcal{C}_4$, pour la mesure MC/DC.

Détails de notations sur les vecteurs des valeurs prises par les conditions et décisions

Désignation des vecteurs, des cases et sous-vecteurs

- δ est un tableau dont la première case est à l'indice 0.
- $\delta_{x,y}$ est le sous-tableau de δ dont la première case est à l'indice x et la longueur est y. Les cases des tableaux sont partagées avec leurs sous-tableaux.
- $\delta_{x,0}$ est un tableau vide, équivalent à $\emptyset$.
- $\delta_{[x]}$ est la case à l'indice x du tableau δ.
- Si $d = \delta_{x,y}$ et $y \neq 0$ alors $d_{[0]} = \delta_{[x]}$.

Allocation d'un vecteur

- *`newVect(i)`* alloue un vecteur de taille `i` dans l'environnement de stockage des traces.

Mises à jour sur les vecteurs

- *`update(expr,cell)`* est une macro qui selon la localisation géographique de son argument (qu'on ne note pas pour alléger la notation), met à jour la cellule *`cell`* avec la valeur prise par *`expr`* et rend la valeur de celui-ci.
- *`updateD(expr,cell,decision)`* est une macro qui invoque *`update(expr,cell)`* puis qui déclare la donnée *`decision`* comme étant « complète ». Cela veut dire que si les données ne sont pas stockées dans la mémoire du programme, le vecteur est prêt à être transmis puis éliminé.
- *`updateCD`*(${}^{i}[\![$ `e` $]\!]$ `, c1 , c2 , d`) invoque à la fois *`update`* et *`updateD`*.

Remarques sur les vecteurs : C'est une bonne structure de données qui a toutes les propriétés idéales pour la formalisation. Après, en pratique, on peut appliquer des optimisations lors de l'implantation, pour par exemple n'utiliser que des tableaux contigus. *On fournit en annexe, section C.1 (page 182), une implantation en* mCaml *de ce modèle de vecteurs.*

On produira des traces très grandes, au niveau de la formalisation, quitte à ce que l'implantation soit moins naïve, pour économiser la concentration du lecteur par l'allégement de la notation.

Application aux expressions booléennes n'appartenant pas à une décision

Dans la configuration ${}^{i}[\![$ `e ,` $\emptyset$ $]\!]$, si `e` est de type booléen, on a affaire à une décision. Il faut alors créer le vecteur de conditions et décision qui servira à recueillir les informations de couverture dans tous les cas où le critère d'analyse choisi détermine qu'on a affaire à une décision.

${}^{i}[\![$ **`true ,`** $\emptyset$ $]\!] \rightsquigarrow$ **`true`**

${}^{i}[\![$ **`false ,`** $\emptyset$ $]\!] \rightsquigarrow$ **`false`**

Cas d'allocation de nouveaux vecteurs :

$^{i}[\![$ `x , ∅` $]\!] \rightsquigarrow$ **let** $\delta =$ *newVect*(1) **in** (*updateD*(`x ,` $\delta_{[0]}$, δ))

$^{i}[\![$ `e . f , ∅` $]\!] \rightsquigarrow$ **let** $\delta =$ *newVect*(1) **in** (*updateD*($^{i}[\![$ `e` $]\!]$ `. f ,` $\delta_{[0]}$, δ))

$^{i}[\![$ e_1 e_2 `, ∅` $]\!] \rightsquigarrow$ **let** $\delta =$ *newVect*($\gamma+\omega+1$) **in**
(*updateD*($^{i}[\![$ e_1 `,` $\delta_{0,\gamma}$ $]\!]$ $^{i}[\![$ e_2 `,` $\delta_{\gamma,\omega}$ $]\!]$, $\delta_{[\gamma+\omega]}$, δ))
où $\gamma = \#_d($ e_1 $)$; $\omega = \#_d($ e_2 $)$

$^{i}[\![$ e_1 **&&** e_2 `, ∅` $]\!] \rightsquigarrow$ **let** $\delta =$ *newVect*($\gamma+\omega+1$) **in**
(*updateD*($^{i}[\![$ e_1 `,` $\delta_{0,\gamma}$ $]\!]$ **&&** $^{i}[\![$ e_2 `,` $\delta_{\gamma,\omega}$ $]\!]$, $\delta_{[\gamma+\omega]}$, δ))
où $\gamma = \#_d($ e_1 $)$; $\omega = \#_d($ e_2 $)$

$^{i}[\![$ e_1 **||** e_2 `, ∅` $]\!] \rightsquigarrow$ **let** $\delta =$ *newVect*($\gamma+\omega+1$) **in**
(*updateD*($^{i}[\![$ e_1 `,` $\delta_{0,\gamma}$ $]\!]$ **||** $^{i}[\![$ e_2 `,` $\delta_{\gamma,\omega}$ $]\!]$, $\delta_{[\gamma+\omega]}$, δ))
où $\gamma = \#_d($ e_1 $)$; $\omega = \#_d($ e_2 $)$

$^{i}[\![$ **if** e_1 **then** e_2 **else** e_3 `,` δ $]\!] \rightsquigarrow$ **let** $\delta =$ newVect(1+γ+ϕ+ω) **in**
if $^{i}[\![$ e_1 `,` $\delta_{0,\gamma}$ $]\!]$ **then** $^{i}[\![$ e_2 `,` $\delta_{\gamma,\phi}$ $]\!]$ **else** $^{i}[\![$ e_3 `,` $\delta_{\gamma+\phi,\omega}$ $]\!]$
où $\gamma = \#_d($ e_1 $)$; $\phi = \#_d($ e_2 $)$; $\omega = \#_d($ e_3 $)$

Cas où on délègue aux sous-expressions :

$^{i}[\![$ **not** `e , ∅` $]\!] \rightsquigarrow$ **not** $^{i}[\![$ `e , ∅` $]\!]$

$^{i}[\![$ `(e) , ∅` $]\!] \rightsquigarrow$ $^{i}[\![$ `e , ∅` $]\!]$

$^{i}[\![$ e_1 `;` e_2 `, ∅` $]\!] \rightsquigarrow$ $^{i}[\![$ e_1 `, ∅` $]\!]$; $^{i}[\![$ e_2 `, ∅` $]\!]$

$^{i}[\![$ **let rec** x_1 **=** **fun** x_2 **->** e_1 **in** e_2 `, ∅` $]\!] \rightsquigarrow$
let rec x_1 **=** **fun** x_2 **->** $^{i}[\![$ e_1 `, ∅` $]\!]$ **in** $^{i}[\![$ e_2 `, ∅` $]\!]$

$^{i}[\![$ **let** x **=** e_1 **in** e_2 `, ∅` $]\!] \rightsquigarrow$ **let** x **=** $^{i}[\![$ e_1 `, ∅` $]\!]$ **in** $^{i}[\![$ e_2 `, ∅` $]\!]$

$^{i}[\![$ **match** e **with** p_1 **->** e_1 **|** ... **|** p_n **->** e_n `, ∅` $]\!] \rightsquigarrow$
match $^{i}[\![$ `e , ∅` $]\!]$ **with** p_1 **->** $^{i}[\![$ e_1 `, ∅` $]\!]$ **|** .. **|** p_n **->** $^{i}[\![$ e_n `, ∅` $]\!]$

$^{i}[\![$ **try** e_1 **with** x **->** e_2 `, ∅` $]\!] \rightsquigarrow$ **try** $^{i}[\![$ e_1 `, ∅` $]\!]$ **with** x **->** $^{i}[\![$ e_2 `, ∅` $]\!]$
où $\gamma = \#_d($ e_1 $)$; $\omega = \#_d($ e_2 $)$

Ne rendra pas réellement de booléen :

$^{i}[\![$ **raise** e `, ∅` $]\!] \rightsquigarrow$ **raise** $^{i}[\![$ `e , ∅` $]\!]$

Les expressions booléennes appartenant à des décisions sont considérées comme des conditions de la décision. Dans le cas de l'application à retour booléen, le résultat de l'application est bien une condition de la décision, mais l'ensemble de l'application forme une nouvelle décision mesurée indépendamment de la première, conformément à la sémantique. Le détail se trouve plus loin.

$^{i}[\![$ **true** `,` δ $]\!] \rightsquigarrow$ **true**

$^{i}[\![$ **false** `,` δ $]\!] \rightsquigarrow$ **false**

$^{i}[\![$ `x ,` δ $]\!] \rightsquigarrow$ *update*(`x ,` $\delta_{[0]}$)

$^{i}[\![$ `e . f ,` δ $]\!] \rightsquigarrow$ *update*($^{i}[\![$ `e ,` $\delta_{0,\gamma}$ $]\!]$ `. f ,` $\delta_{[\gamma]}$) où $\gamma = \#_d($ e $)$

Conformément à $\mathcal{C}_4$ mais aussi à $\mathcal{C}_3$, lorsqu'on rencontre une application de type booléen qui appartient à une décision, l'ensemble de l'application est une condition de la décision, mais les booléens qui composent les sous-expressions de l'application ne sont pas rattachés à la décision. Cependant, pour les prendre en compte, l'ensemble de l'application est considéré en plus comme une nouvelle décision, et permet ainsi de rattacher les sous-expressions booléennes en tant que conditions de celle-ci. Il faut donc créer un nouveau vecteur.

$^{i}[\![\ \mathtt{e_1\ e_2}\ ,\ \delta\ ]\!] \rightsquigarrow$ **let** $\delta 2$ = *newVect*$(\gamma+\omega+1)$ **in**
(*updateCD*($^{i}[\![\ \mathtt{e_1}\ ,\ \delta 2_{0,\gamma}\ ]\!]\ ^{i}[\![\ \mathtt{e_2}\ ,\ \delta 2_{\gamma,\omega}\ ]\!]\ ,\ \delta 2_{[\gamma+\omega]}\ ,\ \delta_{[0]}\ ,\ \delta 2$))
où γ = $\#_d$($\mathtt{e_1}$); ω = $\#_d$($\mathtt{e_2}$)

Pour les expressions suivantes, la mise à jour des valeurs prises est déléguée à l'instrumentation des sous-expressions.

$^{i}[\![\ \mathtt{e_1}$ **&&** $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow\ ^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\gamma}\ ]\!]$ **&&** $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\gamma,\omega}\ ]\!]$ où γ = $\#_d$($\mathtt{e_1}$); ω = $\#_d$($\mathtt{e_2}$)

$^{i}[\![\ \mathtt{e_1}$ **||** $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow\ ^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\gamma}\ ]\!]$ **||** $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\gamma,\omega}\ ]\!]$ où γ = $\#_d$($\mathtt{e_1}$); ω = $\#_d$($\mathtt{e_2}$)

$^{i}[\![$ **not** $\mathtt{e}\ ,\ \delta\ ]\!] \rightsquigarrow$ **not** $^{i}[\![\ \mathtt{e}\ ,\ \delta\ ]\!]$

$^{i}[\![\ \mathtt{(\ e\ )}\ ,\ \delta\ ]\!]$ -> $(^{i}[\![\ \mathtt{e}\ ,\ \delta\ ]\!])$

$^{i}[\![\ \mathtt{e_1}$; $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow\ ^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\omega_1}\ ]\!]$; $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\omega_1+\omega_2}\ ]\!]$ où ω_1 = $\#_d$($\mathtt{e_1}$) ; ω_2 = $\#_d$($\mathtt{e_2}$)

$^{i}[\![$ **if** $\mathtt{e_1}$ **then** $\mathtt{e_2}$ **else** $\mathtt{e_3}\ ,\ \delta\ ]\!] \rightsquigarrow$ **if** $^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\gamma}\ ]\!]$ **then** $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\gamma,\phi}\ ]\!]$ **else** $^{i}[\![\ \mathtt{e_3}\ ,\ \delta_{\gamma+\phi,\omega}\ ]\!]$
où γ = $\#_d$($\mathtt{e_1}$) ; ϕ = $\#_d$($\mathtt{e_2}$) ; ω = $\#_d$($\mathtt{e_3}$)

$^{i}[\![$ **let rec** $\mathtt{x_1}$ = **fun** $\mathtt{x_2}$ -> $\mathtt{e_1}$ **in** $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow$
let rec $\mathtt{x_1}$ = **fun** $\mathtt{x_2}$ -> $^{i}[\![\ \mathtt{e_1}\ ,\ \emptyset\ ]\!]$ **in** $^{i}[\![\ \mathtt{e_2}\ ,\ \delta\ ]\!]$

$^{i}[\![$ **let** $\mathtt{x}$ = $\mathtt{e_1}$ **in** $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow$ **let** $\mathtt{x}$ = $^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\gamma}\ ]\!]$ **in** $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\gamma,\omega}\ ]\!]$
où γ = $\#_d$($\mathtt{e_1}$); ω = $\#_d$($\mathtt{e_2}$)

$^{i}[\![$ **match** $\mathtt{e}$ **with** $\mathtt{p_1}$ -> $\mathtt{e_1}$ | ... | $\mathtt{p_n}$ -> $\mathtt{e_n}\ ,\ \delta\ ]\!] \rightsquigarrow$
match $^{i}[\![\ \mathtt{e}\ ,\ \emptyset\ ]\!]$ **with** $\mathtt{p_1}$ -> $^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\omega_1}\ ]\!]$ | .. | $\mathtt{p_n}$ -> $^{i}[\![\ \mathtt{e_n}\ ,\ \delta_{\omega_1+\ldots+\omega_m,\omega_n}\ ]\!]$
où ω_1 = $\#_d$($\mathtt{e_1}$) ; ... ; ω_n = $\#_d$($\mathtt{e_n}$)

$^{i}[\![$ **raise** $\mathtt{e}\ ,\ \delta\ ]\!] \rightsquigarrow$ **raise** $^{i}[\![\ \mathtt{e}\ ,\ \delta\ ]\!]$

$^{i}[\![$ **try** $\mathtt{e_1}$ **with** $\mathtt{x}$ -> $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow$ **try** $^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\gamma}\ ]\!]$ **with** $\mathtt{x}$ -> $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\gamma,\omega}\ ]\!]$
où γ = $\#_d$($\mathtt{e_1}$); ω = $\#_d$($\mathtt{e_2}$)

Les expressions non booléennes appartenant à une décision transmettent l'appartenance à une décision à leurs sous-expressions quand la sémantique le demande (ce qui le cas général, à part quelques exceptions : par exemple, on ne transmet pas la décision à un corps de fonction). À chaque étape, on calcule le nombre de conditions dans la sous-expression pour lui transmettre seuls les liens nécessaires avec la décision. S'il n'y a aucune condition dans une sous-expression, on ne lui transmet aucune information.

$^{i}[\![\ \mathtt{c}\ ,\ \delta\ ]\!] \rightsquigarrow \mathtt{c}$

$^{i}[\![\ \mathtt{x}\ ,\ \delta\ ]\!] \rightsquigarrow \mathtt{x}$

$^{i}[\![\ \mathtt{e}\ .\ \mathtt{f}\ ,\ \delta\ ]\!] \rightsquigarrow\ ^{i}[\![\ \mathtt{e}\ ,\ \delta\ ]\!]\ .\ \mathtt{f}$

$^{i}[\![\ \mathtt{e_1}\ .\ \mathtt{f}$ <- $\mathtt{e_2}\ ,\ \delta\ ]\!] \rightsquigarrow\ ^{i}[\![\ \mathtt{e_1}\ ,\ \delta_{0,\gamma}\ ]\!]\ .\ \mathtt{f}$ <- $^{i}[\![\ \mathtt{e_2}\ ,\ \delta_{\gamma,\omega}\ ]\!]$
où γ = $\#_d$($\mathtt{e_1}$); ω = $\#_d$($\mathtt{e_2}$)

$^i[\![\ \texttt{e}_1\ \texttt{e}_2\ ,\ \delta\ ]\!] \rightsquigarrow {}^i[\![\ \texttt{e}_1\ ,\ \delta_{0,\gamma}\ ]\!]\ {}^i[\![\ \texttt{e}_2\ ,\ \delta_{\gamma,\omega}\ ]\!]$ où $\gamma = \#_d(\ \texttt{e}_1\ )\ ;\ \omega = \#_d(\ \texttt{e}_2\ )$

$^i[\![\ \texttt{(e)}\ ,\ \delta\ ]\!] \rightsquigarrow (^i[\![\ \texttt{e}\ ,\ \delta\ ]\!])$

$^i[\![\ \texttt{C(e)}\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{C}(^i[\![\ \texttt{e}\ ,\ \delta\ ]\!])$

$^i[\![\ \texttt{\{}\ \texttt{f}_1 = \texttt{e}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \texttt{f}_n = \texttt{e}_n\ \texttt{\}}\ ,\ \delta\ ]\!] \rightsquigarrow$
$\texttt{\{}\ \texttt{f}_1 = {}^i[\![\ \texttt{e}_1\ ,\ \delta_{0,\omega_1}\ ]\!]\ \texttt{;}\ \ldots\ \texttt{;}\ \texttt{f}_n = {}^i[\![\ \texttt{e}_n\ ,\ \delta_{\omega_1+\ldots+\omega_m,\omega_n}\ ]\!]\ \texttt{\}}$
où $\omega_1 = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \ldots\ \texttt{;}\ \omega_n = \#_d(\ \texttt{e}_n\ )$

$^i[\![\ \texttt{e}_1\ \texttt{;}\ \texttt{e}_2\ ,\ \delta\ ]\!] \rightsquigarrow {}^i[\![\ \texttt{e}_1\ ,\ \delta_{0,\omega_1}\ ]\!]\ \texttt{;}\ {}^i[\![\ \texttt{e}_2\ ,\ \delta_{\omega_1+\omega_2}\ ]\!]$ où $\omega_1 = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \omega_2 = \#_d(\ \texttt{e}_2\ )$

$^i[\![\ \texttt{let rec}\ \texttt{x}_1\ \texttt{=}\ \texttt{fun}\ \texttt{x}_2\ \texttt{->}\ \texttt{e}_1\ \texttt{in}\ \texttt{e}_2\ ,\ \delta\ ]\!] \rightsquigarrow$
$\texttt{let rec}\ \texttt{x}_1\ \texttt{=}\ \texttt{fun}\ \texttt{x}_2\ \texttt{->}\ {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{in}\ {}^i[\![\ \texttt{e}_2\ ,\ \delta\ ]\!]$

$^i[\![\ \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{e}_1\ \texttt{in}\ \texttt{e}_2\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{let}\ \texttt{x}\ \texttt{=}\ {}^i[\![\ \texttt{e}_1\ ,\ \delta_{0,\gamma}\ ]\!]\ \texttt{in}\ {}^i[\![\ \texttt{e}_2\ ,\ \delta_{\gamma,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

$^i[\![\ \texttt{fun}\ \texttt{x}\ \texttt{->}\ \texttt{e}\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{fun}\ \texttt{x}\ \texttt{->}\ {}^i[\![\ \texttt{e}\ ,\ \emptyset\ ]\!]$

$^i[\![\ \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{if}\ {}^i[\![\ \texttt{e}_1\ ,\ \delta_{0,\gamma}\ ]\!]\ \texttt{then}\ {}^i[\![\ \texttt{e}_2\ ,\ \delta_{\gamma,\phi}\ ]\!]\ \texttt{else}\ {}^i[\![\ \texttt{e}_3\ ,\ \delta_{\gamma+\phi,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \phi = \#_d(\ \texttt{e}_2\ )\ \texttt{;}\ \omega = \#_d(\ \texttt{e}_3\ )$

$^i[\![\ \texttt{match}\ \texttt{e}\ \texttt{with}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n\ ,\ \delta\ ]\!] \rightsquigarrow$
$\texttt{match}\ {}^i[\![\ \texttt{e}\ ,\ \delta_{0,\gamma}\ ]\!]\ \texttt{with}\ \texttt{p}_1\ \texttt{->}\ {}^i[\![\ \texttt{e}_1\ ,\ \delta_{\gamma,\omega_1}\ ]\!]\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ {}^i[\![\ \texttt{e}_n\ ,\ \delta_{\gamma+\omega_1+\ldots+\omega_m,\omega_n}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}\ )\ \texttt{;}\ \omega_1 = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \ldots\ \texttt{;}\ \omega_n = \#_d(\ \texttt{e}_n\ )$

$^i[\![\ \texttt{raise}\ \texttt{e}\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{raise}\ {}^i[\![\ \texttt{e}\ ,\ \delta\ ]\!]$

$^i[\![\ \texttt{try}\ \texttt{e}_1\ \texttt{with}\ \texttt{x}\ \texttt{->}\ \texttt{e}_2\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{try}\ {}^i[\![\ \texttt{e}_1\ ,\ \delta_{0,\gamma}\ ]\!]\ \texttt{with}\ \texttt{x}\ \texttt{->}\ {}^i[\![\ \texttt{e}_2\ ,\ \delta_{\gamma,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

$^i[\![\ \texttt{while}\ \texttt{e}_1\ \texttt{do}\ \texttt{e}_2\ \texttt{done}\ ,\ \delta\ ]\!] \rightsquigarrow \texttt{while}\ {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{do}\ {}^i[\![\ \texttt{e}_2\ ,\ \emptyset\ ]\!]\ \texttt{done}$

$^i[\![\ \texttt{for}\ \texttt{x}\ \texttt{=}\ \texttt{e}_1\ \texttt{to}\ \texttt{e}_2\ \texttt{do}\ \texttt{e}_3\ \texttt{done}\ ,\ \delta\ ]\!] \rightsquigarrow$
$\texttt{for}\ \texttt{x}\ \texttt{=}\ {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{to}\ {}^i[\![\ \texttt{e}_2\ ,\ \emptyset\ ]\!]\ \texttt{do}\ {}^i[\![\ \texttt{e}_3\ ,\ \emptyset\ ]\!]\ \texttt{done}$

Les expressions non booléennes n'appartenant pas à une décision sont instrumentées selon un simple parcours en profondeur.

$^i[\![\ \texttt{c}\ ,\ \emptyset\ ]\!] \rightsquigarrow \texttt{c}$

$^i[\![\ \texttt{x}\ ,\ \emptyset\ ]\!] \rightsquigarrow \texttt{x}$

$^i[\![\ \texttt{e . f}\ ,\ \emptyset\ ]\!] \rightsquigarrow {}^i[\![\ \texttt{e}\ ,\ \emptyset\ ]\!]\ \texttt{. f}$

$^i[\![\ \texttt{e}_1\ \texttt{. f <-}\ \texttt{e}_2\ ,\ \emptyset\ ]\!] \rightsquigarrow {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{. f <-}\ {}^i[\![\ \texttt{e}_2\ ,\ \emptyset\ ]\!]$

$^i[\![\ \texttt{e}_1\ \texttt{e}_2\ ,\ \emptyset\ ]\!] \rightsquigarrow {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ {}^i[\![\ \texttt{e}_2\ ,\ \emptyset\ ]\!]$

$^i[\![\ \texttt{(e)}\ ,\ \emptyset\ ]\!] \rightsquigarrow (^i[\![\ \texttt{e}\ ,\ \emptyset\ ]\!])$

$^i[\![\ \texttt{C(e)}\ ,\ \emptyset\ ]\!] \rightsquigarrow \texttt{C}(^i[\![\ \texttt{e}\ ,\ \emptyset\ ]\!])$

$^i[\![\ \texttt{\{}\ \texttt{f}_1 = \texttt{e}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \texttt{f}_n = \texttt{e}_n\ \texttt{\}}\ ,\ \emptyset\ ]\!] \rightsquigarrow \texttt{\{}\ \texttt{f}_1 = {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{;}\ \ldots\ \texttt{;}\ \texttt{f}_n = {}^i[\![\ \texttt{e}_n\ ,\ \emptyset\ ]\!]\ \texttt{\}}$

$^i[\![\ \texttt{e}_1\ \texttt{;}\ \texttt{e}_2\ ,\ \emptyset\ ]\!] \rightsquigarrow {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{;}\ {}^i[\![\ \texttt{e}_2\ ,\ \emptyset\ ]\!]$

$^i[\![\ \texttt{let rec}\ \texttt{x}_1\ \texttt{=}\ \texttt{fun}\ \texttt{x}_2\ \texttt{->}\ \texttt{e}_1\ \texttt{in}\ \texttt{e}_2\ ,\ \emptyset\ ]\!] \rightsquigarrow$
$\texttt{let rec}\ \texttt{x}_1\ \texttt{=}\ \texttt{fun}\ \texttt{x}_2\ \texttt{->}\ {}^i[\![\ \texttt{e}_1\ ,\ \emptyset\ ]\!]\ \texttt{in}\ {}^i[\![\ \texttt{e}_2\ ,\ \emptyset\ ]\!]$

$$^{i}[\![\ \texttt{let x = } e_1 \texttt{ in } e_2 \texttt{ , } \emptyset\ ]\!] \rightsquigarrow \texttt{let x = } {}^{i}[\![\ e_1 \texttt{ , } \emptyset\ ]\!] \texttt{ in } {}^{i}[\![\ e_2 \texttt{ , } \emptyset\ ]\!]$$

$$^{i}[\![\ \texttt{fun x -> e , } \emptyset\ ]\!] \rightsquigarrow \texttt{fun x -> } {}^{i}[\![\ \texttt{e , } \emptyset\ ]\!]$$

$$^{i}[\![\ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ , } \emptyset\ ]\!] \rightsquigarrow \texttt{if } {}^{i}[\![\ e_1 \texttt{ , } \emptyset\ ]\!] \texttt{ then } {}^{i}[\![\ e_2 \texttt{ , } \emptyset\ ]\!] \texttt{ else } {}^{i}[\![\ e_3 \texttt{ , } \emptyset\ ]\!]$$

$$^{i}[\![\ \texttt{match e with } p_1 \texttt{ -> } e_1 \texttt{ | ... | } p_n \texttt{ -> } e_n \texttt{ , } \emptyset\ ]\!] \rightsquigarrow \texttt{match } {}^{i}[\![\ \texttt{e , } \emptyset\ ]\!] \texttt{ with } p_1 \texttt{ -> } {}^{i}[\![\ e_1 \texttt{ , } \emptyset\ ]\!] \texttt{ | ... | } p_n \texttt{ -> } {}^{i}[\![\ e_n \texttt{ , } \emptyset\ ]\!]$$

$$^{i}[\![\ \texttt{raise e , } \emptyset\ ]\!] \rightsquigarrow \texttt{raise } {}^{i}[\![\ \texttt{e , } \emptyset\ ]\!]$$

$$^{i}[\![\ \texttt{try } e_1 \texttt{ with x -> } e_2 \texttt{ , } \emptyset\ ]\!] \rightsquigarrow \texttt{try } {}^{i}[\![\ e_1 \texttt{ , } \emptyset\ ]\!] \texttt{ with x -> } {}^{i}[\![\ e_2 \texttt{ , } \emptyset\ ]\!]$$

$$^{i}[\![\ \texttt{while } e_1 \texttt{ do } e_2 \texttt{ done , } \emptyset\ ]\!] \rightsquigarrow \texttt{while } {}^{i}[\![\ e_1 \texttt{ , } \emptyset\ ]\!] \texttt{ do } {}^{i}[\![\ e_2 \texttt{ , } \emptyset\ ]\!] \texttt{ done}$$

$$^{i}[\![\ \texttt{for x = } e_1 \texttt{ to } e_2 \texttt{ do } e_3 \texttt{ done , } \emptyset\ ]\!] \rightsquigarrow \texttt{for x = } {}^{i}[\![\ e_1 \texttt{ , } \emptyset\ ]\!] \texttt{ to } {}^{i}[\![\ e_2 \texttt{ , } \emptyset\ ]\!] \texttt{ do } {}^{i}[\![\ e_3 \texttt{ , } \emptyset\ ]\!] \texttt{ done}$$

V.3 Conclusion du chapitre

L'objectif est atteint : on peut effectuer les mesures pour MC/DC, par réécriture du code source. On fournit en annexe C.1 (page 182) une implantation en mCaml d'une bibliothèque d'exécution pour la gestion des traces : pour avoir un programme instrumenté, il suffit de « coller » cette bibliothèque devant le programme réécrit.

Cette technique est assez rapide à mettre en œuvre, puisqu'il suffit d'implanter une transformation assez simple sur les expressions. Le prix à payer est potentiellement fort, puisque le programme instrumenté donne un binaire plus gros (puisqu'on lui demande de faire des actions supplémentaires), ce qui peut poser problème dans certains environnements. Certaines optimisations du compilateur sont cassées, par exemple les appels terminaux ne sont plus en position terminale, ce qui peut être gênant pour les appels récursifs terminaux.

CHAPITRE VI

Couverture de code structurelle non-intrusive pour mCaml

« *Belle nuit en hiver, jour qui suit souvent couvert.* »
– Proverbe français

Sommaire

VI.1 Instrumentation pour la couverture structurelle simple 108
VI.1.1 Informations de mise au point produites par `ocamldebug` 109
VI.1.2 Génération des marques pour la couverture simple des expressions du langage mCaml 110
VI.2 Instrumentation pour les mesures des conditions et décisions 113
VI.2.1 Instrumentation de l'environnement d'exécution pour la mesure MC/DC 113
VI.2.2 Production de rapports de couverture avec la mesure MC/DC 116
VI.3 Conservation de la sémantique opérationnelle 117
VI.4 Conclusion du chapitre 117

Résumé du chapitre 6

Ce chapitre présente une technique « non intrusive » pour la couverture structurelle de code, avec mesure MC/DC. L'aspect « non intrusif » indique que la technique d'instrumentation ne modifie pas le programme testé. Cette fois-ci, l'instrumentation est effectuée sur l'environnement d'exécution qui va se charger de produire les traces nécessaires à la génération de rapports de couverture. L'avantage et la nouveauté de cette approche sont la possibilité d'utiliser le même binaire pour la production et pour les tests. Bien sûr, d'autres problèmes techniques sont alors rencontrés, et font l'objet de discussion dans ce chapitre.

Objectif et contexte

Objectif. Dans ce chapitre, nous allons présenter une technique de mesure de couverture que nous appelons « non-intrusive » du fait qu'elle ne réécrit pas les programmes, par opposition à la technique présentée dans le chapitre précédent. Les binaires produits sont alors exactement les mêmes que ceux qui sont produits dans un environnement « normal » tel que celui de l'utilisateur final.

Instrumentation de l'environnement d'exécution. C'est l'environnement d'exécution qui se chargera de collecter les informations d'exécution des binaires, plutôt que le binaire lui-même. L'environnement d'exécution sera alors spécial puisque particulièrement adapté pour la production d'informations d'exécution. Ce qui compose alors l'environnement d'exécution est dans notre cas une machine virtuelle. Cela aurait pu être un ensemble de matériels permettant de faire des mesures, mais nous nous intéressons ici aux approches logicielles.

L'approche de la machine virtuelle. Les machines virtuelles sont des logiciels, donc en quelque sorte plus faciles à manipuler que les machines réelles. On peut plus facilement ajouter ou retirer du matériel virtuel sans risque de griller des composants, de même que multiplier du matériel virtuel a un coût proche de zéro par rapport à la multiplication du matériel réel.

Informations de correspondance entre le code source et le binaire. Pour ce faire, des informations de correspondance entre le code source et le binaire sont produites statiquement par le compilateur et mises dans un fichier à part. Ces données serviront lors d'une exécution destinée à mesurer la couverture du programme ainsi que lors de la génération du rapport de couverture après l'exécution.

Correspondance entre codes sources et binaires. Des informations de correspondance entre les binaires et leurs codes sources sont traditionnellement utilisées par les outils de mise au point (ou *debug*), par exemple lors de l'exécution pas à pas ou pour marquer des points d'arrêts. On peut citer dans ce domaine les outils « GDB : The GNU Project Debugger » [1] qui permet par exemple d'exécuter des programmes en mode pas-à-pas et marquer des points d'arrêt. Il existe différents formats pour représenter ces informations de correspondance, dont le standard DWARF [2] pour les langages procéduraux (*e.g.*, C, C++, Fortran).

Informations de mise au point pour OCaml. Pour OCaml, la distribution standard fournit l'outil `ocamldebug` (assez similaire à GDB au niveau de l'interface utilisateur) pour les programmes compilés en code octet. Les informations de correspondance avec le source sont une sérialisation [3] de données OCaml. Ces informations sont ajoutées dans une section indépendante de l'exécutable binaire produit, sans influer en quoi que ce soit les autres parties du binaire.

1. `http://www.gnu.org/software/gdb/`
2. `http://www.dwarfstd.org/`
3. La sérialisation consiste à encoder les données d'un graphe mémoire de sorte à pouvoir l'écrire dans un fichier ou le transmettre sur le réseau. La désérialisation consiste à lire des données sérialisées et reconstruire le graphe mémoire correspondant.

Insuffisance des informations de mise au point. Cependant, les informations de correspondance produites pour `ocamldebug` ne sont pas complètes, au sens où elles ne sont pas suffisantes pour produire les rapports de couverture avec toutes les mesures qui nous intéressent. Certaines parties du code ont alors un statut qui ne peut être qu'inconnu en ce qui concerne leur couverture.

D'autre part, il est également nécessaire de toutes manières de produire des informations supplémentaires sur les expressions du code source pour pouvoir établir les mesures spécifiques aux critères de couverture conditionnelles (*i.e.*, les prises de valeurs des conditions et/ou des décisions). Sans ces informations, l'environnement d'exécution serait contraint de garder toutes les traces de l'exécution du programme (*i.e.*, tous les états de l'exécution) pour une analyse post-exécution, ce qui est possible mais pose des problèmes de taille des traces d'exécution.

Enrichissement des compilateurs pour produire davantage d'informations. XCOV est un outil de couverture non intrusive de code pour le langage Ada utilisant le compilateur GNAT [4]. Il a été développé dans le cadre du projet Couverture, et utilise la machine virtuelle QEMU [5] [6][3] pour engendrer les traces d'exécution[10].

Le même constat d'insuffisance des informations de mise au point a été fait lors de son élaboration. Le compilateur GNAT a alors été enrichi pour pouvoir produire davantage d'informations de correspondance entre le code source et le code machine pour permettre la réalisation des mesures par l'outil XCOV.

Zamcov est un ensemble d'outils pour la mesure de couverture de code structurelle pour OCaml, adoptant l'approche non intrusive. Tout comme XCOV, l'outil Zamcov a été développé dans le cadre du projet Couverture.

Suite de ce chapitre

Dans la suite de ce chapitre, nous verrons l'instrumentation de l'environnement d'exécution pour la couverture structurelle des expressions, puis l'instrumentation de l'environnement d'exécution pour la mesure MC/DC en accord avec les deux chapitres précédents.

VI.1 Instrumentation pour la couverture structurelle simple

La mesure de couverture structurelle des expressions est une étape à franchir avant de nous intéresser aux mesures de couvertures à critères conditionnels. Ainsi, cette section présente la production des traces d'exécution permettant d'établir cette première mesure.

Les rapports de couverture de code structurelle devront être équivalents à ceux produits par l'approche intrusive présentée dans le chapitre précédent, mais cette fois-ci sans réécriture du code source du programme afin de pouvoir utiliser le même binaire pour les activités de tests et pour la production.

Pour ce faire, nous avons besoin d'informations sur le binaire produit par le compilateur afin de pouvoir faire correspondre le flot d'exécution dans la machine avec le flot d'exécution du code source.

Ces informations doivent être produites par le compilateur (ou bien par un programme connaissant bien le schéma de compilation du compilateur, ce que nous assimilerons abusivement au compilateur).

4. GNAT est le compilateur pour le langage Ada, du projet GNU. `http://www.gnu.org/software/gnat/gnat.html`
5. QEMU est un émulateur de processeur et un virtualisateur. Son instrumentation permet d'engendrer des informations sur les programmes qui sont exécutés en son sein. `http://www.qemu.org`.

VI.1.1 Informations de mise au point produites par ocamldebug

ocamlc et ocamldebug. Pour OCaml, le programme `ocamldebug` utilise les informations fournies par le compilateur `ocamlc` lorsque l'option `-g` est activée. Ces informations peuvent être suffisantes pour établir un rapport de couverture structurelle.

En fait, certaines informations ne sont pas disponibles et doivent être déduites du contexte si possible, à défaut il faut faire sans. Par exemple, cette option de compilation ne supprime pas l'optimisation des appels terminaux et ne gère que la correspondance entre le code source et son code machine correspondant. Les appels terminaux n'ayant pas d'instruction marquant le retour, on ne peut pas simplement se baser sur les adresses des instructions pour savoir quand se passe le retour.

Les applications et les fonctions tiennent compte des optimisations de décurryfications du compilateur. L'expression `(x1 x2 x3)` est en fait composée de deux applications, c'est-à-dire le résultat de `(x1 x2)` appliqué à `x3`. La variable `x1` a pu être définie ainsi par exemple :
`let x1 = fun a -> fun b -> e`
Pour éviter d'effectuer une double application, le compilateur peut utiliser une seule application en considérant `x2` et `x3` comme les deux arguments de `x1`.

Liens entre code source et code octet. Lors de l'exécution du programme compilé, il faut pouvoir faire le lien entre « l'instruction en train d'être interprétée » et « l'endroit dans le code source qui l'a engendrée ».

On peut représenter la genèse de ces liens à l'aide de marques * décorant le code source. En revanche, ces marques ne serviront qu'à engendrer les liens dans un conteneur d'information indépendant du code binaire (sinon la technique serait intrusive).

La figure VI.1 (page 109) montre le placement des marques sur le code source par `ocamlc` lorsque l'option `-g` est invoquée.

- On note ⟦ e ⟧ la fonction récursive de placement des marques appliquée à `e`.
- ⋆ représente une marque. Il est important de savoir que les marques sont placées en dehors du programme, donc le code machine avec ou sans marques est exactement le même. Dans `ocamldebug`, les marques sont représentées par `<|b|>` ou `<|a|>` selon qu'elles soient placées avant ou après une expression (`<|b|>` pour *before* et `<|a|>` pour *after*). Dans la documentation d'OCaml [52, chapitre 16], elles sont représentées par le symbole §§ et appelées « *debugger events* ». La documentation souligne l'absence de marque après les appels terminaux.

⟦ atom ⟧	=	atom
⟦ e_1 e_2 ⟧	=	⟦ e_1 ⟧ ⟦ e_2 ⟧ ⋆
⟦ **let** x = e_1 **in** e_2 ⟧	=	**let** x = ⟦ e_1 ⟧ **in** ⋆ ⟦ e_2 ⟧
⟦ **let rec** x_1 x_2 = e_1 **in** e_2 ⟧	=	**let rec** x_1 x_2 = ⋆ ⟦ e_1 ⟧ **in** ⋆ ⟦ e_2 ⟧
⟦ **fun** x -> e_1 ⟧	=	**fun** x -> ⋆ ⟦ e_1 ⟧
⟦ **match** e_1 **with** p_i -> e_2 ⟧	=	**match** ⟦ e_1 ⟧ **with** p_i -> ⋆ ⟦ e_2 ⟧
⟦ **try** e_1 **with** e -> e_2 ⟧	=	**try** ⟦ e_1 ⟧ **with** e -> ⋆ ⟦ e_2 ⟧
⟦ e_1 ; e_2 ⟧	=	⟦ e_1 ⟧ ; ⋆ ⟦ e_2 ⟧
⟦ **if** e_1 **then** e_2 **else** e_3 ⟧	=	**if** ⟦ e_1 ⟧ **then** ⋆ ⟦ e_2 ⟧ **else** ⋆ ⟦ e_3 ⟧
⟦ **while** e_1 **do** e_2 **done** ⟧	=	**while** ⟦ e_1 ⟧ **do** ⋆ ⟦ e_2 ⟧ **done**
⟦ **for** i = e_1 **to** e_2 **do** e_3 **done** ⟧	=	**for** i = ⟦ e_1 ⟧ **to** ⟦ e_2 ⟧ **do** ⋆ ⟦ e_3 ⟧ **done**

FIGURE VI.1 – Placement des marques par `ocamldebug`

Les informations de la figure VI.1 (page 109) sont suffisantes pour établir une couverture structu-

relle des expressions. C'est-à-dire qu'un environnement d'exécution instrumenté pour se servir de ces informations est capable d'engendrer les traces d'exécution nécessaires à l'établissement d'un rapport de couverture structurelle des expressions.

En revanche, nous ne pouvons pas obtenir de couverture des conditions en raison d'un manque d'informations, car il n'est pas possible de reconnaître les valeurs booléennes avec ces informations.

On peut également remarquer que les expressions atomiques (variables et constantes) sont parfois laissées sans marque. Pour une activité de mise au point, cela n'est pas forcément gênant puisque ces expressions n'engendrent pas de calcul.

VI.1.2 Génération des marques pour la couverture simple des expressions du langage mCaml

Il s'agit ici de générer des informations reliant le code machine au code source. Le traitement de ces informations sera entièrement effectué par l'environnement d'exécution. Cela diffère de ce qui a été présenté dans le chapitre précédent, où il fallait instrumenter le code pour qu'il engendre à l'exécution les traces nécessaires pour la génération du rapport.

Toutefois, on choisit de s'inspirer de ce qui a été présenté dans le chapitre précédent pour aller plus directement à l'algorithme choisi.

Des marques englobantes. Contrairement aux informations générées pour `ocamldebug` par `ocamlc`, on ne discriminera pas des marques placées avant et après les expressions : les informations concernent l'ensemble de l'expression sans qu'il ne soit question d'un avant ou d'un après. L'environnement d'exécution instrumenté se chargera d'interpréter comme il le souhaite les informations.

L'algorithme de génération des informations de liaison en fonction de la nature des expressions (*i.e.*, les marques) est inscrit dans la figure VI.2 (page 111). La fonction ${}^{s}[\![\ ]\!]$ parcourt les expressions récursivement pour engendrer les informations de liaison entre le code source et le code machine. La fonction $\{\!|\ |\!\}$ sert à produire les informations pour les expressions pour lesquelles il existe une marque plus loin permettant de déduire qu'elles ont bien été évaluées.

C'est le même principe qui a été utilisé dans le chapitre précédent. On le rappelle avec un exemple.

La conditionnelle est instrumentée de la même manière avec ${}^{s}[\![\ ]\!]$ et $\{\!|\ |\!\}$ car avoir évalué une expression qui se situe après elle (dans l'arbre d'évaluation) ne donne pas d'information sur quelle branche parmi les deux a été effectivement évaluée.

$${}^{s}[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!] \rightsquigarrow \{\!|\ e_1\ |\!\} \cup {}^{s}[\![\ e_2\ ]\!] \cup {}^{s}[\![\ e_3\ ]\!]$$
$$\{\!|\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ |\!\} \rightsquigarrow \{\!|\ e_1\ |\!\} \cup {}^{s}[\![\ e_2\ ]\!] \cup {}^{s}[\![\ e_3\ ]\!]$$

Pour reprendre la notation utilisée précédemment pour `ocamldebug`, on représente la génération d'une information de liaison par une paire d'étoiles $\star$.

Caractéristiques importantes de l'algorithme. Conformément aux spécifications présentées dans le chapitre 3 :

- l'atome seul engendre une information de liaison entre l'expression et le code machine ;
- l'application et les boucles itératives sont les 3 seules autres constructions du langage qui engendrent une information de liaison entre l'expression et le code machine.

$$
\begin{array}{l}
{}^{s}[\![\ \texttt{atom}\ ]\!] \rightsquigarrow \{\star \texttt{atom} \star\} \\
\natural\ \texttt{atom}\ \flat \rightsquigarrow [] \\
{}^{s}[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!] \rightsquigarrow \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \cup {}^{s}[\![\ e_3\ ]\!] \\
\natural\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \cup {}^{s}[\![\ e_3\ ]\!] \\
{}^{s}[\![\ e_1\ e_2\ ]\!] \rightsquigarrow \{\star\ e_1\ e_2\ \star\} \cup \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \\
\natural\ e_1\ e_2\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \\
{}^{s}[\![\ \texttt{fun x ->}\ e\ ]\!] \rightsquigarrow {}^{s}[\![\ e\ ]\!] \\
\natural\ \texttt{fun x ->}\ e\ \flat \rightsquigarrow {}^{s}[\![\ e\ ]\!] \\
{}^{s}[\![\ e_1\ \texttt{;}\ e_2\ ]\!] \rightsquigarrow \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \\
\natural\ e_1\ \texttt{;}\ e_2\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \\
{}^{s}[\![\ \{\ f_1 = e_1\ \texttt{;}\ \ldots\ \texttt{;}\ f_m = e_m\ \texttt{;}\ f_n = e_n\ \}\ ]\!] \rightsquigarrow \natural\ e_1\ \flat \cup \ldots \cup \natural\ e_m\ \flat \cup {}^{s}[\![\ e_n\ ]\!] \\
\natural\ \{\ f_1 = e_1\ \texttt{;}\ \ldots\ \texttt{;}\ f_n = e_n\ \}\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup \ldots \cup \natural\ e_n\ \flat \\
{}^{s}[\![\ \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done}\ ]\!] \rightsquigarrow \{\star \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \star\} \cup \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \\
\natural\ \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done}\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \\
{}^{s}[\![\ \texttt{for x =}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done}\ ]\!] \rightsquigarrow \\
\quad \{\star \texttt{for x =}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} \star\} \cup \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \cup {}^{s}[\![\ e_3\ ]\!] \\
\natural\ \texttt{for x =}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done}\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \cup {}^{s}[\![\ e_3\ ]\!] \\
{}^{s}[\![\ \texttt{let x =}\ e_1\ \texttt{in}\ e_2\ ]\!] \rightsquigarrow \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \\
\natural\ \texttt{let x =}\ e_1\ \texttt{in}\ e_2\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \\
{}^{s}[\![\ \texttt{let rec}\ x_1\ x_2\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ ]\!] \rightsquigarrow {}^{s}[\![\ e_1\ ]\!] \cup {}^{s}[\![\ e_2\ ]\!] \\
\natural\ \texttt{let rec}\ x_1\ x_2\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \flat \rightsquigarrow {}^{s}[\![\ e_1\ ]\!] \cup \natural\ e_2\ \flat \\
{}^{s}[\![\ \texttt{try}\ e_1\ \texttt{with x ->}\ e_2\ ]\!] \rightsquigarrow {}^{s}[\![\ e_1\ ]\!] \cup {}^{s}[\![\ e_2\ ]\!] \\
\natural\ \texttt{try}\ e_1\ \texttt{with x ->}\ e_2\ \flat \rightsquigarrow {}^{s}[\![\ e_1\ ]\!] \cup {}^{s}[\![\ e_2\ ]\!] \\
{}^{s}[\![\ \texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n\ ]\!] \rightsquigarrow \natural\ e\ \flat \cup {}^{s}[\![\ e_1\ ]\!] \cup \ldots \cup {}^{s}[\![\ e_n\ ]\!] \\
\natural\ \texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n\ \flat \rightsquigarrow \natural\ e\ \flat \cup {}^{s}[\![\ e_1\ ]\!] \cup \ldots \cup {}^{s}[\![\ e_n\ ]\!] \\
{}^{s}[\![\ e\ \texttt{.}\ f\ ]\!] \rightsquigarrow {}^{s}[\![\ e\ ]\!] \\
\natural\ e\ \texttt{.}\ f\ \flat \rightsquigarrow \natural\ e\ \flat \\
{}^{s}[\![\ e_1\ \texttt{.}\ f\ \texttt{<-}\ e_2\ ]\!] \rightsquigarrow \natural\ e_1\ \flat \cup {}^{s}[\![\ e_2\ ]\!] \\
\natural\ e_1\ \texttt{.}\ f\ \texttt{<-}\ e_2\ \flat \rightsquigarrow \natural\ e_1\ \flat \cup \natural\ e_2\ \flat \\
{}^{s}[\![\ \texttt{C(e)}\ ]\!] \rightsquigarrow {}^{s}[\![\ e\ ]\!] \\
\natural\ \texttt{C(e)}\ \flat \rightsquigarrow \natural\ e\ \flat
\end{array}
$$

FIGURE VI.2 – Génération d'informations pour la couverture simple des expressions

La génération d'une information de liaison se fait au niveau des expressions étoilées de la figure VI.2.

Une telle information comporte :

- l'emplacement du début et de la fin de l'expression dans le fichier source ;
- l'emplacement du début et de la fin de l'expression dans le code machine correspondant.

VI.1.2.1 Interférences des optimisations du compilateur

Si le compilateur décide d'éliminer du code mort.
Par exemple, le code `let x = if false then do_something() else ()` peut être simplifié par le compilateur en `let x = ()` et si `x` n'est jamais utilisé, ce code peut être carrément supprimé.

En fait, cela ne pose pas de vrai problème du fait que la génération du rapport n'aura pas d'informations indiquant que ce code a été exécuté. Il sera donc considéré comme étant **non couvert** (donc problématique du point de vue de celui qui veut ou a besoin de couvrir l'ensemble du programme).

Si le compilateur décide de pré-évaluer certaines expressions.
Par exemple, le code `let x = 42 + 2` peut être remplacé par `let x = 44`. Là nous avons un potentiel problème de correspondance entre le code source et le code machine. Le code machine ne correspondra pas directement avec le code source d'origine mais on peut toujours considérer que « si le `44` est couvert, alors `42 + 2` est couvert. »

Factorisation de code machine. Si le compilateur décide de factoriser du code machine pour produire un binaire plus compact, nous pouvons avoir de grandes difficultés à savoir quelle partie du code source est effectivement exécutée lors de l'exécution du code machine qui correspond à plusieurs morceaux distincts du code source. C'est une optimisation à interdire, ou bien il faut engendrer davantage d'informations non triviales sur le flot d'exécution du code machine pour en tenir compte à l'exécution.

Intégration (*inlining* en anglais) de code. Si le compilateur décide d'intégrer du code, nous avons deux problèmes :

- si on décide que l'exécution de n'importe quelle version du code machine suffit à indiquer la couverture du code source, alors on a un code source « mieux couvert » que le code machine ;
- si on décide que l'exécution de toutes les versions du code machine doivent être effectives pour que le code source soit couvert, une couverture complète du code machine correspondrait bien à une couverture complète du code source, mais une couverture incomplète mettrait l'optimisation en cause sur la non-couverture du code source, ce qui serait difficile à faire accepter.

Cette optimisation serait alors plutôt à désactiver.

Optimisations du compilateur en général. Il serait plutôt mauvais d'interdire toutes les optimisations du compilateur, ou en tout cas il serait difficile de faire accepter d'une manière générale l'interdiction générale des optimisations du compilateur, ne serait-ce que pour des raisons de performances d'exécution des programmes. En revanche, l'obligation pour les optimisations de garder correctes les informations reliant code source et code machine devrait être acceptable mais est surtout nécessaire pour ne pas indiquer des « faux-négatifs », c'est-à-dire du code non couvert dans le source alors qu'il l'est.

VI.2 Instrumentation pour les mesures des conditions et décisions

Il est plus délicat de mesurer la couverture des conditions et des décisions du fait qu'il faut récupérer les valeurs prises lors de l'exécution. Il ne s'agit plus de se contenter de savoir si des blocs d'instructions sont effectivement interprétés, il faut en plus connaître les valeurs prises en regardant dans les registres et/ou les piles d'exécution (cela dépend du schéma de compilation et de la machine).

Pour la ZAM et avec les programmes compilés avec le compilateur `ocamlc`, il s'agira de lire dans le registre `ACCU` la valeur qui y réside.

> *Des informations détaillées sur le fonctionnement de la ZAM se trouvent en annexe, section B.1 (page 175).*

VI.2.1 Instrumentation de l'environnement d'exécution pour la mesure MC/DC

Pour la mesure MC/DC, il faut engendrer des informations supplémentaires pour que l'environnement d'exécution puisse enregistrer les valeurs prises par les expressions booléennes à l'exécution.

Il s'agira alors de lire dans un registre ou sur la pile la valeur courante correspondant à une expression dans le code source. Avec la machine virtuelle d'OCaml et son schéma de compilation standard, il s'agira de lire la valeur contenue dans le registre `ACCU`.

Les informations pour la mesure MC/DC sont suffisantes pour effectuer les mesures de couverture des conditions et de couverture des décisions, on se concentrera sur MC/DC. En fait, il faudra enregistrer les valeurs contenues dans les vecteurs de conditions composant les décisions. La généricité de l'approche s'étend donc partout où ces notions (condition et décision) partagent la même définition.

Contrairement à la mesure de la couverture des expressions, on distinguera les constantes des variables (alors que précédemment les deux catégories étaient confondues au sein de la catégorie des atomes).

Notation

On reprend la sémantique de l'instrumentation présentée dans le chapitre précédent, section V.2 (page 99), et on l'adapte pour générer les informations nécessaires pour relier les conditions et décisions au code machine. Il s'agit donc cette fois-ci non pas de réécrire le code mais de générer des informations sur le code. On définit alors la fonction ${}^{z}[\![\ ]\!]$ qui rend des informations, pas du code.
`(e , c)` est l'information reliant l'expression `e` à l'information de condition `c`.
δ = *`newVect`*`(e , n)` est l'information indiquant que la décision `e` a besoin d'un vecteur de taille `n` pour contenir les informations sur les valeurs prises par elle-même ainsi que les conditions qui composent `e`. Le vecteur est nommé δ pour pouvoir établir un lien entre ce vecteur et les conditions qui composent la décision `e`.

VI.2.1.1 Expressions booléennes n'appartenant pas à une décision

${}^{z}[\![$ **`true`** `, ∅` $]\!] \rightsquigarrow$ `[]`
${}^{z}[\![$ **`false`** `, ∅` $]\!] \rightsquigarrow$ `[]`

Avec une information de création de vecteur :
${}^{z}[\![$ `x , ∅` $]\!] \rightsquigarrow \left(\star\delta =$ *`newVect`*`(x , 1)`$\star\right) \lhd \left(\star$`x , `$\delta_{[0]}\star\right) \lhd$ `[]`
Lorsque δ est affecté d'un vecteur, c'est qu'il s'agit d'une variable fraîche. Ici, elle n'est pas transmise aux sous-décisions puisqu'il n'y en a pas.

$^{z}[\![\ \texttt{e . f , } \emptyset\ ]\!] \rightsquigarrow \left(\star\delta = \textit{newVect}\texttt{(e . f , 1)}\star\right) \lhd \left(\star\texttt{e . f , } \delta_{[0]}\star\right) \lhd \texttt{[]}$

Dans les trois cas suivants, un vecteur δ est fraîchement créé et des sous-vecteurs sont transmis aux sous-expressions.

$^{z}[\![\ \texttt{e}_1\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow \left(\star\delta = \textit{newVect}\texttt{(e}_1\ \texttt{e}_2\ \texttt{,}\ \gamma+\omega+1\texttt{)}\star\right) \lhd \left(\star\texttt{e}_1\ \texttt{e}_2\ \texttt{,}\ \delta_{[\gamma+\omega]}\star\right) \lhd$
$\quad {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \delta_{0,\gamma}\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \delta_{\gamma,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

$^{z}[\![\ \texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow \left(\star\delta = \textit{newVect}\texttt{(e}_1\ \texttt{\&\&}\ \texttt{e}_2\ \texttt{,}\ \gamma+\omega+1\texttt{)}\star\right) \lhd \left(\star\texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2\ \texttt{,}\ \delta_{[\gamma+\omega]}\star\right) \lhd$
$\quad {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \delta_{0,\gamma}\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \delta_{\gamma,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

$^{z}[\![\ \texttt{e}_1\ \texttt{||}\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow \left(\star\delta = \textit{newVect}\texttt{(e}_1\ \texttt{||}\ \texttt{e}_2\ \texttt{,}\ \gamma+\omega+1\texttt{)}\star\right) \lhd \left(\star\texttt{e}_1\ \texttt{||}\ \texttt{e}_2\ \texttt{,}\ \delta_{[\gamma+\omega]}\star\right) \lhd$
$\quad {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \delta_{0,\gamma}\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \delta_{\gamma,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

Par délégation aux sous-expressions :

$^{z}[\![\ \texttt{not e , } \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e , } \emptyset\ ]\!]$
$^{z}[\![\ \texttt{(e) , } \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e , } \emptyset\ ]\!]$
$^{z}[\![\ \texttt{e}_1\ \texttt{; ... ; e}_n\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \emptyset\ ]\!] \cup \texttt{...} \cup {}^{z}[\![\ \texttt{e}_n\ \texttt{,}\ \emptyset\ ]\!]$
$^{z}[\![\ \texttt{let rec x}_1\ \texttt{=}\ \texttt{fun x}_2\ \texttt{->}\ \texttt{e}_1\ \texttt{in e}_2\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \emptyset\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!]$
$^{z}[\![\ \texttt{let x =}\ \texttt{e}_1\ \texttt{in e}_2\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \emptyset\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!]$
$^{z}[\![\ \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \emptyset\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!] \cup {}^{z}[\![\ \texttt{e}_3\ \texttt{,}\ \emptyset\ ]\!]$
$^{z}[\![\ \texttt{match e with p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \texttt{...}\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow$
$\quad {}^{z}[\![\ \texttt{e}\ \texttt{,}\ \emptyset\ ]\!] \cup {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \emptyset\ ]\!] \cup \texttt{...} \cup {}^{z}[\![\ \texttt{e}_n\ \texttt{,}\ \emptyset\ ]\!]$
$^{z}[\![\ \texttt{raise e , } \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e , } \emptyset\ ]\!]$
$^{z}[\![\ \texttt{try}\ \texttt{e}_1\ \texttt{with x ->}\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \emptyset\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \emptyset\ ]\!]$

VI.2.1.2 Expressions booléennes appartenant à des décisions

Les constantes :

$^{z}[\![\ \texttt{true , } \delta\ ]\!] \rightsquigarrow \texttt{[]}$
$^{z}[\![\ \texttt{false , } \delta\ ]\!] \rightsquigarrow \texttt{[]}$

Le cas où on appartient à une décision et où on découvre une nouvelle décision :

$^{z}[\![\ \texttt{e}_1\ \texttt{e}_2\ \texttt{,}\ \delta\ ]\!] \rightsquigarrow \left(\star\delta' = \textit{newVect}\texttt{(e}_1\ \texttt{e}_2\ \texttt{,}\ \gamma+\omega+1\texttt{)}\star\right) \lhd \left(\star\texttt{e}_1\ \texttt{e}_2\ \texttt{,}\ \delta'_{[\gamma+\omega]}\star\right)$
$\quad \lhd \left(\star\texttt{e}_1\ \texttt{e}_2\ \texttt{,}\ \delta_{[0]}\star\right) \lhd {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \delta'_{0,\gamma}\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \delta'_{\gamma,\omega}\ ]\!]$
où $\gamma = \#_d(\ \texttt{e}_1\ )\ \texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

Les autres cas, où il suffit d'indiquer les mises à jour sur les vecteurs ou de seulement déléguer aux sous-expressions :

$^{z}[\![\ \texttt{x , } \delta\ ]\!] \rightsquigarrow \left(\star\texttt{x , } \delta_{[0]}\star\right)$
$^{z}[\![\ \texttt{e . f , } \delta\ ]\!] \rightsquigarrow \left(\star\texttt{e . f , } \delta_{[\gamma]}\star\right) \lhd {}^{z}[\![\ \texttt{e , } \delta_{0,\gamma}\ ]\!]$ où $\gamma = \#_d(\ \texttt{e}\ )$
$^{z}[\![\ \texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2\ \texttt{,}\ \delta\ ]\!] \rightsquigarrow {}^{z}[\![\ \texttt{e}_1\ \texttt{,}\ \delta_{0,\gamma}\ ]\!] \cup {}^{z}[\![\ \texttt{e}_2\ \texttt{,}\ \delta_{\gamma,\omega}\ ]\!]$ où $\gamma = \#_d(\ \texttt{e}_1\ )\texttt{;}\ \omega = \#_d(\ \texttt{e}_2\ )$

$^z[\![$ e$_1$ || e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_2$, $\delta_{\gamma,\omega}$ $]\!]$ où γ = $\#_d$(e$_1$) ; ω = $\#_d$(e$_2$)

$^z[\![$ **not** e , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ (e) , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ e$_1$; ... ; e$_n$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, δ_{0,ω_1} $]\!] \cup$ ... $\cup {}^z[\![$ e$_n$, $\delta_{\omega_1+\ldots+\omega_m,\omega_n}$ $]\!]$
où ω_1 = $\#_d$(e$_1$) ; ... ; ω_n = $\#_d$(e$_n$)

$^z[\![$ **let rec** x$_1$ = **fun** x$_2$ -> e$_1$ **in** e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\emptyset$ $]\!] \cup {}^z[\![$ e$_2$, δ $]\!]$
$^z[\![$ **let** x = e$_1$ **in** e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_2$, $\delta_{\gamma,\omega}$ $]\!]$
où γ = $\#_d$(e$_1$) ; ω = $\#_d$(e$_2$)

$^z[\![$ **if** e$_1$ **then** e$_2$ **else** e$_3$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_2$, $\delta_{\gamma,\phi}$ $]\!] \cup {}^z[\![$ e$_3$, $\delta_{\gamma+\phi,\omega}$ $]\!]$
où γ = $\#_d$(e$_1$) ; ϕ = $\#_d$(e$_2$) ; ω = $\#_d$(e$_3$)

$^z[\![$ **match** e **with** p$_1$ -> e$_1$ | ... | p$_n$ -> e$_n$, δ $]\!] \rightsquigarrow$
$\quad {}^z[\![$ e , $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_1$, δ_{γ,ω_1} $]\!] \cup$ ... $\cup {}^z[\![$ e$_n$, $\delta_{\gamma+\omega_1+\ldots+\omega_m,\omega_n}$ $]\!]$
où γ = $\#_d$(e); ω_1 = $\#_d$(e$_1$) ; ... ; ω_n = $\#_d$(e$_n$)

$^z[\![$ **raise** e , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ **try** e$_1$ **with** x -> e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, δ $]\!]$; $^z[\![$ e$_2$, δ $]\!]$

VI.2.1.3 Expressions non booléennes appartenant à une décision

Les expressions atomiques sans booléens :

$^z[\![$ c , δ $]\!] \rightsquigarrow$ []
$^z[\![$ x , δ $]\!] \rightsquigarrow$ []

Le cas qui ne propage pas l'appartenance à une décision :

$^z[\![$ **fun** x -> e , δ $]\!] \rightsquigarrow {}^z[\![$ e , $\emptyset$ $]\!]$

Dans les autres cas, on propage les décisions :

$^z[\![$ C(e) , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ { f$_1$ = e$_1$; ... ; f$_n$ = e$_n$ } , δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, δ_{0,ω_1} $]\!] \cup$ ... $\cup {}^z[\![$ e$_n$, $\delta_{\omega_1+\ldots+\omega_m,\omega_n}$ $]\!]$
où ω_1 = $\#_d$(e$_1$) ; ... ; ω_n = $\#_d$(e$_n$)

$^z[\![$ e . f , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ e$_1$ e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_2$, $\delta_{\gamma,\omega}$ $]\!]$
où γ = $\#_d$(e$_1$) ; ω = $\#_d$(e$_2$)

$^z[\![$ (e) , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ e$_1$; ... ; e$_n$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, δ_{0,ω_1} $]\!] \cup$ ... $\cup {}^z[\![$ e$_n$, $\delta_{\omega_1+\ldots+\omega_m,\omega_n}$ $]\!]$
où ω_1 = $\#_d$(e$_1$) ; ... ; ω_n = $\#_d$(e$_n$)

$^z[\![$ **let rec** x$_1$ = **fun** x$_2$ -> e$_1$ **in** e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\emptyset$ $]\!] \cup {}^z[\![$ e$_2$, δ $]\!]$
$^z[\![$ **let** x = e$_1$ **in** e$_2$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_2$, $\delta_{\gamma,\omega}$ $]\!]$
où γ = $\#_d$(e$_1$); ω = $\#_d$(e$_2$)

$^z[\![$ **if** e$_1$ **then** e$_2$ **else** e$_3$, δ $]\!] \rightsquigarrow {}^z[\![$ e$_1$, $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_2$, $\delta_{\gamma,\phi}$ $]\!] \cup {}^z[\![$ e$_3$, $\delta_{\gamma+\phi,\omega}$ $]\!]$
où γ = $\#_d$(e$_1$) ; ϕ = $\#_d$(e$_2$) ; ω = $\#_d$(e$_3$)

$^z[\![$ **match** e **with** p$_1$ -> e$_1$ | ... | p$_n$ -> e$_n$, δ $]\!] \rightsquigarrow$
$\quad {}^z[\![$ e , $\delta_{0,\gamma}$ $]\!] \cup {}^z[\![$ e$_1$, δ_{γ,ω_1} $]\!] \cup$ ... $\cup {}^z[\![$ e$_n$, $\delta_{\gamma+\omega_1+\ldots+\omega_m,\omega_n}$ $]\!]$

où γ = #$_d$(e); ω_1 = #$_d$(e_1) ; ... ; ω_n = #$_d$(e_n)

$^z[\![$ **raise** e , δ $]\!] \rightsquigarrow {}^z[\![$ e , δ $]\!]$

$^z[\![$ **try** e_1 **with** x -> e_2 , δ $]\!] \rightsquigarrow {}^z[\![e_1 , \delta]\!] \cup {}^z[\![e_2 , \delta]\!]$

VI.2.1.4 Expressions non booléennes n'appartenant pas à une décision

Ici on ne fait que parcourir en profondeur à la « recherche » de nouvelles décisions.

$^z[\![$ c , $\emptyset$ $]\!] \rightsquigarrow$ []

$^z[\![$ x , $\emptyset$ $]\!] \rightsquigarrow$ []

$^z[\![$ **fun** x -> e , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![$ e , $\emptyset$ $]\!]$

$^z[\![$ C(e) , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![$ e , $\emptyset$ $]\!]$

$^z[\![$ { f_1 = e_1 ; ... ; f_n = e_n } , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup ... \cup {}^z[\![e_n , \emptyset]\!]$

$^z[\![$ e . f , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![$ e , $\emptyset$ $]\!]$

$^z[\![e_1\ e_2 , \emptyset]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup {}^z[\![e_2 , \emptyset]\!]$

$^z[\![$ (e) , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![$ e , $\emptyset$ $]\!]$

$^z[\![e_1 ; ... ; e_n , \emptyset]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup ... \cup {}^z[\![e_n , \emptyset]\!]$

$^z[\![$ **let rec** x_1 = **fun** x_2 -> e_1 **in** e_2 , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup {}^z[\![e_2 , \emptyset]\!]$

$^z[\![$ **let** x = e_1 **in** e_2 , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup {}^z[\![e_2 , \emptyset]\!]$

$^z[\![$ **if** e_1 **then** e_2 **else** e_3 , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup {}^z[\![e_2 , \emptyset]\!] \cup {}^z[\![e_3 , \emptyset]\!]$

$^z[\![$ **match** e **with** p_1 -> e_1 | ... | p_n -> e_n , $\emptyset$ $]\!] \rightsquigarrow$
$^z[\![e , \emptyset]\!] \cup {}^z[\![e_1 , \emptyset]\!] \cup ... \cup {}^z[\![e_n , \emptyset]\!]$

$^z[\![$ **raise** e , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![$ e , $\emptyset$ $]\!]$

$^z[\![$ **try** e_1 **with** x -> e_2 , $\emptyset$ $]\!] \rightsquigarrow {}^z[\![e_1 , \emptyset]\!] \cup {}^z[\![e_2 , \emptyset]\!]$

VI.2.2 Production de rapports de couverture avec la mesure MC/DC

Dans la boucle d'interprétation de la machine abstraite, à chaque instruction lue, il suffit de chercher dans la table des informations de correspondance avec le code source l'ensemble des actions à effectuer pour émettre les traces voulues, et d'effectuer ces actions. Dans le cas où il n'existe pas d'information de correspondance, on continue tout de suite à l'instruction suivante.

On doit obtenir ainsi les mêmes résultats qu'avec l'approche intrusive, mais cette fois-ci sans avoir réécrit le code.

Différences avec l'instrumentation du code (présentée dans le chapitre précédent).

La création d'un vecteur, qui se faisait dans le code instrumenté, représentée par *newVect*, est désormais déléguée à l'environnement d'exécution. On relie alors le code machine de l'expression avec la taille du vecteur à créer pour recueillir les valeurs prises par les conditions.

Le recueil de la valeur prise par les conditions et décisions est également délégué, de la même manière, à l'environnement d'exécution.

Le prochain chapitre VII fournit une comparaison plus étendue des deux techniques.

VI.3 Conservation de la sémantique opérationnelle

Le code machine (ou code compilé) d'un programme est exactement le même avec ou sans instrumentation non intrusive.

La machine interprétant le code machine fait donc les mêmes opérations avec les mêmes instructions. La différence se situe dans les actions supplémentaires effectuées par la machine lorsqu'elle dispose d'informations de « traçage » (*i.e.*, génération de traces). Ces actions n'influençant pas l'état de la machine (puisqu'elle n'accède aux données de la machine qu'en lecture seule), la sémantique opérationnelle du programme est donc bien préservée.

VI.4 Conclusion du chapitre

Nous avons montré une sémantique de génération d'informations de correspondance entre le code source et le code machine, à destination de l'environnement d'exécution, à la fois pour établir la couverture structurelle simple et la couverture des conditions et décisions. Les mesures effectuées pour les conditions et décisions sont utilisables pour les critères les plus usuels, dont MC/DC.

CHAPITRE VII

Comparaison entre les approches intrusive et non-intrusive

« La vérité c'est comme une couverture trop petite.
Tu peux tirer dessus de tous les côtés, tu auras toujours les pieds froids. »
– Peter Weir

Sommaire

VII.1 Comparaison sémantique des deux techniques 121
VII.1.1 Équivalence des résultats produits par les deux techniques en l'absence de traitement particulier des appels terminaux 121
VII.1.2 Une différence notable des deux approches : les appels terminaux . . . 125
VII.2 Comparaison pratique des approches intrusive et non intrusive 127
VII.2.1 Du point de vue de l'utilisateur 127
VII.2.2 Avantages et inconvénients 128
VII.3 MLcov : couverture structurelle par instrumentation du code source . . . 129
VII.3.1 Caractéristiques générales de l'outil MLcov 129
VII.3.2 Sémantique utilisée par MLcov 130
VII.3.3 Un exemple de rapport de couverture avec MLcov 130
VII.4 Zamcov : couverture structurelle non intrusive 137
VII.4.1 Génération des informations de couverture 137
VII.4.2 Illustration 138
VII.5 Conclusion du chapitre 142

Résumé du chapitre 7

Ce chapitre propose d'abord une comparaison formelle des techniques d'instrumentation intrusive et non-intrusive presentées dans les deux chapitres précédents, puis une comparaison du point de vue de l'utilisateur.

Ensuite, il présente les logiciels MLcov et Zamcov, deux outils pour la couverture structurelle de code, le premier suivant la méthode intrusive et le second suivant la méthode non intrusive.

Nous avons proposé dans les deux chapitres précédents la formalisation du travail à effectuer pour obtenir les traces nécessaires selon les deux approches.

Dans ce chapitre, nous commençons par donner une comparaison sémantique entre les techniques intrusive et non intrusive (section VII.1). Ensuite, nous donnons une comparaison pratique des deux techniques, notamment les avantages et inconvénients de chacune (section VII.2).

Nous présenterons également MLcov, une implantation de l'approche intrusive utilisée industriellement par la société Esterel Technologies (section VII.3), et Zamcov, une implantation de l'approche non intrusive à l'état de prototype (section VII.4).

VII.1 Comparaison sémantique des deux techniques

Cette section présente une première comparaison qui montre que les deux techniques sont équivalentes en terme de sémantique opérationnelle, c'est-à-dire que les deux techniques permettent d'obtenir les mêmes résultats. On pose toutefois la restriction suivante : on ne tient pas compte de la position terminale des appels [1], en ce sens qu'on considère que les appels terminaux sont compilés de la même manière que les appels non terminaux. La raison de cette restriction est qu'elle permet de simplifier le problème essentiellement dû au fait que l'instrumentation intrusive par réécriture du code source rend certains appels terminaux non terminaux.

Une seconde comparaison se concentre précisément sur la restriction de la première en proposant le développement de la problématique des appels terminaux. Du mécanisme de compilation et d'exécution par la machine des appels terminaux, nous verrons les détails qui permettent de comprendre la difficulté et la manière de garder le mécanisme des appels terminaux dans l'instrumentation non intrusive.

VII.1.1 Équivalence des résultats produits par les deux techniques en l'absence de traitement particulier des appels terminaux

Notations.

- $[\![\texttt{P}]\!]$ est le code machine du programme P.
- $\mathcal{I}$(P) est l'instrumentation par réécriture du programme P, dont les détails font l'objet du chapitre V.
- CMI(P) = ($[\![\mathcal{I}(\texttt{P})]\!]$, $\emptyset$) est le résultat de la compilation en code machine du programme P instrumenté.
- $\mathcal{Z}$(P) est les informations de correspondance entre le code source et le code machine ainsi que les informations sur les conditions et décisions : on les appellera « informations de traçage ».
- CMN(P) = ($[\![\texttt{P}]\!]$, $\mathcal{Z}$(P)) est le résultat de la compilation non intrusive.
- (*Résultat* , *Trace*) = $\mathcal{M}([\![\texttt{P}]\!], \mathcal{Z}(\texttt{P}))$, où *Résultat* est le résultat de l'exécution du code machine $[\![\texttt{P}]\!]$ par la machine $\mathcal{M}$, et *Trace* l'ensemble des traces demandées par $\mathcal{Z}$(P) et qui ont pu être obtenues par la machine $\mathcal{M}$.

- Remarque : si P est un programme non instrumenté,
  alors ((*Résultat* , *Trace*) = $\mathcal{M}([\![\texttt{P}]\!], \emptyset)) \Rightarrow$ (*Trace* = $\emptyset$).

1. Un appel est terminal si dans le corps d'une fonction, il s'agit de la dernière action effectuée.

Hypothèses. Soit e une expression, alors on fait l'hypothèse que le compilateur et la machine sont corrects, ce qui peut s'exprimer ainsi :

$$\Big((\texttt{e} \rightsquigarrow v) \wedge ((Rs,\ Tr) = \mathcal{M}([\![\ \texttt{e}\ ]\!], \emptyset))\Big) \Rightarrow (Rs = v)$$

où e $\rightsquigarrow$ v signifie que e s'évalue en v selon la sémantique opérationnelle présentée à la section III.1.2.

On fait également l'hypothèse que la machine $\mathcal{M}$ exécutant le code compilé d'un programme P est bien conforme à la sémantique opérationnelle du programme P telle qu'elle est définie dans ce manuscrit pour l'ensemble du langage mCaml.

Proposition. L'exécution du code machine du programme P instrumenté produit le même résultat et les mêmes traces que l'exécution du code machine du programme P non instrumenté accompagné de ses informations de traçage.

En utilisant les notations précédemment définies, cela s'exprime aussi ainsi
en forme simple : $\mathcal{M}(\texttt{CMI(P)}) = \mathcal{M}(\texttt{CMN(P)})$
en forme développée : $\mathcal{M}([\![\ \mathcal{I}\texttt{(P)}\ ]\!]\ \texttt{,}\ \emptyset) = \mathcal{M}([\![\ \texttt{P}\ ]\!]\ \texttt{,}\ \mathcal{Z}\texttt{(P)})$

Preuve.

Instrumentation intrusive. Il existe plusieurs instances de la fonction $\mathcal{I}$ dans le chapitre IV que nous pouvons résumer ainsi :

- les instrumentations des expressions pour la couverture simple, c'est-à-dire celles qui permettent de savoir si une expression a été évaluée ;
- les instrumentations des conditions et des décisions pour les mesures les faisant intervenir, par exemple la mesure MC/DC.

Il existe dans ces instrumentations trois sortes de placement de marques :

- une marque se trouve avant l'expression, par exemple le cas où on place une marque avant une variable pour savoir si elle a été évaluée ;
- une marque se trouve après l'expression, par exemple le cas où on place une marque après une boucle pour savoir si on en est sorti ;
- deux marques encadrent l'expression, par exemple le cas où on veut connaître les valeurs prises par les conditions d'une décision ainsi que la valeur de la décision elle-même.

En généralisant, nous obtenons :

1. actions pour le recueil d'informations avant évaluation d'une expression ;
2. actions pour le recueil d'informations après évaluation d'une expression ;
3. actions pour le recueil d'informations avant et après évaluation d'une expression.

Lorsque la fonction $\mathcal{I}$ est appliquée à une expression e, selon la nature de l'expression e et son contexte, nous avons les quatre cas de figure suivants :

$\mathcal{I}$(e) = e (aucune action à ajouter)
ou $\mathcal{I}$(e) = $\star_1$; e (actions avant évaluation)
ou $\mathcal{I}$(e) = e ; $\star_2$ (actions après évaluation)
ou $\mathcal{I}$(e) = $\star_1$; e ; $\star_2$ (actions avant et après évaluation)

que nous pouvons compléter par les cas où $\mathcal{I}$ est appelé récursivement sur les sous-expressions (ce qui pourra être le cas pour toutes les expressions non atomiques).

On note que e ; $\star_2$ s'écrit **let** r = e **in** $\star_2$; r lorsqu'on veut récupérer et rendre le résultat de l'évaluation de e, et $\star_1$; e ; $\star_2$ s'écrit **let** r = $\star_1$; e **in** $\star_2$; r lorsque $\star_2$ a besoin de données créées par $\star_1$.

Les actions $\star_1$ et $\star_2$ n'affectent pas l'évaluation de e. Elles accèdent en lecture seule à l'environnement d'évaluation de e ainsi qu'à son état mémoire.

Maintenant, considérons le cas le plus général, c'est-à-dire
let r = $\star_1$; e **in** $\star_2$; r.

Sa compilation ⟦ **let** r = $\star_1$; e **in** $\star_2$; r ⟧
peut être simplifiée en ⟦ $\star_1$ ⟧ ; ⟦ e ⟧ ; ⟦ $\star_2$ ⟧ sous les conditions suivantes :

- les données créées par ⟦ $\star_1$ ⟧ sont bien accessibles depuis ⟦ $\star_2$ ⟧ ;
- ⟦ $\star_2$ ⟧ est un code machine qui restaure la pile et le registre ACCU à la fin de son exécution, tels qu'ils soient identiques à l'avant exécution ; ainsi l'état de la pile et du registre ACCU est le même qu'à la fin de l'exécution de ⟦ e ⟧.

Ainsi, $\mathcal{M}$(⟦ $\star_1$ ⟧ ; ⟦ e ⟧ ; ⟦ $\star_2$ ⟧ , ∅) produit bien le résultat de l'exécution de ⟦ e ⟧ en plus des traces d'exécution attendues.

Par induction sur P, on peut déduire que $\mathcal{M}$(⟦ $\mathcal{I}$(P) ⟧ , ∅) produit bien le résultat de l'exécution de ⟦ P ⟧ ainsi que les traces d'exécution voulues.

Instrumentation non intrusive. Considérons la compilation d'une expression e et ses informations de couverture, c'est-à-dire CMN(e) = (⟦ e ⟧ , $\mathcal{Z}$(e))

L'exécution de CMN(e) revient à exécuter les instructions du code machine ⟦ e ⟧ de l'expression e et de vérifier pour chaque instruction s'il y a une action supplémentaire à effectuer. L'action à effectuer consiste à éventuellement allouer des structures de données pour stocker les informations de traces, et lire l'état des registres ou de la pile [2].

Ainsi, si on reprend l'exemple précédent, où
CMI(e) = (⟦ $\mathcal{I}$(e) ⟧ , ∅) = (⟦ $\star_1$ ⟧ ; ⟦ e ⟧ ; ⟦ $\star_2$ ⟧ , ∅)
on a cette fois-ci CMN(e) = (⟦ e ⟧ , $\mathcal{Z}$(e))
L'exécution $\mathcal{M}$(CMN(e)) = $\mathcal{M}$(⟦ e ⟧ , $\mathcal{Z}$(e)) produira en fait la séquence

1. actions avant évaluation de l'expression e, ce qui correspond bien à $\star_1$;
2. évaluation de l'expression e ;
3. actions après évaluation de l'expression e, ce qui correspond bien à $\star_2$.

ce qui est donc bien la même séquence que ⟦ $\star_1$ ⟧ ; ⟦ e ⟧ ; ⟦ $\star_2$ ⟧.

Par induction sur P, on peut déduire que $\mathcal{M}$(⟦ P ⟧ , $\mathcal{Z}$(P)) produit bien le résultat de l'exécution de ⟦ P ⟧ ainsi que les traces d'exécution voulues.

Conclusion. La comparaison de l'évaluation $\mathcal{M}$(⟦ $\star_1$ ⟧ ; ⟦ e ⟧ ; ⟦ $\star_2$ ⟧ , ∅) avec l'évaluation $\mathcal{M}$(⟦ P ⟧ , $\mathcal{Z}$(P)) est développée dans la figure VII.1, page 124 pour l'ensemble des constructions des expressions du langage mCaml.

Cela permet de conclure : $\mathcal{M}$(⟦ $\mathcal{I}$(P) ⟧ , ∅) = $\mathcal{M}$(⟦ P ⟧ , $\mathcal{Z}$(P))

□

2. Pour la machine virtuelle OCaml, il s'agira de lire la valeur contenue dans le registre ACCU quand on voudra connaître la valeur rendue par l'évaluation d'une expression.

formule générale

$$\mathcal{M}(\,[\![\texttt{e}]\!]\,,\,\mathcal{Z}(\texttt{e})\,) \;=\; \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e})\;]\!]\,,\,\emptyset\,) = \text{Rs}\,,\,\text{Tr}$$

cas terminaux

$$\begin{aligned}\mathcal{M}(\,[\![\texttt{c}]\!]\,,\,(\texttt{c}\leftrightarrow[\![\texttt{c}]\!])\,) &= \mathcal{M}(\,[\![\;\star_1\ \texttt{;}\ \texttt{c}\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{x}]\!]\,,\,(\texttt{x}\leftrightarrow[\![\texttt{x}]\!])\,) &= \mathcal{M}(\,[\![\;\texttt{let r = x in}\ \star_2\ \texttt{;}\ \texttt{r}\;]\!]\,,\,\emptyset\,)\end{aligned}$$

exemple de l'application, avec un niveau de développement supplémentaire

$$\begin{aligned}\mathcal{M}(\,[\![\texttt{e}_1\ \texttt{e}_2]\!]\,,\,\mathcal{Z}(\texttt{e}_1\ \texttt{e}_2)\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e}_1\ \texttt{e}_2)\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{e}_1\ \texttt{e}_2]\!]\,,\{\,\texttt{e}_1\ \texttt{e}_2\leftrightarrow[\![\texttt{e}_1\ \texttt{e}_2]\!]\,;\mathcal{Z}(\texttt{e}_1)\,;\mathcal{Z}(\texttt{e}_2)\}\,) &= \mathcal{M}(\,[\![\;\texttt{let r =}\\ &\qquad \star_1\ \texttt{;}\ (\mathcal{I}(\texttt{e}_1))\ (\mathcal{I}(\texttt{e}_2))\\ &\quad \texttt{in}\ \star_2\ \texttt{;}\ \texttt{r}\;]\!]\,,\,\emptyset\,)\end{aligned}$$

autres cas

$$\begin{aligned}\mathcal{M}(\,[\![\texttt{C(e)}]\!]\,,\,\mathcal{Z}(\texttt{C(e)})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{C(e)})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{fun x -> e}]\!]\,,\,\mathcal{Z}(\texttt{fun x -> e})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{fun x -> e})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{e . f}]\!]\,,\,\mathcal{Z}(\texttt{e . f})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e . f})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{e}_1\ \texttt{. f <- e}_2]\!]\,,\,\mathcal{Z}(\texttt{e}_1\ \texttt{. f <- e}_2)\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e}_1\ \texttt{. f <- e}_2)\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{e}_1\ \texttt{\&\& e}_2]\!]\,,\,\mathcal{Z}(\texttt{e}_1\ \texttt{\&\& e}_2)\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e}_1\ \texttt{\&\& e}_2)\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{e}_1\ \texttt{|| e}_2]\!]\,,\,\mathcal{Z}(\texttt{e}_1\ \texttt{|| e}_2)\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e}_1\ \texttt{|| e}_2)\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{not e}]\!]\,,\,\mathcal{Z}(\texttt{not e})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{not e})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{(e)}]\!]\,,\,\mathcal{Z}(\texttt{(e)})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{(e)})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{raise e}]\!]\,,\,\mathcal{Z}(\texttt{raise e})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{raise e})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{try e}_1\ \texttt{with x -> e}_2]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,[\![\texttt{e}_1\ \texttt{; e}_2]\!]\,,\,\mathcal{Z}(\texttt{e}_1\ \texttt{; e}_2)\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\texttt{e}_1\ \texttt{; e}_2)\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{let x = e}_1\ \texttt{in e}_2]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{let rec x = fun x}_1\ \texttt{-> e}_1\ \texttt{in e}_2]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{\{ f}_1\ \texttt{= e}_1\ \texttt{; } \ldots \texttt{ ; f}_n\ \texttt{= e}_n\ \texttt{\}}]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{for x = e}_1\ \texttt{to e}_2\ \texttt{do e}_3\ \texttt{done}]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{while e}_1\ \texttt{do e}_2\ \texttt{done}]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\\ \mathcal{M}(\,\underbrace{[\![\texttt{match e with p}_1\ \texttt{-> e}_1\ \texttt{| ... |}\ \texttt{p}_n\ \texttt{-> e}_n]\!]}_{\mathcal{E}}\,,\,\mathcal{Z}(\mathcal{E})\,) &= \mathcal{M}(\,[\![\;\mathcal{I}(\mathcal{E})\;]\!]\,,\,\emptyset\,)\end{aligned}$$

où $\mathcal{I}$ est la fonction d'instrumentation intrusive, $\mathcal{Z}$ est la fonction de production d'informations reliant le code source au code machine, $\mathcal{M}$ est la machine qui prend un code compilé et un ensemble d'informations de traçage, et $[\![\ ]\!]$ est la fonction de compilation.

FIGURE VII.1 – Premier niveau de développement pour la preuve par cas sur les expressions du langage avec une induction sur les fonctions d'instrumentation intrusive ($\mathcal{I}$) et non intrusive ($\mathcal{Z}$).

VII.1.2 Une différence notable des deux approches : les appels terminaux

Précédemment, en ayant fait l'hypothèse que les appels terminaux se compilaient de la même manière que les appels non terminaux, la différence entre les deux techniques n'est pas intervenue.

Ici, on va considérer cette optimisation du compilateur et les instructions de la machine virtuelle OCaml spécifiques à cette optimisation.

> *Des informations détaillées sur le fonctionnement de la ZAM se trouvent en annexe, section B.1 (page 175). Les appels terminaux sont définis et leur compilation pour la ZAM est expliquée en annexe, section B.2 (page 178).*

Description du problème. Lors de l'instrumentation intrusive d'un site d'appel, pour savoir si l'appel a rendu un résultat, il est réécrit de manière à récupérer le résultat pour effectuer une action d'émission de traces. Ainsi, plus aucun des appels instrumentés de cette manière ne sont terminaux puisqu'une action est effectuée après eux.

Dans l'approche non intrusive, le code machine n'est pas modifié par rapport au code machine compilé classiquement (aucune instruction n'est changée, ni enlevée, ni ajoutée).

Ainsi, dans la pratique, en présence d'utilisation des instructions spécifiques aux appels terminaux, l'ajout des instructions d'émission de traces pour la technique intrusive change une partie du code qu'on considérait identique entre les deux techniques.

Pour l'équivalence sémantique, cela ne pose pas réellement de problème, car dans tous les cas on obtient les bonnes traces et le bon comportement, toujours du fait de l'hypothèse que le compilateur implante bien la sémantique opérationnelle du langage. En effet, que les appels soient terminaux ou pas, la sémantique opérationnelle de tous les appels reste la même pour le langage. C'est le compilateur et la machine qui prennent à leur charge la préservation de la sémantique s'ils font usage de cette optimisation.

En revanche, pour l'implantation de la technique non intrusive, cela soulève une difficulté.

De manière générale, l'environnement d'exécution se charge de regarder pour chaque instruction interprétée s'il existe une action à effectuer avant ou après l'exécution de l'instruction. Pour les appels non terminaux, il suffit de lire la valeur du registre avant d'exécuter l'instruction qui se trouve à l'adresse suivante qui est toujours ce qu'il faut exécuter si l'appel s'est bien passé (c'est-à-dire que l'appel n'a pas laissé échapper une exception et que la machine ne s'est pas arrêtée). Mais pour les appels terminaux, l'instruction qui suit l'appel ne fait pas partie du code de la fonction courante, on ne peut donc pas compter sur elle pour savoir quand lire la valeur de l'accumulateur.

Solution 1. Pour pallier cela, on peut interdire l'utilisation de l'instruction des appels terminaux. Dans ce cas on obtient la même limitation que pour la version intrusive pour ce qui concerne l'utilisation de la pile pour les appels récursifs terminaux.

Solution 2. On surveille le niveau de la pile (la manière est explicitée plus loin). En effet, lorsque l'appel terminal est terminé et que la valeur attendue se trouve dans l'accumulateur, le niveau de la pile se situe sous le niveau du dernier appel non terminal effectué. Il faut remarquer que si on effectue un appel terminal, c'est que par définition on se trouve dans le corps d'une fonction f, qui elle-même a été appelée, et si l'appel de f était en position terminale, c'est qu'il se trouvait dans un corps de fonction, et ainsi de suite, jusqu'à remonter à un appel non terminal. Il faudra alors lire la valeur du bon registre avant de l'avoir écrasée. Mais s'il est produit une levée d'exception qui a subitement fait

baisser le niveau de pile, alors il faudra savoir que le résultat ne pourra pas être obtenu, ce qui est tout à fait normal et attendu.

Détails de la solution 2

Nous distinguons trois types d'appels de fonctions.

Les appels classiques (instruction `APPLY` et variantes) mettent l'adresse de retour sur la pile pour qu'une fois l'appel terminé, on puisse revenir.

Les appels externes (instruction `CCALL` et variantes) n'ont pas besoin de mettre l'adresse de retour sur la pile car les appels externes n'influent pas sur le pointeur de code, donc l'instruction qui est exécutée après est toujours celle qui se situe juste après. (Dans les appels classiques, l'instruction exécutée juste après n'est pas celle qui la suit puisque le principe est d'exécuter du code qui se trouve ailleurs puis de revenir.)

Les appels terminaux (instruction `APPTERM` et variantes) ont la particularité d'être la dernière instruction exécutée d'un corps de fonction. L'optimisation principale associée aux appels terminaux est de ne pas revenir, ce qui signifie qu'une fois l'appel terminé, on ne revient pas dans le corps de la fonction appelante contrairement aux appels classiques. C'est ce qui permet aux appels récursifs terminaux de ne pas consommer un niveau de pile par appel (ce qui ferait une consommation de la pile en $O(n)$ pour n appels), mais plutôt de consommer une quantité de niveaux de pile indépendante du nombre d'appels [3].

Savoir quand un appel terminal s'est bien terminé de manière générale.
Partant de l'hypothèse qu'un appel terminal est toujours invoqué dans le corps d'une fonction, et qu'en remontant le graphe des appels on tombe toujours sur un appel non terminal, c'est ce dernier qu'il faudra prendre en compte pour savoir si l'appel terminal s'est bien terminé. Pour cela, il suffira de surveiller la pile pour savoir quand est dépilée *l'adresse de retour du plus proche appel non terminal non encore terminé.* En effet, si cette adresse de retour est dépilée et recopiée dans le registre de pointeur de code, alors l'appel s'est bien terminé. Si une exception s'est déclenchée, empêchant l'appel de bien se terminer, alors le mécanisme de gestion des exceptions fera un dépilage brutal sans effectuer cette action, mais comme le niveau de pile sera passé en dessous du niveau surveillé, on saura qu'aucune valeur n'a été rendue. (Et si on a rencontré une boucle infinie, de toutes façons le programme ne termine jamais.)

Récupérer la valeur quand un appel terminal s'est bien terminé est nécessaire lorsqu'on mesure les valeurs prises par les conditions et décisions.

On fait cette fois-ci l'hypothèse que le code machine a bien préservé la sémantique du code source mCaml, notamment le système de types. En effet, le système de types nous garantit que le retour

3. Voici un exemple où on définit une fonction récursive terminale :
```
let rec f n = if n = 0 then n else f (n-1)
```
puis on l'appelle deux fois :
```
let x = f 3
let g = f 1000000
```
Les exécutions des deux appels à `f` consomment exactement la même quantité de pile.

d'appel terminal est du même type que le retour du précédent appel non terminal effectué et non terminé.

Comme on sait quand le retour est effectué, il suffit de récupérer la valeur de l'appel non terminal, qui est également la valeur de retour de l'appel terminal (ou des appels terminaux) en attente de connaître le résultat de retour.

VII.2 Comparaison pratique des approches intrusive et non intrusive

Dans cette section, nous proposons une comparaison du point de vue de l'utilisateur des outils selon les approches intrusives et non intrusives, puis nous dressons un tableau des avantages et inconvénients des deux approches.

VII.2.1 Du point de vue de l'utilisateur

La figure VII.2 schématise l'utilisation des outils selon les approches par instrumentation du code source dite « intrusive » et par instrumentation de l'environnement d'exécution dite « non intrusive », depuis le code source jusqu'au rapport de couverture incluant la mesure MC/DC.

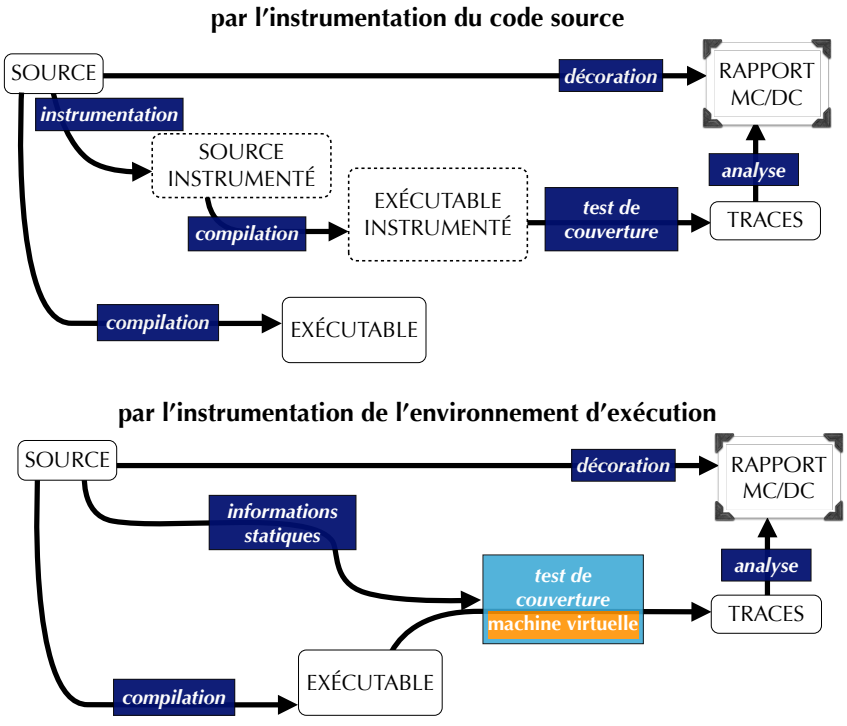


FIGURE VII.2 – Schéma des approches intrusive et non intrusive

L'instrumentation de l'environnement d'exécution permet de simplifier le processus de test en permettant l'utilisation d'un seul et même code machine pour la production et pour les tests, sans devoir les différencier. Ceci utilise l'hypothèse que le client de l'utilisateur ne veut probablement pas que son logiciel produise des traces d'exécution dont il ne saurait quoi faire.

VII.2.2 Avantages et inconvénients

Si on s'interdit certaines optimisations du compilateur qui par exemple éliminent du code par factorisation ou engendrent du code supplémentaire par intégration, l'approche non intrusive permet d'obtenir les mêmes mesures qu'avec l'approche intrusive.

Le tableau VII.1 présente un récapitulatif des caractéristiques des deux techniques.

Approche intrusive	Approche non intrusive
Mise en œuvre	
• réécriture du code source. • bibliothèque de gestion de traces. • exécution du programme instrumenté.	• récupération d'informations sur le source. • bibliothèque de gestion de traces. • machine virtuelle instrumentée. • exécution du programme dans l'environnement instrumenté.
Limitations	
• appels terminaux rétrogradés. • traits complexes du langage impossibles à traiter simplement. • taille du binaire augmentée, peut ne pas convenir à certains environnements d'exécution.	• certaines optimisations du compilateur sont déconseillées voire prohibées (*e.g.*, intégration, factorisation de code).
Avantages	
• simplicité de la mise en œuvre. • ne dépend pas du compilateur. • ne dépend pas de l'architecture.	• pas de binaire spécifique, une seule compilation. • environnement d'exécution facilement paramétrable (car logiciel). • facilité de propagation de l'environnement d'exécution car toujours logiciel.
Inconvénients	
• plusieurs exécutables. • exécutable plus volumineux. • plusieurs compilations. • l'environnement de tests doit avoir les capacités matérielles pour enregistrer les traces d'exécution, ce qui n'est pas toujours le cas lorsqu'on traite des systèmes embarqués.	• instrumentation de l'environnement d'exécution à faire pour chaque cible du compilateur.

Table VII.1 – Caractéristiques principales des techniques intrusives et non intrusive

VII.3 MLcov : couverture structurelle par instrumentation du code source

Dans cette section, nous proposons une description de l'outil MLcov qui adopte une stratégie par instrumentation du code source.

VII.3.1 Caractéristiques générales de l'outil MLcov

MLcov est un outil pour la couverture structurelle pour le langage OCaml. MLcov est capable d'établir la mesure de couverture structurelle simple ainsi que la mesure MC/DC. Il fonctionne par réécriture du code source, en produisant ainsi en bout de chaîne un binaire spécifique contenant les instructions nécessaires pour produire les traces d'exécution permettant d'établir les rapports de couverture structurelle.

MLcov ne gère que la partie la plus usitée du langage OCaml, c'est-à-dire la partie fonctionnelle et impérative avec les modules (simples et paramétrés). Notamment, la couche de programmation par objets n'est pas supportée.

L'implantation actuelle est basée sur le code originel du compilateur OCaml. Ce sont les premières phases du compilateur qui sont reprises (analyses lexicale et syntaxique, et inférence et vérification des types). MLcov se charge d'imprimer, à partir de l'arbre de syntaxe décoré par les types, le programme instrumenté. La figure VII.3 présente un schéma de la chaîne de compilation d'un programme utilisant MLcov.

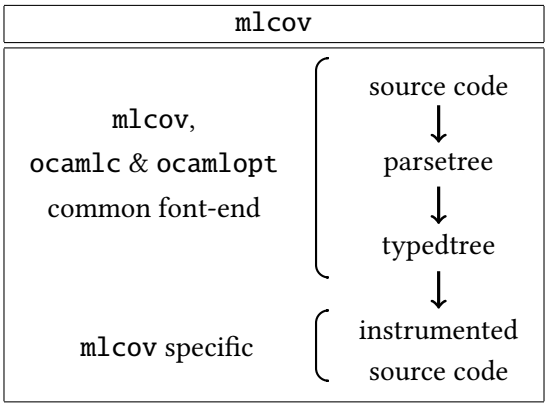


FIGURE VII.3 – Chaîne de l'instrumentation du code par MLcov

Histoire de MLcov Un prototype de MLcov a été développée en OCaml au laboratoire PPS par Benjamin Canou et Philippe Wang durant l'été 2006, pour la société Esterel Technologies. L'implantation ne reprenait pas le code du compilateur. Elle s'affranchissait de l'évolution du compilateur OCaml en n'utilisant que le noyau impératif et fonctionnel très stable du langage.

Esterel Technologies a ensuite développé une version de production de MLcov, en utilisant cette fois une partie du code compilateur OCaml. Le code des analyses lexicale et syntaxique ainsi que le code du typeur ont été repris tels quels. L'instrumentation par MLcov est donc effectuée après la phase de l'inférence et de la vérification des types par le code du compilateur OCaml. La reprise du code

attache MLcov aux évolutions du langage OCaml mais permet aussi de ne pas rejeter les programmes qui compilent avec OCaml non instrumentables par MLcov (*e.g.*, modules de première classe, objets).

VII.3.2 Sémantique utilisée par MLcov

Une définition simple du critère MC/DC. MLcov propose d'effectuer les mesures MC/DC sur les sites d'appels à retours booléens, ainsi que sur les expressions booléennes composées par les opérateurs booléens séquentiels **&&** et **||**. On peut considérer sa sémantique proche de celle que nous avons présentée à la section IV.2.3 avec quelques variations comme par exemple certaines règles de codage en moins.

On pourra constater dans la section suivante que, dans ses rapports de couverture, MLcov n'inclut pas dans la mesure de couverture structurelle simple les expressions qui sont mesurées selon le critère MC/DC.

VII.3.3 Un exemple de rapport de couverture avec MLcov

On reprend l'exemple présenté au chapitre II (section II.9.3, page 34) ainsi que son jeu de tests qui sera rappelé plus loin à la table VII.2 (page 133).

La figure VII.4 (page 130) contient le rapport avec une couverture de 100% par MLcov.

MLcov — Coverage Report (file scalene.ml)

generated by MLcov, version 1.2

STRUCTURAL COVERAGE STATISTICS

Function name	Covered points	Total points	Percentage
TOTAL	0	0	--

MC/DC STATISTICS

Decision number	Covered conditions	Total conditions	Percentage
#1 (all_different)	3	3	100.0 %
#2 (all_positive)	3	3	100.0 %
#3 (is_triangle)	4	4	100.0 %
#4 (is_scalene)	2	2	100.0 %
TOTAL	12	12	100.0 %

MLcov — Source Code Report (file scalene.ml)

generated by MLcov, version 1.2

```
let all_different a b c =
  a <> b && b <> c && c <> a

let all_positive a b c =
  a > 0 && b > 0 && c > 0

let is_triangle a b c =
  all_positive a b c
  && a + b > c
  && a + c > b
  && b + c > a

let is_scalene a b c =
  is_triangle a b c
  && all_different a b c
```

(Légende des couleurs : *cf.* page 131)

FIGURE VII.4 – Rapport de mesure de couverture pour le programme `scalene.ml` par MLcov

On remarque notamment que les expressions mesurées selon le critère MC/DC n'apparaissent pas dans la mesure de couverture simple.

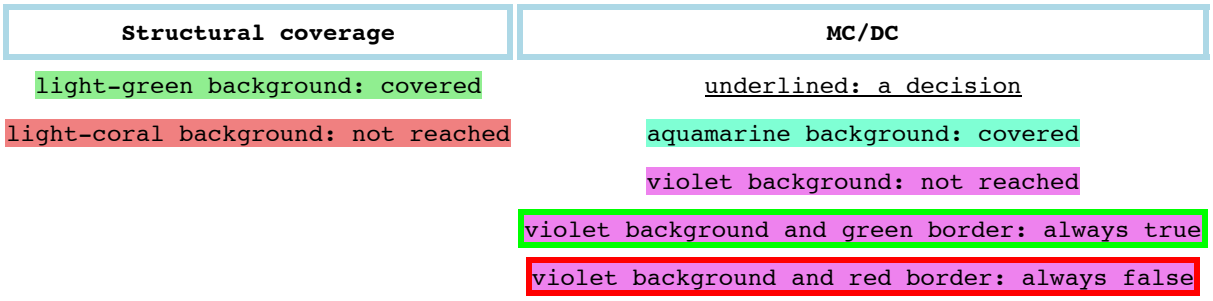


FIGURE VII.5 – MLcov : légende des couleurs

Pour le vérifier, nous donnons un rapport où les expressions booléennes composées (avec l'opérateur **&&**) ont été réécrites en utilisant des conditionnelles (construction **if**) à la place. Le rapport de couverture présenté en figure VII.6 (page 132) donne alors également une couverture de 100%, mais la mesure est répartie différemment.

Tests fonctionnels et effets sur la couverture structurelle Les tests fonctionnels engendrent la mesure de couverture structurelle. La table VII.2 (page 133) présente le jeu de tests précédemment présenté (section II.9.3, page 34) avec des annotations supplémentaires.

> **Rappel**
> On rappelle que la couverture structurelle doit idéalement être obtenue via des tests fonctionnels créés à partir des spécifications du programme (donc sans accès à son code source).

> « C. struct » est l'abréviation de « couverture structurelle simple. »

La couverture structurelle est absente sur la table VII.2 (page 133) du fait que le programme mesuré ne contient que des expressions qui sont mesurées selon le critère MC/DC.

On peut observer que la mesure MC/DC n'atteint pas 100% si on ne considère que des valeurs pour des triangles valides.

Pour faire apparaître la mesure de couverture structurelle, on fait de même avec la version sans opérateurs booléens « `scalene_struct.ml` ». La table VII.3 (page 134) montre les résultats des nouvelles mesures.

En comparant ces deux tables (VII.2 et VII.3), on observe d'une part que la mesure MC/DC est inchangée. L'explication est que le même nombre de conditions est couvert pour un même jeu de tests, et que le nombre total de conditions ne varie pas d'une version à l'autre du programme. Ceci s'applique à MLcov du fait que le taux de couverture des décisions n'est pas comptabilisé dans le taux de couverture global selon le critère MC/DC ; en effet, seules les conditions sont comptées.

Et d'autre part, on observe que la couverture structurelle apparaît bien cette fois-ci et que les taux de couverture simple évoluent à peu près de la même manière que les taux de MC/DC.

MLcov — Coverage Report (file scalene_struct.ml)

generated by MLcov, version 1.2

Structural coverage statistics

Function name	Covered points	Total points	Percentage
all_different	2	2	100.0 %
all_positive	2	2	100.0 %
is_triangle	3	3	100.0 %
is_scalene	1	1	100.0 %
TOTAL	**8**	**8**	**100.0 %**

MC/DC statistics

Decision number	Covered conditions	Total conditions	Percentage
#1 (all_different)	1	1	100.0 %
#2 (all_different)	1	1	100.0 %
#3 (all_different)	1	1	100.0 %
#4 (all_positive)	1	1	100.0 %
#5 (all_positive)	1	1	100.0 %
#6 (all_positive)	1	1	100.0 %
#7 (is_triangle)	1	1	100.0 %
#8 (is_triangle)	1	1	100.0 %
#9 (is_triangle)	1	1	100.0 %
#10 (is_triangle)	1	1	100.0 %
#11 (is_scalene)	1	1	100.0 %
#12 (is_scalene)	1	1	100.0 %
TOTAL	**12**	**12**	**100.0 %**

MLcov — Source Code Report (file scalene_struct.ml)

generated by MLcov, version 1.2

```
let all_different a b c =
  if a <> b then
     if b <> c then
        c <> a
      else
        false
  else
    false

let all_positive a b c =
  if a > 0 then
    if b > 0 then
      c > 0
     else
      false
   else
    false

let is_triangle a b c =
  if all_positive a b c then
    if a + b > c then
       if a + c > b then
         b + c > a
       else
         false
    else
      false
   else
     false

let is_scalene a b c =
  if is_triangle a b c then
    all_different a b c
   else
     false
```

(Légende des couleurs : *cf.* page 131)

Figure VII.6 – Rapport de mesure de couverture pour le programme `scalene_struct.ml` par MLcov

#	entrées			sortie	résultat	commentaire
Triangle valide ⇝ **C. struct. : –%** **MC/DC : 0,0 %**						
1	3	2	4	true	true	b < a < c
2	1238	2389	2839	true	true	a < b < c
3	23982389	19828390	12293381	true	true	c < b < a
4	9820	9932	8293	true	true	c < a < b
Triangle isocèle ⇝ **C. struct. : –%** **MC/DC : 33,3 %**						
5	10	10	4	false	false	a = b
6	42	8	42	false	false	a = c
7	55	54	54	false	false	b = c
Triangle équilatéral ⇝ **C. struct. : –%** **MC/DC : 33,3 %**						
8	89	89	89	false	false	a = b = c
Côtés trop courts (pas un triangle) ⇝ **C. struct. : –%** **MC/DC : 66,6 %**						
9	1	1	2	false	false	a + b = c
10	111	222	111	false	false	a + c = b
11	2	4	1	false	false	b > a + c
12	32	3	1	false	false	a > b + c
Valeurs négatives (pas un triangle) ⇝ **C. struct. : –%** **MC/DC : 100,0 %**						
13	-1	3	3	false	false	a < 0
14	4	-4190	4293	false	false	b < 0
15	34289	12833	-92238	false	false	c < 0
16	-3493	-3583	-3324	false	false	tous < 0
Zéro (pas un triangle) ⇝ **C. struct. : –%** **MC/DC : 100,0 %**						
17	0	23	21	false	false	a = 0
18	7	7	0	false	false	c = 0
19	0	0	0	false	false	tous = 0
20	29	0	0	false	false	b = c = 0
21	0	0	2	false	false	a = b = 0
22	0	23	0	false	false	a = c = 0

Table VII.2 – Effets des tests fonctionnels sur la mesure MC/DC pour `scalene.ml` avec MLcov

#	entrées			sortie	résultat	commentaire
Triangle valide ⇝ **C. struct. : 0,0 %** **MC/DC : 0,0 %**						
1	3	2	4	true	true	b < a < c
2	1238	2389	2839	true	true	a < b < c
3	23982389	19828390	12293381	true	true	c < b < a
4	9820	9932	8293	true	true	c < a < b
Triangle isocèle ⇝ **C. struct. : 25,0 %** **MC/DC : 33,3 %**						
5	10	10	4	false	false	a = b
6	42	8	42	false	false	a = c
7	55	54	54	false	false	b = c
Triangle équilatéral ⇝ **C. struct. : 25,0 %** **MC/DC : 33,3 %**						
8	89	89	89	false	false	a = b = c
Côtés trop courts (pas un triangle) ⇝ **C. struct. : 62,5 %** **MC/DC : 66,6 %**						
9	1	1	2	false	false	a + b = c
10	111	222	111	false	false	a + c = b
11	2	4	1	false	false	b > a + c
12	32	3	1	false	false	a > b + c
Valeurs négatives (pas un triangle) ⇝ **C. struct. : 100,0 %** **MC/DC : 100,0 %**						
13	-1	3	3	false	false	a < 0
14	4	-4190	4293	false	false	b < 0
15	34289	12833	-92238	false	false	c < 0
16	-3493	-3583	-3324	false	false	tous < 0
Zéro (pas un triangle) ⇝ **C. struct. : 100,0 %** **MC/DC : 100,0 %**						
17	0	23	21	false	false	a = 0
18	7	7	0	false	false	c = 0
19	0	0	0	false	false	tous = 0
20	29	0	0	false	false	b = c = 0
21	0	0	2	false	false	a = b = 0
22	0	23	0	false	false	a = c = 0

TABLE VII.3 – Effets des tests fonctionnels sur la mesure MC/DC pour `scalene_struct.ml` avec ML-cov

MLcov — Coverage Report (file scalene.ml)

generated by MLcov, version 1.2

STRUCTURAL COVERAGE STATISTICS

Function name	Covered points	Total points	Percentage
TOTAL	0	0	--

MC/DC STATISTICS

Decision number	Covered conditions	Total conditions	Percentage
#1 (all_different)	3	3	100.0 %
#2 (all_positive)	0	3	0.0 %
#3 (is_triangle)	0	4	0.0 %
#4 (is_scalene)	1	2	50.0 %
TOTAL	4	12	33.3 %

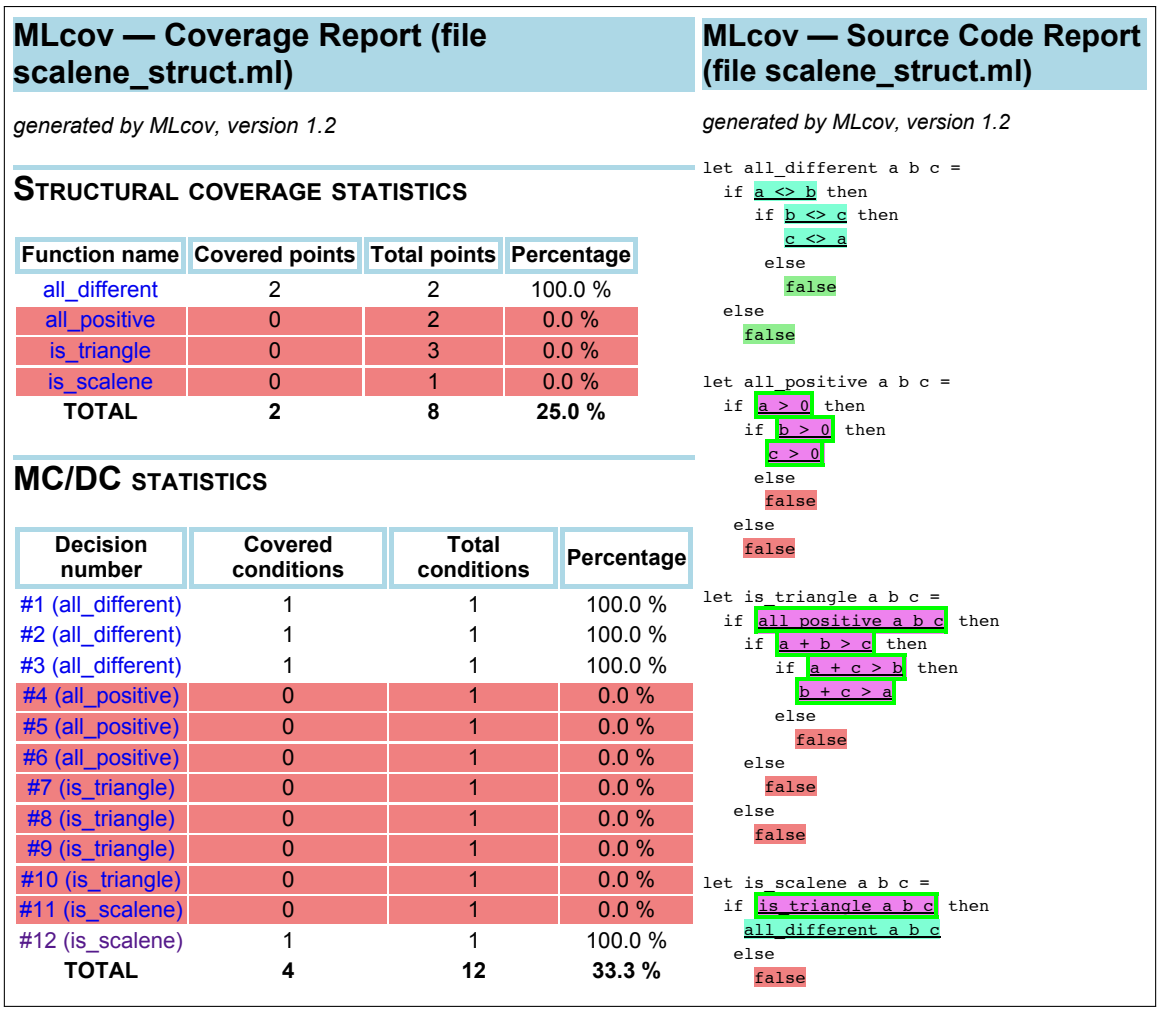
MLcov — Source Code Report (file scalene.ml)

generated by MLcov, version 1.2

```
let all_different a b c =
  a <> b && b <> c && c <> a

let all_positive a b c =
  a > 0 && b > 0 && c > 0

let is_triangle a b c =
  all_positive a b c
  && a + b > c
  && a + c > b
  && b + c > a

let is_scalene a b c =
  is_triangle a b c
  && all_different a b c
```

(Légende des couleurs : *cf.* page 131)

FIGURE VII.7 – MLcov : rapport avec couverture partielle, pour `scalene.ml`



MLcov — Coverage Report (file scalene_struct.ml)

generated by MLcov, version 1.2

Structural coverage statistics

Function name	Covered points	Total points	Percentage
all_different	2	2	100.0 %
all_positive	0	2	0.0 %
is_triangle	0	3	0.0 %
is_scalene	0	1	0.0 %
TOTAL	**2**	**8**	**25.0 %**

MC/DC statistics

Decision number	Covered conditions	Total conditions	Percentage
#1 (all_different)	1	1	100.0 %
#2 (all_different)	1	1	100.0 %
#3 (all_different)	1	1	100.0 %
#4 (all_positive)	0	1	0.0 %
#5 (all_positive)	0	1	0.0 %
#6 (all_positive)	0	1	0.0 %
#7 (is_triangle)	0	1	0.0 %
#8 (is_triangle)	0	1	0.0 %
#9 (is_triangle)	0	1	0.0 %
#10 (is_triangle)	0	1	0.0 %
#11 (is_scalene)	0	1	0.0 %
#12 (is_scalene)	1	1	100.0 %
TOTAL	**4**	**12**	**33.3 %**

MLcov — Source Code Report (file scalene_struct.ml)

generated by MLcov, version 1.2

```
let all_different a b c =
  if a <> b then
     if b <> c then
        c <> a
      else
        false
  else
    false

let all_positive a b c =
  if a > 0 then
    if b > 0 then
      c > 0
     else
      false
   else
    false

let is_triangle a b c =
  if all_positive a b c then
    if a + b > c then
       if a + c > b then
          b + c > a
        else
          false
    else
       false
   else
     false

let is_scalene a b c =
  if is_triangle a b c then
    all_different a b c
   else
     false
```

(Légende des couleurs : *cf.* page 131)

Figure VII.8 – MLcov : rapport avec couverture partielle, pour `scalene_struct.ml`

VII.4 Zamcov : couverture structurelle non intrusive

Dans cette section, nous proposons une description de Zamcov, un ensemble d'outils développés selon l'approche non intrusive.

Histoire de Zamcov Zamcov a été intialement développé dans le cadre du projet Couverture [84], selon les périodes par Adrien Jonquet, Alexis Darrasse, Mathias Bourgoin et Philippe Wang. Après quelques expérimentations avec le code de la machine virtuelle fournie par la distribution officielle, *i.e.*, `ocamlrun`, une machine virtuelle OCaml a été développée en OCaml (`zamcov-run`). Cela aura permis d'implanter l'ensemble des outils Zamcov en OCaml, donc de ne pas avoir à implanter d'outils en C (puisque la machine virtuelle standard, `ocamlrun`, est implantée en C).

Ensuite, les informations de mise au point, produites par le compilateur à destination de l'outil `ocamldebug`, ont été utilisées pour produire les traces permettant de générer des rapports de couverture structurelle simple.

Les informations de mise au point n'étant pas suffisantes pour la mesure MC/DC du fait de devoir surveiller des valeurs booléennes et en particulier la relation des conditions se trouvant au sein d'une décision, il a fallu engendrer des informations plus riches. Le compilateur `ocamlc` a alors été modifié pour produire ces informations, ce qui donna alors l'outil `zamcov-compile`.

Pour la génération des rapports à partir du code source et des traces d'exécution, c'est `zamcov-cover` qui s'en charge en produisant des rapports utilisant les technologies standards de description de pages web (HTML/CSS).

Dans cette section, la description de Zamcov correspond à la version 1.0 du système, librement disponible à l'URL `http://www.algo-prog.info/zamcov/`. Les travaux présentés dans cette thèse ne sont pas tous représentés dans cette version de Zamcov qui devrait évoluer sous peu.

VII.4.1 Génération des informations de couverture

Zamcov embarque une modification du compilateur `ocamlc`, générant les informations nécessaires à l'environnement d'exécution pour émettre les traces nécessaires à l'établissement de rapports de couverture structurelle des expressions et des conditions et décisions.

L'implantation consiste à remplacer le code servant à générer les informations de mise au point (pour `ocamldebug`) par un code servant à générer les informations qui nous intéressent.

La figure VII.9 représente la chaîne de compilation du compilateur `ocamlc` original, ainsi que la chaîne de compilation de `zamcov-compile` qui en est une modification induisant une production d'informations supplémentaires (facultatives).

La figure VII.10 schématise le mécanisme de production de rapport par `zamcov-cover`.

Ainsi, Zamcov est un ensemble de trois outils :

1. `zamcov-compile` est une modification du compilateur `ocamlc` fourni avec la distribution standard du langage OCaml. Ce dernier produit du code pour la machine virtuelle OCaml, également fournie dans la distribution OCaml. `zamcov-compile` ajoute simplement la génération d'informations de correspondance entre le code source et le code machine produit, afin de pouvoir générer les traces permettant d'établir les rapports de couverture incluant les mesures des conditions et des décisions.
2. `zamcov-run` est une nouvelle implantation de la machine virtuelle OCaml, et permet de tirer profit des informations de correspondance entre code source et code machine produites par `zamcov-compile` afin de produire les traces voulues.

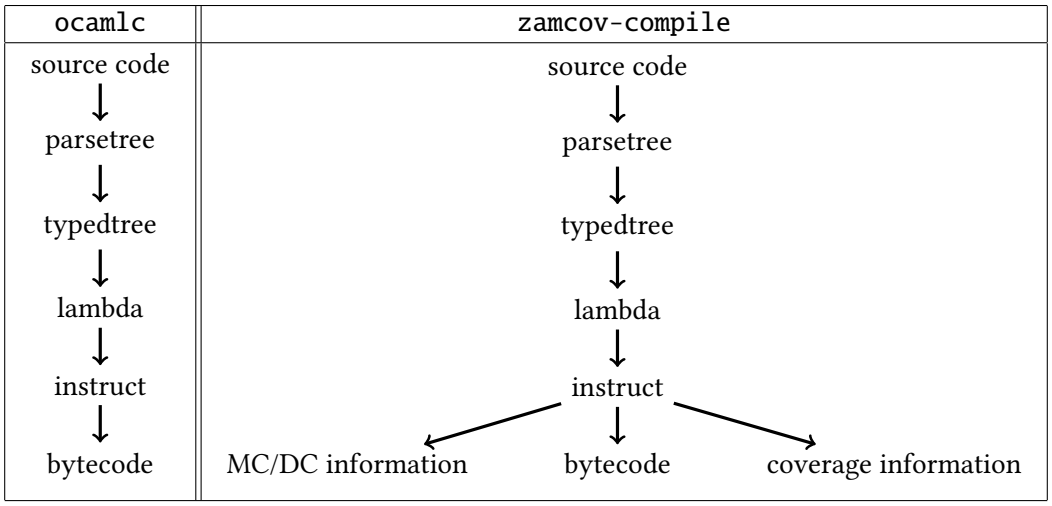


FIGURE VII.9 – Chaînes de compilation pour `ocamlc` et `zamcov-compile`

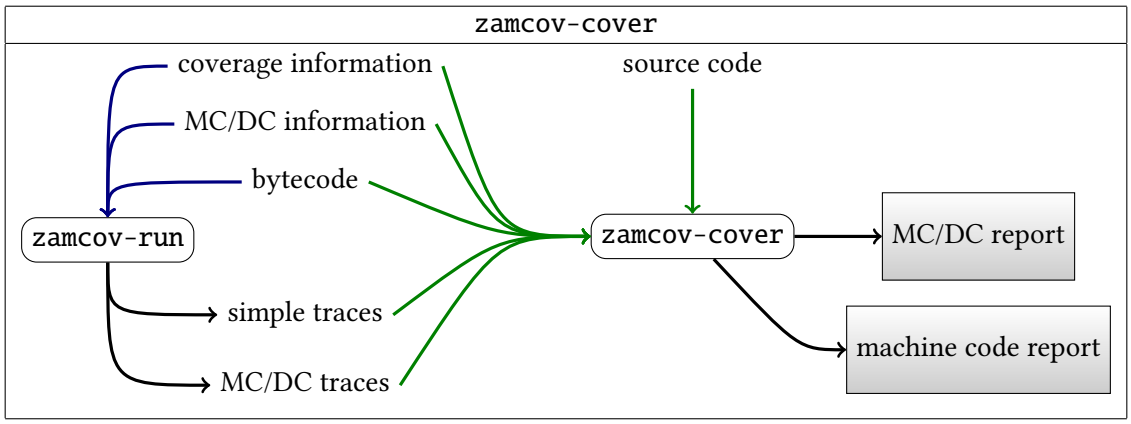


FIGURE VII.10 – Chaîne de production de rapport de Zamcov

3. `zamcov-cover` est le générateur de rapports de couverture, prenant le code source ainsi que les données engendrées par `zamcov-compile` et l'exécution du programme par `zamcov-run`.

VII.4.2 Illustration

Les rapports de couverture sont similaires à ceux de MLcov. Pour montrer les différences de présentation, on choisit de ne montrer que les couvertures partielles.

La figure VII.11 (page 139) montre le code de `scalene.ml` partiellement couvert, décoré par Zamcov. Elle est accompagnée par la figure VII.12 (page 139) qui légende le code couleur employé par Zamcov.

- La ligne 2 contient une grande expression couverte en tant qu'expression : l'arrière plan est vert. Elle est couverte en tant que décision : elle est soulignée d'un trait vert. Ses trois conditions sont également couvertes en tant que conditions : elles sont surlignées d'un trait vert.
- La ligne 5 contient une grande expression couverte en tant qu'expression. Elle est non couverte en tant que décision : elle est sous lignée d'un trait rouge. Ses trois conditions ne sont pas non plus couvertes : elles sont surlignées d'un trait rouge.

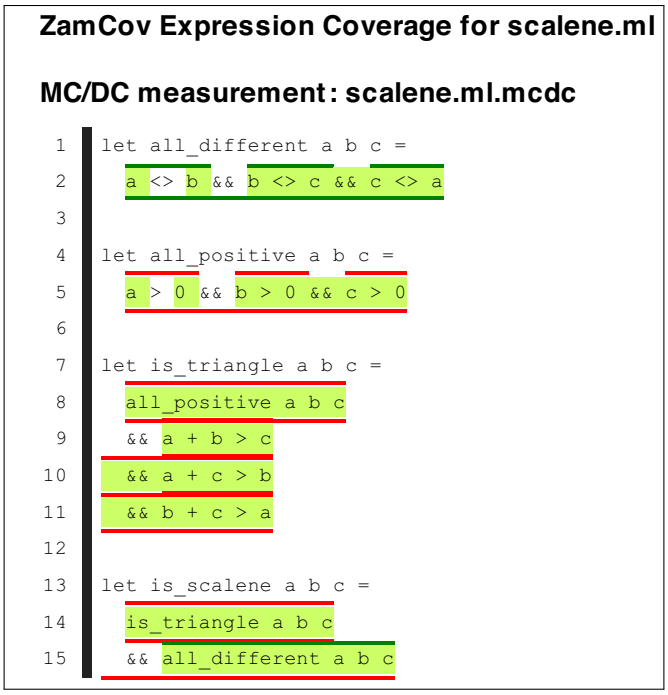


FIGURE VII.11 – Zamcov : rapport de couverture partielle : code de `scalene.ml` décoré

- Le cas de l'expression contenue de la ligne 8 à la ligne 11 est similaire au cas de l'expression de la ligne 5.
- L'expression des lignes 14 et 15 est couverte en tant qu'expression. Elle est non couverte en tant que décision. Elle contient deux conditions, dont une couverte et une non couverte.

La figure VII.13 (page 140) présente les mesures MC/DC sur le code.

La figure VII.14 (page 141) montre que certaines parties du code de `scalene_struct.ml` ne sont pas du tout atteinte du point de vue de la couverture des expressions.

Par rapport à MLcov. Zamcov mesure et présente la couverture des expressions indifféremment de la couverture des conditions et des décisions, contrairement à MLcov qui ne mesure pas la couverture

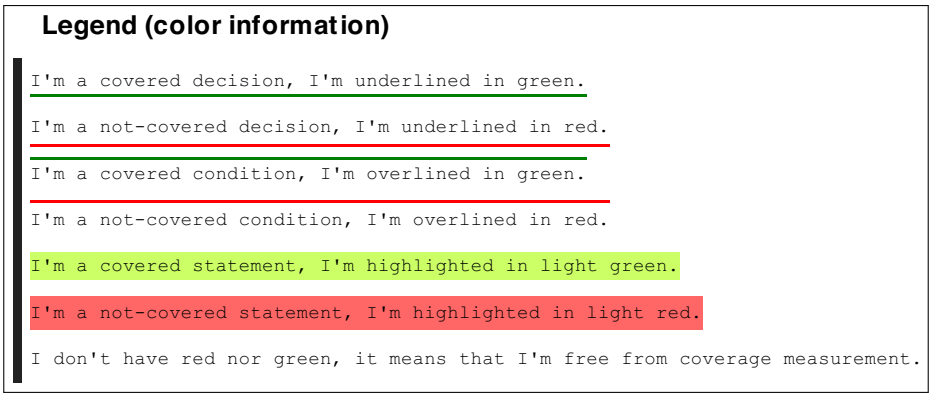


FIGURE VII.12 – Zamcov : légende du code décoré

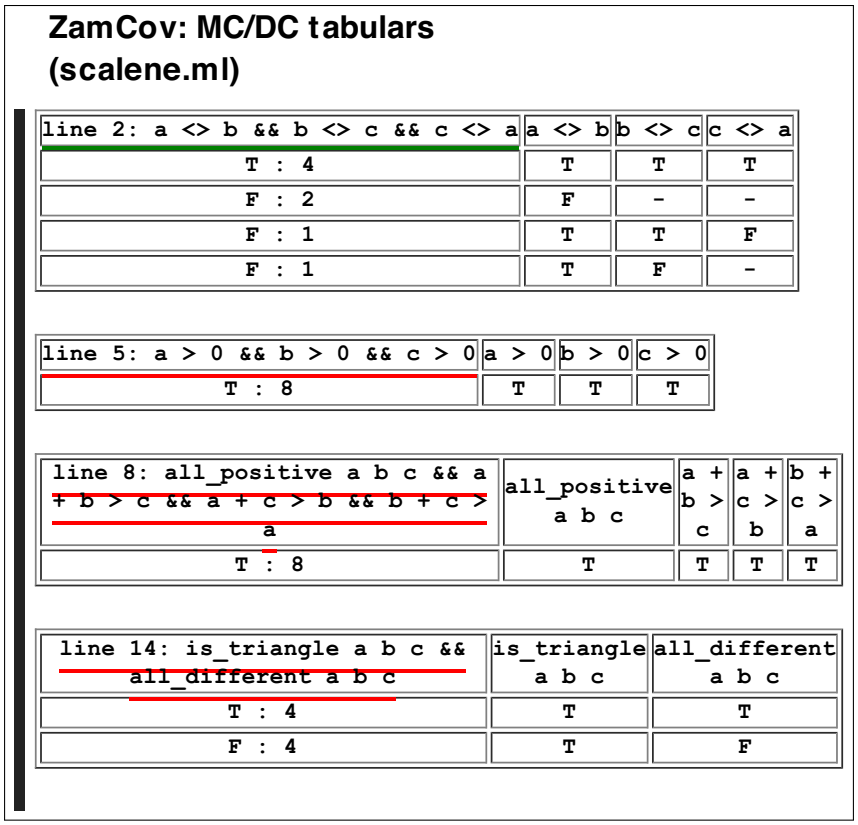
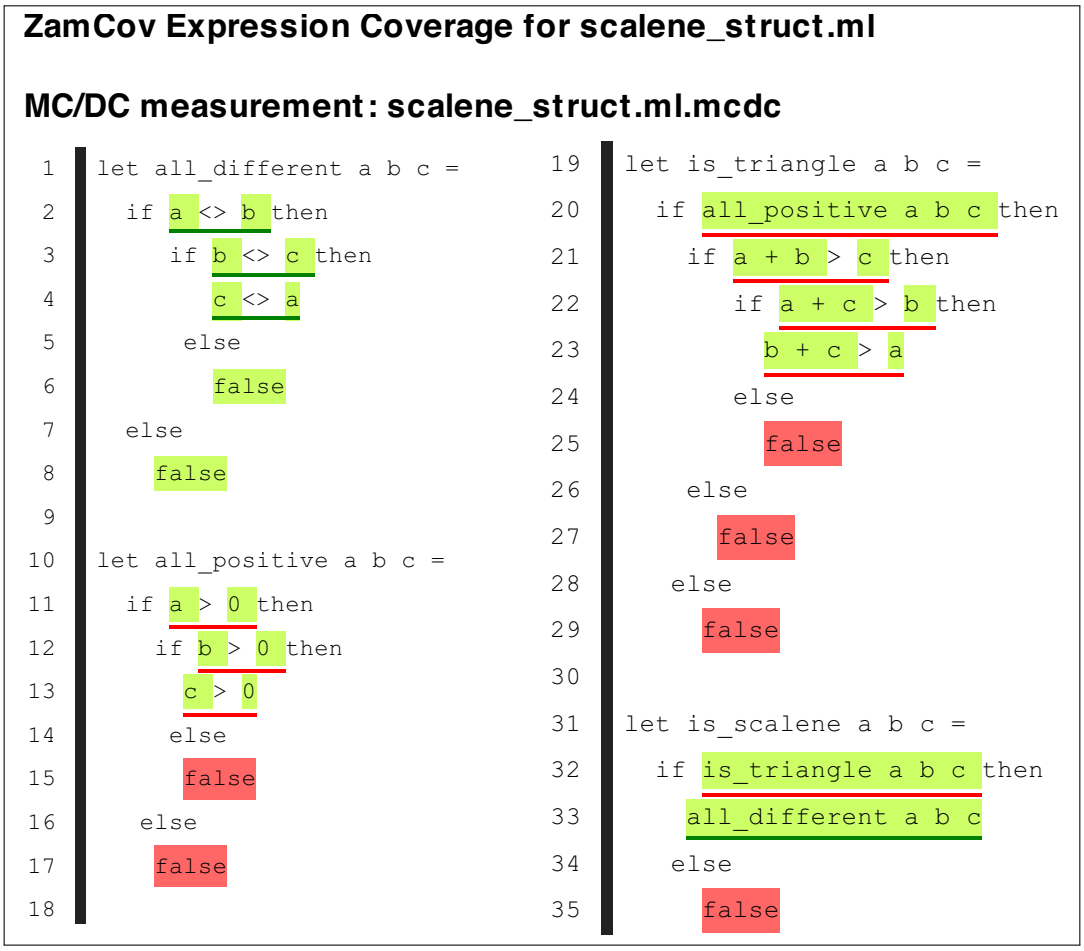


FIGURE VII.13 – Zamcov : mesures MC/DC

des expressions pour les expressions qui sont mesurées selon les critères conditionnels : *cf.* figure VII.11 (page 139) pour Zamcov, figure VII.7 (page 135) pour MLcov.

On peut aussi constater qu'avec Zamcov, les décisions n'ayant qu'une seule condition ne sont pas mesurées en tant que conditions : *cf.* figure VII.14 (page 141).



La légende des couleurs se trouve page 139.

FIGURE VII.14 – Zamcov : rapport de couverture partielle : code de `scalene_struct.ml` décoré

VII.5 Conclusion du chapitre

Ce chapitre a d'abord donné une comparaison formelle des deux approches, intrusive et non intrusive, basée sur les formalisations présentées dans les deux chapitres précédents. Nous avons développé ce qui concerne l'optimisation des appels terminaux, qui est cassée lorsqu'on adopte l'approche intrusive et qui est le cas compliqué lorsqu'on adopte l'approche non intrusive.

Puis nous avons donné une comparaison « pratique » des deux approches, c'est-à-dire du point de vue de l'utilisateur.

Enfin, nous avons présenté les deux outils MLcov et Zamcov, qui adoptent respectivement les deux approches intrusives et non intrusives, à travers une description générale illustrée par les rapports de couverture générés par les deux outils.

CHAPITRE VIII

Conclusion et Perspectives

« *Faire du code qui marche pas, c'est pas cher.*
Faire du code qui marche, c'est pas le même prix.
Il faut choisir ses ennuis. »
– Gérard Berry

Cette thèse établit un lien entre deux domaines, qui sont celui des activités de test dans le cadre de développements de logiciels critiques et celui des langages applicatifs.

Parmi les langages applicatifs, nous avons choisi un langage de la famille ML pour étudier les problématiques de son introduction dans le cadre des applications critiques.

D'une part, cette action fait écho à un besoin industriel. En effet, l'entreprise Esterel Technologies a développé en OCaml le générateur de code de SCADE 6, nommé KCG, produisant du code devant potentiellement être certifié pour le plus haut niveau de criticité (*e.g.*, niveau A pour l'avionique civile). Cela a impliqué l'adaptation des exigences du processus de certification et l'adaptation de l'usage du langage OCaml. Plus précisément, cela a nécessité l'application du processus de certification pour le plus haut niveau de criticité pour KCG, car un générateur de code doit suivre les mêmes obligations de certification que le code produit. Il a donc fallu pallier les manques d'outils (*e.g.*, outil de couverture de code) et de documentation (*e.g.*, traçabilité du compilateur) autour du langage OCaml pour permettre son utilisation dans ce contexte. C'est dans cette démarche que l'outil MLcov [1] a vu le jour.

D'autre part, cette action s'intègre à l'initiative du projet Couverture [10] [84] visant à permettre l'établissement de la couverture de programmes sans instrumentation du code source. Cela permet aux binaires produits pour les tests et les mesures de couvertures d'être les mêmes que les binaires finaux. Pour l'établissement des traces d'exécution pour les rapports de couverture, c'est l'environnement d'exécution qui est instrumenté pour générer les informations nécessaires. Cette démarche a été mise en pratique dans le cadre du travail de cette thèse par le développement de Zamcov, un ensemble d'outils permettant d'établir des rapports de couverture structurelle de code pour des programmes écrits en OCaml, sans instrumentation du code source.

Une formalisation des deux approches, intrusive (*i.e.*, par réécriture du code source) et non intrusive (*i.e.*, sans réécriture du code source mais par instrumentation de l'environnement d'exécution), a été présentée, et leur équivalence a été démontrée à optimisation du compilateur près.

1. On rappelle que MLcov est un outil de couverture de code pour OCaml, avec capacité de mesure MC/DC.

L'approche par réécriture du code source empêche l'optimisation classique des appels terminaux (*i.e.*, consommation bornée de niveaux de pile au lieu d'une consommation linéaire en fonction du nombre d'appels). Pour ce genre de programmes, l'approche non intrusive permet une plus grande flexibilité dans la mesure où l'instrumentation de l'environnement d'exécution a accès à la pile d'exécution et ne casse pas l'optimisation des appels terminaux. On peut ainsi affirmer que l'approche non intrusive peut être appliquée à davantage de programmes. On note cependant que les fonctions récursives profitant usuellement de l'optimisation des appels terminaux peuvent généralement être réécrites avec des boucles itératives.

L'approche sans modification du programme ne fonctionne pas avec certaines optimisations du compilateur. En effet, on a besoin que le compilateur soit une fonction injective du code source vers le code machine. C'est-à-dire qu'en exécutant une instruction du code machine, il faut savoir à quel endroit dans le code source elle correspond. Si le compilateur factorise deux fonctions qui sont identiques dans le code machine, il est généralement impossible de déterminer quelle fonction du code source est appelée. On note toutefois que, idéalement, pour la DO-178B ou la DO-178C, il faudrait que le compilateur soit une fonction bijective, car cela voudrait dire une traçabilité parfaite entre le code source et le code machine.

Le langage de la famille ML que nous avons formellement défini est mCaml, un sous-ensemble du langage OCaml. Il est à la fois suffisamment petit pour pouvoir raisonner formellement autour sans trop de peine, et suffisamment riche pour qu'il soit assez proche d'un langage qui peut être utilisé en pratique. Il n'est donc pas tout à fait minimaliste.

mCaml étant un langage d'expressions (il n'y a pas d'instructions), nous avons adapté la notion de couverture des instructions (*statement coverage*) pour les expressions en donnant une définition formelle de la couverture des expressions.

Les notions de décision et de condition qui sont définies dans la littérature ont dû être adaptées (formellement) pour notre langage. Du fait que mCaml est un langage d'expressions, les expressions booléennes peuvent apparaître quasiment dans n'importe quelle forme d'expression. Cela pose le problème de savoir lesquelles sont des conditions et lesquelles sont des décisions. Nous avons consacré une large place à la définition de ces notions, en donnant quatre sémantiques allant progressivement de la plus simple (directement appliquée de la littérature à notre langage mCaml) en posant de fortes restrictions syntaxiques (ou règles de codage), jusqu'à une définition innovatrice qui s'affranchit complètement des règles de codage en les remplaçant par des obligations de mesures, ce qui permet à notre langage de retrouver son expressivité. Cet ensemble permet alors de donner un sens formel **complet** à la définition de la couverture des conditions/décisions modifiée (MC/DC).

La sémantique 4 (définie en section IV.2.5, page 78) est celle que nous avons adoptée pour établir formellement les méthodes de mesure par les deux approches intrusive et non intrusive des valeurs des conditions et des décisions pour la génération de rapports de mesure MC/DC. Les mesures alternatives à MC/DC (*e.g.*, C/DC, RC/DC, MUMCUT) peuvent bien sûr profiter des mêmes méthodes.

Toutefois, cette sémantique 4 laisse des perspectives, puisqu'elle n'a pas traité le problème des applications polymorphes. En effet, dans le cadre du typage statique à la ML, une fonction générique peut être une fonction qui accepte un argument de n'importe quel type. Cet argument est donc potentiellement un booléen. Aucune des quatre sémantiques que nous avons définies n'exige la mesure de la valeur des booléens qui passent sur un site d'appel polymorphe.

Quatre catégories de fonctions polymorphes acceptant des booléens

Une première catégorie est celle où les fonctions n'effectuent pas de calculs dépendant des paramètres polymorphes. Celles-ci ne posent pas de problème même lorsque les paramètres sont booléens puisque les calculs ne dépendent pas des valeurs prises par ces booléens : on peut considérer qu'il ne servirait à rien d'inspecter les valeurs puisqu'elles ne sont pas lues et donc n'influent sur aucun branchement.

Une seconde catégorie est celle où les paramètres peuvent être des fonctions à qui déléguer des calculs qui sont potentiellement sur des valeurs booléennes.

En troisième catégorie, on peut prendre en exemple l'opérateur d'égalité (physique ou structurelle) qui prend deux éléments du même type et retourne un booléen. Si les deux éléments sont des booléens, ils ne sont pas reconnus en tant que tels par les sémantiques proposées y compris par la sémantique 4 si le site d'appel est polymorphe. On peut donc se retrouver à effectuer un calcul dépendant de deux valeurs booléennes sans faire de mesures sur celles-ci. Cette catégorie est celle des primitives polymorphes (et des appels externes) qui ne peuvent pas être définis en mCaml pur.

La quatrième catégorie est celle qui mélange les trois autres.

Que faire des fonctions polymorphes acceptant des booléens

Un choix possible est alors de considérer de manière générale qu'il ne faut pas utiliser des fonctions polymorphes sur des booléens afin d'éviter en particulier le troisième cas (soit via les règles de codage, soit via le typeur).

Alternativement, on peut considérer que les mesures effectuées sur le site d'appel à paramètres booléens suffisent puisqu'en fait, en amont un des sites d'appels mettra en évidence qu'il reçoit des booléens qui seront donc détectés et inspectés en tant que tels. Le problème est alors qu'on peut plus difficilement détecter les factorisations effectuées dans le but de faciliter la satisfaction du critère MC/DC.

On peut aussi enrichir le langage au niveau du système de types pour pouvoir exprimer des restrictions sur le polymorphisme : si on peut représenter un type polymorphe où tous les types sont acceptés sauf les booléens, cela peut permettre de garantir qu'aucune valeur booléenne n'a affecté une décision sans le faire paraître au niveau de la décision. Autrement dit, en plus ou à la place du schéma de type $\forall t.t$, on pourrait avoir $\forall t \neq \texttt{bool}.t$.

En perspectives, on pourra définir des sémantiques formelles pour un langage plus riche que mCaml, mais aussi et surtout pour d'autres familles de langages de haut-niveau. On pourra également s'adapter à d'autres obligations de mesures, comme par exemple sur le mécanisme d'appels de méthodes pour la programmation par objets.

MLcov et Zamcov

MLcov est un outil développé par Esterel Technologies, adoptant l'approche intrusive pour la couverture de code OCaml. Il adopte une sémantique proche de la sémantique 2.

Dans le cadre de cette thèse et du projet Couverture [2], l'approche innovatrice (non intrusive) a été expérimentée par l'implantation du prototype Zamcov 1.0 qui propose la génération de rapports de couverture du code source avec mesure MC/DC. (Zamcov propose aussi l'établissement de rapports de

2. Notons par ailleurs que dans le cadre du projet Couverture, AdaCore a développé l'outil XCOV qui adopte une approche non intrusive pour la couverture de code structurelle de programmes Ada avec mesure MC/DC.

couverture du code machine.) Il adopte actuellement une sémantique proche de la sémantique 2 pour la mesure MC/DC du code source. La sémantique 4 est en cours d'implantation pour Zamcov 2.0.

On rappelle que la sémantique 2 consiste à distinguer une décision via les opérateurs booléens (conjonction et disjonction) s'il s'agit d'une décision à plusieurs conditions, et via son type seulement s'il s'agit d'une décision à une seule condition. Notamment, les applications à paramètres et retour booléens ne sont pas vues en tant que décisions à plusieurs conditions mais en tant qu'une décision à une seule condition. Et une partie des constructions possibles du langage est rejetée par des règles de codage.

Perspectives

Performances des outils

Les performances des programmes instrumentés par l'outil MLcov sont assez peu impactées lorsqu'il s'agit de l'utiliser pour le générateur de code KCG mais sont pénalisées lorsque les programmes qui usent intensivement de la récursion [66].

Les performances des programmes exécutés dans l'environnement Zamcov sont assez fortement impactées, pour deux raisons principales. La première est que la machine virtuelle de Zamcov est écrite en OCaml, ce qui entraîne souvent des calculs plus nombreux par rapport à celle de référence qui est écrite en C. La seconde raison est que Zamcov n'a pas encore profité de travaux sur les performances, il reste encore [3] un prototype.

Différentes versions [4] de Zamcov ont pu être essayées par Thomas Moniot de la société Esterel Technologies sur le générateur de code KCG qui est écrit en OCaml. Les résultats préliminaires indiquent qu'il y a des progrès sur les performances de Zamcov et qu'il reste un potentiel important pour leur amélioration.

Optimisation des mesures

Certains critères de couverture peuvent avoir des propriétés intéressantes pour réduire les opérations de génération de traces, ce qui peut aider à l'amélioration des performances. Par exemple, le critère OBC [23] a une correspondance avec le critère MC/DC pour certaines expressions (*cf.* section IV.1.6.3, page 70), ce qui permet dans des cas spécifiques d'avoir l'équivalence entre le critère de couverture des conditions simple avec le critère MC/DC.

De manière plus générale, pour réduire la quantité de calculs à effectuer à l'exécution pour la génération des traces, on peut penser à utiliser l'analyse statique comme par exemple les analyses de contrôle de flot ou l'interprétation abstraite. Cela peut permettre de factoriser certains calculs et de les effectuer avant l'exécution.

Autres encodages des booléens

Dans l'optique de réduire les obligations de mesure, il pourrait être possible d'utiliser d'autres encodages des booléens pour qu'ils ne soient pas détectés en tant que tels par les outils de couverture.

En mCaml, on peut par exemple utiliser des entiers ou un type somme à la place, et redéfinir les opérateurs logiques sur ces données. Ce genre de pratiques est en fait d'emblée une assez mauvaise idée.

3. Septembre 2012.
4. La version 0.9 date de 2010, la version 1.0 date de mars 2012, et la version 2.0-alpha date de septembre 2012.

En effet, d'une manière générale, cacher les booléens dans ce but serait une pratique très mal vue par les auditeurs. D'autre part, si des valeurs booléennes sont codées autrement et qu'elles échappent ainsi aux capacités de l'outil de mesure, l'autorité de certification pourra demander les mesures sur ces booléens encodés autrement comme s'il s'agissait de booléens standards, ce qui annihile le potentiel intérêt d'une telle ruse.

Cette question est toutefois très proche de celle qui demande si les expressions de types énumérés ou types sommes doivent avoir pris toutes les valeurs possibles. Par exemple, si une fonction prend un arbre en argument, faut-il imposer à la fonction d'avoir été utilisée à la fois sur des arbres vides et des arbres non vides ?

Si d'autres mesures s'imposent à l'avenir, l'approche non intrusive pourra offrir davantage de latitude pour leur établissement, notamment parce qu'elle a accès à l'ensemble de l'environnement d'exécution.

Filtrage par motifs

Le filtrage par motifs est fondamentalement une construction à branches puisque sa compilation [1] [49] [57] construit un arbre de décisions avec plus ou moins de partage (donc éventuellement un graphe acyclique orienté) selon les optimisations.

Mais contrairement aux constructions à branches plus classiques telles que la conditionnelle (**`if`**) ou la boucle non bornée (**`while`**), ses branches ne sont pas empruntées selon l'évaluation d'une expression booléenne mais selon le filtrage d'une valeur de n'importe quel type. Le filtrage par motifs ne fait pas apparaître dans le code source des valeurs booléennes de décision. Toutefois sa compilation produit dans le code machine des décisions.

Voici deux positions possibles sur cette question.

L'une est de vérifier qu'au niveau du code machine, tous les branchements sont effectivement couverts lorsque le code source est couvert. Cela reviendrait à établir deux mesures : l'une au niveau du code source, et l'autre au niveau du code machine.

L'autre consiste à faire confiance au compilateur (après l'avoir méticuleusement examiné, par exemple en vérifiant la traçabilité entre le code source et le code objet) et à accepter que couvrir chacune des branches du **`match`** est équivalent à satisfaire le critère MC/DC si le filtrage était écrit sous forme de compositions de **`if`**. On peut même considérer qu'on peut accorder une meilleure confiance en la version utilisant **`match`** car plus compacte, plus lisible, de plus haut niveau d'abstraction et plus proche des spécifications en plus des raisons techniques qui sont les suivantes :

- il est possible de détecter à la compilation la plupart des branches inutiles qui peuvent être rencontrées en pratique, c'est-à-dire les cas redondants qui ne pourront jamais être atteints puisqu'un filtre identique ou plus général a été placé plus tôt ;
- les cas qui manquent en fonction du type de la valeur filtrée peuvent être détectées à la compilation, ce qui permet de garantir qu'il existe un filtre pour tous les cas possibles.

Ensuite, concernant l'indépendance de chacune des branches, si on prend en considération le critère MC/DC, on peut la considérer immédiate du fait que chaque branche correspond à une décision. Un autre point de vue pourrait être de considérer que l'indépendance est impossible à montrer du fait que jamais deux branches ne soient empruntées consécutivement.

On peut aussi s'appuyer sur le fait que la traçabilité exigée par la DO-178B ou la nouvelle DO-178C n'oblige pas à avoir à la fois la couverture du code source et la couverture du code machine. Dans ce cas, la couverture des branches du filtrage par motifs au niveau du code source suffit. On note tout de même que dans tous les cas, si on s'appuie sur la DO-178B ou la DO-178C, la traçabilité

du code machine aux exigences de haut-niveau doit être montrée, que ce soit en passant par le code source ou non. C'est-à-dire que la traçabilité du code machine au code source peut être montrée au niveau du compilateur, ce qui permet par composition de montrer la traçabilité du code machine aux spécifications si on a montré par ailleurs la traçabilité du code source aux spécifications.

Programmation concurrente ou parallèle

Nature des mesures

Les critères de couverture conditionnelle (*e.g.*, couverture des conditions, couverture des décisions, MC/DC) existent pour mesurer la manière dont la décision pour les branchements conditionnels est établie. Ces mesures ont pour but de montrer que les expressions en position de décision ont bien été exercées convenablement (*e.g.*, pas de condition neutre, pas de redondance ni d'erreur).

La concurrence ou le parallélisme implique généralement un déroulement non déterministe et difficile à reproduire.

Si on exécute deux programmes P1 et P2 en concurrence, de manière générale l'un peut s'exécuter avant l'autre ou l'inverse, ou bien les deux peuvent s'exécuter en même temps. Cela pose un problème surtout si les deux programmes partagent une ressource.

Moyen de mesures

Outre la nature des mesures, pour les programmes concurrents ou parallèles la problématique de la mise en œuvre des techniques de mesure n'a pas été étudiée dans cette thèse. De nombreuses problématiques se posent lorsque les programmes deviennent concurrents, par exemple :

- cohérence des données : une donnée partagée mais accédée séquentiellement soit en lecture soit en écriture peut devenir une donnée accédée de manière concurrente, où l'absence de protection (par exemple par verrous) peut rendre la donnée incohérente ;
- taille des traces : si on fait le choix que chaque fil d'exécution produise une trace, on risque d'avoir une taille des traces dépendant linéairement du nombre de fils d'exécution ;
- nature des mesures : des obligations supplémentaires peuvent intervenir pour des programmes concurrents, les différentes techniques d'instrumentation (intrusives ou non intrusives) peuvent avoir d'autres avantages et inconvénients lorsqu'on change de modèle d'exécution.

Comme de nombreuses extensions existent pour le langage OCaml (*e.g.*, squelettes pour le parallélisme : OCamlP3L [18], data-parallélisme : CamlFlight[29], modèle synchrone : ReactiveML [56]), il sera intéressant de voir comment faire les mesures pour les traits de langage ajoutés et comment se comportent les outils de mesure pour du code généré.

Vers des programmes de plus en plus complexes

Les progrès dans le domaine de la haute technologie tendent à complexifier de plus en plus les logiciels. Cela vient d'abord des exigences de haut-niveau qui sont de plus en plus riches.

En conséquence, les coûts tendent à augmenter puisque le travail à fournir est de plus en plus important. En effet, d'une part on produit des logiciels de plus en plus gros, et d'autre part le coût des tests évolue selon la même tendance. Parallèlement, on veut donc réduire ces coûts, par l'automatisation des tâches ou par l'amélioration des processus d'automatisation. Si la proportion des coûts des tests n'a que peu évolué face à la complexification des applications, c'est en partie grâce aux outils de développement et aux outils de tests qui deviennent de plus en plus performants.

Dans le cadre des développements d'applications critiques, la tendance est d'attendre une plus grande maturité des technologies avant de les utiliser, si on compare aux pratiques des domaines peu sensibles qui sont plus aptes à être précurseurs de nouvelles technologies. Néanmoins, pour construire des applications de plus en plus complexes, il est nécessaire d'intégrer des technologies elles-mêmes de plus en plus complexes dans les processus de développement.

C'est dans cette optique de progrès que nous avons proposé un langage applicatif pour développer des outils aidant à la réalisation d'applications critiques.

Dans la même veine, les langages pour la programmation par objets sont introduits par un document [75] accompagnant la nouvelle DO-178C [74], ce qui est l'une des nouveautés attendues par cette révision. Toutefois, les problématiques de mesure pour les langages à objets sont encore plus complexes à cause des vulnérabilités (principalement dues à l'héritage (liaisons tardives) et au sous-typage) soulignées par ce document, et pourront difficilement être résolues par une instrumentation par réécriture du code source. Il faudra vraisemblablement établir des instrumentations de l'environnement d'exécution, par exemple pour ce qui concerne les mesures liées au système d'appels de méthodes (ou envois de messages, ou encore *dispatch*). Cela ouvre une nouvelle problématique qui s'inscrit dans la lignée du travail présenté dans cette thèse.

Bibliographie

[1] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer Berlin / Heidelberg, 1985. http://dx.doi.org/10.1007/3-540-15975-4_48.

[2] John Barnes. *High Integrity Software : The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[3] Daniel Bartholomew. QEMU : a multihost, multitarget emulator. *Linux J.*, 2006(145) :3–, May 2006.

[4] Michael Beine and Michael Jungmann. Code generation for safety-critical systems. In *Embedded Real Time Software (ERTS 2004)*, Toulouse, France, jan 2004. http://www.sia.fr/evenement_detail_embedded_real_time_software_actes_183.htm.

[5] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[7] Dines Bjørner. *Software Engineering 3 : Domains, Requirements, and Software Design*. Texts in theoretical computer science. Springer, 2006.

[8] Dines Bjørner. *Software Engineering : Abstraction and modelling*. Texts in theoretical computer science. Springer, 2006.

[9] Dines Bjørner. *Software Engineering Vol. 2 : Specification Of Systems And Languages*. Texts in theoretical computer science. Springer, 2006.

[10] Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, and Thomas Quinot. Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework. In *Embedded Real Time Software and Systems 2010*, Toulouse, France, May 2010.

[11] Benjamin Canou, Vincent Balat, and Emmanuel Chailloux. O'Browser : Objective Caml on browsers. In *ML '08 : Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 69–78, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1411304.1411315.

[12] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From Simulink to SCADE/lustre to TTA : a layered approach for distributed embedded applications. *SIGPLAN Not.*, 38(7) :153–162, June 2003. http://www-verimag.imag.fr/~tripakis/papers/lctes03.ps.

[13] Certification Authorities Software Team. What is a “Decision” in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC) ? Position Papier CAST-10, FAA, jun 2002. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf.

[14] John Joseph Chilenski. An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion. Technical report, FAA, apr 2001.

[15] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language : mini-ml. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP ’86, pages 13–27, New York, NY, USA, 1986. ACM. http://doi.acm.org/10.1145/319838.319847.

[16] Xavier Clerc. Cadmium, 2007. http://cadmium.x9c.fr.

[17] Xavier Clerc. Caml Virtual Machine – Instruction set Document version : 1.4, feb 2010. http://cadmium.x9c.fr/distrib/caml-instructions.pdf.

[18] Roberto Di Cosmo, Zheng Li, Susanna Pelagatti, and Pierre Weis. Skeletal Parallel Programming with OCamlP3l 2.0. *Parallel Processing Letters*, 18(1) :149–164, 2008.

[19] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 50–64. Springer Verlag, 1985.

[20] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2) :173 – 202, 1987.

[21] Guy Cousineau and Gérard Huet. The CAML primer. Technical Report RT-0122, INRIA, September 1990. http://hal.inria.fr/inria-00070045/PDF/RT-0122.pdf.

[22] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP’05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2005. © Springer.

[23] Couverture project Team. Technical Report on OBC/MCDC properties. Technical report, OpenDo, oct 2010. https://forge.open-do.org/scm/viewvc.php/trunk/couverture/techreports/obc-mcdc.pdf?view=log&root=couverture&sortby=author&pathrev=2734.

[24] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report : OCaml for an industrial-strength static analysis framework. In *ICFP ’09 : Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 281–286, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1596550.1596591.

[25] René David, Karim Nour, Pierre-Louis Curien, and Christophe Raffali. *Introduction à la logique : théorie de la démonstration : cours et exercices corrigés. Licence 3e année ; Master ; CAPES ; Agrégation.* Dunod, Paris, 2nd edition, 2004.

[26] Mark Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2) :84, 1997.

[27] Guy Durrieu, Hélène Waeselynck, and Virginie Wiels. LETO - A Lustre-Based Test Oracle for Airbus Critical Systems. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin / Heidelberg, 2009. http://dx.doi.org/10.1007/978-3-642-03240-0_5.

[28] Henry Cejtin *et al.* MLton documentation, April 2012. http://mlton.org/Documentation.

[29] Christian Foisy and Emmanuel Chailloux. Caml Flight : a Portable SPMD Extension of ML for Distributed Memory Multiprocessors, 1995.

[30] François-Xavier Fornari. Certification of a Scade 6 compiler, December 2008. http://synchron2008.lri.fr/slides/fornari.pdf.

[31] Jacques Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming, Nara, LNCS 2998*, pages 196–213. Springer-Verlag, 2004.

[32] James Gosling and Henry McGilton. The Java Language Environment. White Paper, May 1996. http://java.sun.com/docs/white/langenv/index.html.

[33] Wolfgang A. Halang and Janusz Zalewski. Programming languages for use in safety-related applications. *Annual Reviews in Control*, 27 :39–45, 2003. http://itech.fgcu.edu/faculty/zalewski/CEN3213/pdf/zalLang.pdf.

[34] N. Halbwachs. A synchronous language at work : the story of Lustre. In *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, pages 3–11, Washington, DC, USA, 2005. IEEE Computer Society. http://hal.archives-ouvertes.fr/docs/00/19/08/83/PDF/halbwachs.memocodes.pdf.

[35] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[36] S.P. Harbison and G.L. Steele. *C, a reference manual.* Prentice-Hall, 5 edition, 2002.

[37] Therèse Hardin, François Pessaux, Pierre Weis, and Damien Doligez. Focalize reference manual 0.6.0, dec 2009. http://focalize.inria.fr/documentation/focalize-0.6.0_refman.pdf.

[38] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, NASA/TM-2001-210876, May 2001. http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20010057789_2001090482.pdf.

[39] Grégoire Henry, Michel Mauny, Emmanuel Chailloux, and Pascal Manoury. Typing unmarshalling without marshalling types. In Peter Thiemann and Robby Bruce Findler, editors, *ICFP*, pages 287–298. ACM, 2012.

[40] Grégoire Henry. *Typer la sérialisation sans typer la désérialisation.* PhD thesis, Université Paris Diderot, jun 2011.

[41] Intel. FDIV Replacement Program. Statistical Analysis of Floating Point Flaw : Intel White Paper. Section 3 : Description of the Flaw., November 2009. http://www.intel.com/support/processors/pentium/sb/CS-013007.htm.

[42] Mike Jones. What really happened on Mars ?, 1997. http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html.

[43] Dusko Koncaliev. Pentium FDIV bug, 1998. http://www.cs.earlham.edu/~dusko/cs63/fdiv.html.

[44] Andrew J. Kornecki. Software Development Tools for Safety-Critical, Real-Time Systems Handbook. Final Report DOT/FAA/AR-06/35, National Technical Information Service (NTIS), Sprinfield, Virginia 22161, jun 2007. http://www.tc.faa.gov/its/worldpac/techrpt/ar0635.pdf.

[45] Andrew J. Kornecki, Nick Brixius, Janusz Zalewski Herman Lau, Jean-Philippe Linardon, Jonathan Labbe, Darryl Hearn, Kimberley Hall, Lazar Crawford, Celine Sanouillet, and Mathieu Milesi. Assessment of Software Development Tools for Safety- Critical, Real-Time Systems. Final Report DOT/FAA/AR-06/36, National Technical Information Service (NTIS), Sprinfield, Virginia 22161, jul 2007. http://www.tc.faa.gov/its/worldpac/techrpt/ar0636.pdf.

[46] Andrew J. Kornecki and Janusz Zalewski. Software certification for safety-critical systems : A status report. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 665–672. IEEE, October 2008.

[47] Andrew J. Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems : state of the art. *Innovations in Systems and Software Engineering*, 5(2) :149–161, jun 2009. http://www.springerlink.com/content/p3111754577xh842/.

[48] Gérard Ladier. Le DO 178-B / ED-12B : Logique, historique, contenu, application, mar 2003. RIS Atelier « Justification de sûreté de fonctionnement » http://www.ris.prd.fr/Ateliers/DEPEND-CASE/03-Ladier.pdf.

[49] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01 : Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM. http://doi.acm.org/10.1145/507635.507641.

[50] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical report 117, INRIA, 1990. http://gallium.inria.fr/~xleroy/publi/ZINC.pdf.

[51] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[52] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system (release 3.11) : Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, November 2008. http://caml.inria.fr/pub/docs/manual-ocaml/.

[53] Nancy Leveson. Medical Devices : The Therac-25, 1993. http://sunnyday.mit.edu/papers/therac.pdf.

[54] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Prentice Hall PTR, second edition, April 1999. Une version plus récente pour *Java SE 7 Edition* est disponible en PDF : http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf.

[55] Magnetrol International. Understanding safety integrity level, jan 2011. http://literature.magnetrol.com/1/41-299.pdf.

[56] Louis Mandel and Marc Pouzet. ReactiveML : un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques*, 27/9-10 :1097–1128, 2008.

[57] Luc Maranget. Compiling pattern matching to good decision trees. In *ML '08 : Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, New York, NY, USA, 2008. ACM.

[58] Robin Milner. Logic for computable functions : description of a machine implementation. Technical report, Stanford University, Stanford, CA, USA, 1972.

[59] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[60] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[61] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, rev sub edition, May 1997.

[62] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[63] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., third edition, 2011.

[64] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., second edition, 2004.

[65] WG14/N1256 – ISO/IEC 9899 : TC3. Committee Draft, September 2007. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[66] Bruno Pagano, Olivier Andrieu, Benjamin Canou, Emmanuel Chailloux, Jean-Louis Colaço, Thomas Moniot, and Philippe Wang. Certified development tools implementation in Objective Caml. In *Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag Berlin Heidelberg, 2008. http://dx.doi.org/10.1007/978-3-540-77442-6_2.

[67] Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury, and Jean-Louis Colaço. Experience Report : Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework. In *ICFP '09 : Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 215–220, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1596550.1596582.

[68] Bruno Pagano, Benjamin Canou, Emmanuel Chailloux, Jean-Louis Colaço, and Philippe Wang. Couverture de code Caml pour la réalisation d'outils de développement certifiés. In *Actes des Journées Francophones des Langages Applicatifs*, pages 61 – 75, Jan 2007. http://www.loria.fr/~moreau/jfla2007/PDF/05_pagano.pdf.

[69] Robert Pickering. *Foundations of F#*. Apress, Berkely, CA, USA, 2007.

[70] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, feb 2002. http://www.cis.upenn.edu/~bcpierce/tapl/.

[71] François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference. MIT Press, 2005.

[72] RTCA Special Committee 152. DO-178A – Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics*, 1985.

[73] RTCA Special Committee 167 and EUROCAE Working Group 12. DO-178B/ED-12B – Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics*, December 1992.

[74] RTCA Special Committee 205 and EUROCAE Working Group 71. DO-178C/ED-12C – Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics*, December 2011.

[75] RTCA Special Committee 205 and EUROCAE Working Group 71. Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A. *Radio Technical Commission for Aeronautics*, December 2011.

[76] Didier Rémy and Jérôme Vouillon. Objective ML : a Simple Object-Oriented Extension of ML. In *Symposium on Principles of Programming Languages*, pages 40 – 53, 1997. http://citeseer.ist.psu.edu/remy97objective.html.

[77] Didier Rémy and Jérôme Vouillon. Objective ML : An Effective Object-Oriented Extension to ML. *Theory and Practice of Object Systems*, 4(1) :27 – 50, 1998. http://citeseer.ist.psu.edu/remy98objective.html.

[78] Standard ML of New Jersey User's Guide, jan 2012. http://www.smlnj.org/doc/.

[79] Jean Souyris and Denis Favre-Félix. Proof of properties in avionics. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 527–535. Springer Boston, 2004.

[80] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal Verification of Avionics Software Products. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009 : Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin / Heidelberg, 2009. http://dx.doi.org/10.1007/978-3-642-05089-3_34.

[81] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 8.3pl2, jul 2011. http://coq.inria.fr/distrib/current/files/Reference-Manual.pdf.

[82] Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. Les microcontrôleurs PIC programmés en Objective Caml. In *Actes des vingt-deuxièmes Journées Francophones des Langages Applicatifs*, Studia Informatica Universalis, pages 177–207, La Bresse, France, January 2011. Hermann.

[83] Sergiy A. Vilkomir and Jonathan P. Bowen. From MC/DC to RC/DC : formalization and analysis of control-flow testing criteria. *Formal Aspects of Computing*, 18 :42–62, 2006. 10.1007/s00165-005-0084-7.

[84] Philippe Wang, Adrien Jonquet, and Emmanuel Chailloux. Non-Intrusive Structural Coverage for Objective Caml. *Electronic Notes in Theoretical Computer Science*, 264(4) :59 – 73, 2011. Proceedings of the Fifth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2010) http://www.sciencedirect.com/science/article/B75H1-526KXYF-1/2/6b9a2a612f450c7f53bb69ba76f0c686.

[85] Pierre Weis and Xavier Leroy. *Le langage Caml.* InterEditions, 1993. Nouvelle version (libre) téléchargeable : http://caml.inria.fr/pub/distrib/books/llc.pdf.

[86] Lauren Ruth Wiener. *Digital Woes : Why We Should Not Depend on Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[87] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4) :343–355, December 1995. http://dx.doi.org/10.1007/BF01018828.

[88] Spyros Xanthakis, Pascal Régnier, and Constantin Karapoulios. *Le test des logiciels.* Études et logiciels informatiques. Hermès, 2000.

[89] Yuen Tak Yu and Man Fai Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *J. Syst. Softw.*, 79 :577–590, May 2006. http://dx.doi.org/10.1016/j.jss.2005.05.030.

ANNEXE A

Sémantique formelle du langage mCaml (annexe du chapitre 3)

A.1 La grammaire en BNF

La grammaire du langage mCaml est définie à l'aide d'une notation à la BNF réduite au minimum. Les noms des termes de la grammaire sont en italique. Les mots-clefs du langage mCaml sont notés en gras non-italique. Enfin les constructions du langage BNF sont en fonte normale (il ne s'agit que des lexèmes `::=`, `|` et `::=|` ainsi que les expressions rationnelles).

Un programme mCaml est un ensemble de phrases séparées par un espace (composé d'une suite de caractères espace, tabulation ou retour à la ligne).

programme `::=` *phrase*
`|` *phrase* *programme*

Une phrase est une liaison globale d'une expression à une valeur, une définition de type ou une définition d'exception.

phrase `::=` *liaison-globale*
`|` *définition-type*
`|` *définition-exception*

Une liaison globale d'une variable à une expression permet de définir des expressions et de les associer à des variables, afin que les expressions soient accessibles depuis la suite du programme. Les fonctions récursives sont déclarées avec **`let rec`**.

liaison-globale `::=` **`let`** *nom-variable* **`=`** *expression*
`|` **`let rec`** *nom-variable* **`=`** **`fun`** *nom-variable* **`->`** *expression*
nom-variable `::=` `[a-z][a-zA-Z0-9_]*`

La définition d'un type permet d'ajouter aux types existants des types de données algébriques définis par le programmeur : les types enregistrements (produits) et les types variants (sommes). Il pourra alors construire des valeurs de ces nouveaux types.

Nous distinguons les noms de types des variables de type. Les premiers désignent un type bien précis tandis que les seconds peuvent incarner différents types en fonction du contexte. Pour les distinguer lexicalement, les variables de types ont des noms qui commencent par une apostrophe.

Les types peuvent être paramétrés par zéro, une ou plusieurs variables de type. Les paramètres servent alors à lier les variables libres de types apparaissant dans la définition du type. Lorsqu'il y a au moins deux paramètres de type, on les met obligatoirement entre parenthèses et on marque leurs séparations avec des virgules.

Les types primitifs sont les nombres entiers (type **int**), les booléens (type **bool**) et le type singleton (type **unit**).

définition-type	::=	**type** *nom-type* **=** *type-enregistrement*
	|	**type** *nom-type* **=** *type-variant*
nom-type	::=	*type-primitif*
	|	`[a-z][a-zA-Z0-9]*`
	|	*variable-de-type* `[a-z][a-zA-Z0-9]*`
	|	**(** *paramètres-de-types* **)** `[a-z][a-zA-Z0-9]*`
type-primitif	::=	**int**
	|	**bool**
	|	**unit**
	|	**exn**
paramètres-de-types	::=	*variable-de-type*
	|	*variable-de-type* **,** *paramètres-de-types*
variable-de-type	::=	`'[a-z][a-zA-Z0-9]*`

Un type enregistrement est défini par un ensemble de champs qui associent chacun un label (ou étiquette) à un type. Ils servent à construire des produits regroupant différentes valeurs accessibles ensuite grâce aux noms de leur champ. Un champ peut être déclaré mutable, afin d'autoriser l'affectation du champ avec une nouvelle valeur. En effet, implicitement, les valeurs du langage mCaml ne sont pas mutables, il faut donc déclarer explicitement la mutabilité. Seuls les champs des types produits peuvent être déclarés mutables.

type-enregistrement	::=	**{** *type-champs-enregistrement* **}**
type-champs-enregistrement	::=	*champ-immutable*
	|	*champ-mutable*
	|	*champ-immutable* **;** *type-champs-enregistrement*
	|	*champ-mutable* **;** *type-champs-enregistrement*
champ-immutable	::=	*nom-champ* **:** *expression-type*
champ-mutable	::=	**mutable** *nom-champ* **:** *expression-type*
nom-champ	::=	`[a-z][a-zA-Z0-9]*`

Un type variant est défini par un ensemble de constructeurs avec ou sans argument. Un constructeur avec un argument est associé au type de l'argument, pour connaître l'ensemble des valeurs qu'il est autorisé à « transporter » avec lui.

type-variant	::=	*nom-constructeur*
	|	*nom-constructeur* **|** *type-variant*
	|	*nom-constructeur* **of** *expression-type*
	|	*nom-constructeur* **of** *expression-type* **|** *type-variant*
nom-constructeur	::=	`[A-Z][a-zA-Z0-9]*`

Les expressions de types du langage sont les noms de types (types primitifs et types définis par l'utilisateur), les variables de types et les types des fermetures (les flèches). Pour lever de potentielles ambiguïtés d'associativité, les expressions de types pourront être parenthésées.

expression-type ::= *nom-type*
| *variable-de-type*
| *type-à-paramètre*
| *expression-type* `->` *expression-type*
| `(` *expression-type* `)`

Nous avons besoin de *type-à-paramètre* pour les types paramétrés par un type monomorphe, comme par exemple `int list`.

type-à-paramètre ::= *expression-type nom-type*

Une définition d'exception enrichit le type extensible spécial **`exn`**, et se déclare à l'aide du mot-clef **`exception`**. Une exception est soit un nom de constructeur, soit un nom de constructeur et une expression de type. (Le typeur vérifiera que l'expression de type est monomorphe.)

définition-exception ::= **`exception`** *nom-constructeur*
| **`exception`** *nom-constructeur* **`of`** *expression-type*

Une expression est soit un nom de variable, soit une constante, soit une application, soit une fonction (ou abstraction), soit une séquence, soit une conditionnelle, soit une boucle itérative conditionnelle, soit une liaison locale, soit une liaison locale récursive, soit une conjonction ou une disjonction booléenne (opérateurs infixes séquentiels), soit un constructeur avec un argument, soit un constructeur constant (sans argument), soit une séquence, soit la création d'un enregistrement, soit l'affectation d'un champ d'enregistrement, soit un contrôle d'exceptions, soit un filtrage par motifs, soit une levée d'exception.

Pour ne pas avoir une grammaire encore plus volumineuse, on s'autorise à ne pas expliciter le parenthésage implicite. Ainsi, en cas d'ambiguïté, les parenthèses trancheront. Par exemple, si `a`, `b` et `c` sont des variables, alors `a (b c)` ne pourra pas s'écrire avec moins de parenthèses, bien que la grammaire ne le note pas formellement.
Remarquons que `a b c` équivaut à `(a b) c`.

expression	::=	*nom-variable*
	|	*constante*
	|	*expression expression*
	|	**fun** *nom-variable* **->** *expression*
	|	**if** *expression* **then** *expression* **else** *expression*
	|	**let** *nom-variable* **=** *expression* **in** *expression*
	|	**let rec** *nom-variable* **=** **fun** *nom-variable* **->** *expression* **in** *expression*
	|	*nom-constructeur* **(** *expression* **)**
	|	*expression* **;** *expression*
	|	**{** *champs-enregistrement* **}**
	|	*expression***.** *nom-champ*
	|	*expression***.** *nom-champ* **<-** *expression*
	|	**for** *nom-variable* **=** *expression* **to** *expression* **do** *expression* **done**
	|	**while** *expression* **do** *expression* **done**
	|	**match** *expression* **with** *branches-match*
	|	**try** *expression* **with** *nom-variable* **->** *expression*
	|	*expression* **&&** *expression*
	|	*expression* **||** *expression*
	|	**not** *expression*
	|	**raise** *expression*
	|	**(** *expression* **)**
champs-enregistrement	::=	*nom-champ* **=** *valeur*
	|	*nom-champ* **=** *valeur* **;** *champs-enregistrement*
constante	::=	*nombre-entier*
	|	**false**
	|	**true**
	|	**()**
	|	*nom-constructeur*
nombre-entier	::=	`[0-9]+`
branches-match	::=	*motif* **->** *expression*
	|	*motif* **->** *expression* **|** *branches-match*
motif	::=	*nom-variable*
	|	*constante*
	|	*nom-constructeur* **(***motif***)**
	|	**{** *motif-champs* **}**
	|	**(** *motif* **)**
motif-champs	::=	*nom-champ* **=** *motif*
	|	*nom-champ* **=** *motif* **;** *motif-champs*

A.2 Système de types du langage mCaml

Cette section présente formellement le système de types du langage mCaml. La présentation est assez standard, et se base sur celui défini par Hindley et Milner [59] pour le langage ML, donc avec un polymorphisme à la Damas-Milner (*i.e.*, la « généralisation » se fait au niveau du **`let`**, ce qui fait qu'elle est connue sous le nom de *let-polymorphism*) mais contraint par la *value restriction* de Wright[87] (*i.e.*, seules les expressions qui sont syntaxiquement des « valeurs » ont le droit d'avoir leurs types généralisées).

Toute expression ne satisfaisant aucune des règles du système de type présentées est invalide et ne pourra donc pas être évaluée.

A.2.1 Notations

Pour décrire formellement le système de types du langage mCaml, nous utilisons les notations qui sont ci-après définies.

La présentation des règles de typage de mCaml emprunte classiquement la syntaxe du « calcul des séquents », dont on peut trouver une introduction dans le chapitre 5 du livre [25].

• **Les types**

Un schéma de type, noté σ, est
- soit un type quantifié : $\forall t.\sigma$, ce qui signifie que pour tout type t appartenant à l'ensemble des types possibles, σ est un schéma de type (généralement t apparaît dans σ, même si ce n'est pas forcément obligatoire).
- soit un type (non quantifié) : t.

Un type est
- soit un nom (type simple), noté t (tout comme les types en général) ;
- soit une variable de type, notée α ;
- soit un type paramétré par un ou plusieurs types, noté $\phi\ t$ où ϕ qui représente l'ensemble des paramètres ; on pourra également noter $\phi\ t$ le type t en considérant que ϕ est un ensemble vide ;
- soit un type construit à l'aide d'une flèche, noté t `->` t ;
- soit un ensemble associatif de paires, dont le premier élément de la première paire est $type$ et est associé au « type utilisateur » (c'est-à-dire le type vu par le programmeur), ce type est particulier et n'est pas accessible par le programmeur (c'est-à-dire qu'il n'a pas la possibilité de l'exprimer d'après la grammaire du langage, tout comme pour les schémas de type).

• **Les constantes** sont notées c, c_0, c_1, c_2, c_n, etc.

• **Les nombres** sont notés n, n_0, n_1, n_2, n_n, etc., quand ils sont inconnus, ou leur représentation décimale lorsque c'est possible. Ils sont de type **`int`**, et on utilise par convention les entiers signés standards sur 64 bits.

• **Les booléens** sont notés **`true`** ou bien **`false`**.

• **Les variables** sont notées x, x_0, x_1, x_2, x_n, etc.

• **Les expressions du langage** sont notées e, e_0, e_1, e_2, e_n, etc.

• **Les champs d'enregistrements** sont notés f, f_0, f_1, f_2, f_n, etc.

• **Les constructeurs constants** sont notés C, C_0, C_1, C_2, C_n, etc.

• **Les constructeurs non constants** sont notés $C(e)$, $C_0(e)$, $C_1(e)$, $C_2(e)$, $C_n(e)$, etc.

• **Les environnements de typage des expressions** sont notés $\mathbb{E}$, $\mathbb{E}_0$, $\mathbb{E}_1$, $\mathbb{E}_2$, $\mathbb{E}_n$, etc. Ces environnements associent des noms de variables du langage mCaml à des schémas de type.

• **Les environnements lexicaux de types** sont notés $\mathbb{T}$, $\mathbb{T}_0$, $\mathbb{T}_1$, $\mathbb{T}_2$, $\mathbb{T}_n$, etc. Ces environnements contiennent tous les noms des types valides (les types primitifs plus les types déclarés par le programmeur). Un tel environnement est un ensemble de paires qui associe les bases des noms des types (*i.e.*, les noms de types sans les paramètres éventuels) aux noms complets correspondants.

• **Les environnements de typage des constructeurs** sont notés $\mathbb{C}$, $\mathbb{C}_0$, $\mathbb{C}_1$, $\mathbb{C}_2$, $\mathbb{C}_n$, etc. Ils contiennent les informations sur les constructeurs valides, c'est-à-dire les noms ainsi que le type auquel ils appartiennent et le type de leur argument éventuel.

• **Les environnements de typage des enregistrements** sont notés $\mathbb{R}$, $\mathbb{R}_0$, $\mathbb{R}_1$, $\mathbb{R}_2$, $\mathbb{R}_n$, etc. Ils contiennent les informations sur les champs d'enregistrements valides, c'est-à-dire les noms ainsi que le type auquel ils appartiennent et les types des valeurs associées.

• **Enrichissement d'un environnement et accès aux informations.** Si E est un environnement et $E' \triangleq (x,t) \triangleleft E$, alors E' est l'environnement E enrichi de la liaison (x,t), donc $E'(x) = t$.

Lorsqu'une association est absente d'un environnement, on rend $\bot$.

• **L'environnement vide** est noté $[]$. Ainsi, si $E = []$, alors $\forall w, E(w) = \bot$.

• **L'ensemble des variables libres de types de** t est noté $Vars(t)$.

Voici sa définition inductive.

$Vars(\alpha)$	=	$\alpha \triangleleft []$	α est une variable de type
$Vars(\alpha\ t)$	=	$\alpha \triangleleft []$	t est paramétré par la variable de type α
$Vars(p\ t)$	=	$Vars(p)$	t est paramétré par le type paramétré p
$Vars(\phi\ t)$	=	$\bigcup_{\forall t' \in \phi} Vars(t')$	t est paramétré par plusieurs types représentés par ϕ
$Vars(t)$	=	$[]$	t n'est pas une variable de type et n'est pas paramétré
$Vars(t_1 \texttt{ -> } t_2)$	=	$Vars(t_1) \cup Vars(t_2)$	t_1 et t_2 sont des types

• **L'ensemble des variables de types de** t est noté $Free(t)$.

Voici sa définition inductive.

$Free((\_,t) \triangleleft E)$	=	$Free(t) \cup Free(E)$	E est un environnement associant des noms à des types
$Free(\forall \phi.t)$	=	$Free(t) \backslash \phi$	$\forall \phi.t$ est un type quantifié
$Free(\alpha)$	=	$\alpha \triangleleft []$	α est une variable de type
$Free(\alpha\ t)$	=	$\alpha \triangleleft []$	t est paramétré par la variable de type α
$Free(p\ t)$	=	$Free(p)$	t est paramétré par le type paramétré p
$Free(\phi\ t)$	=	$\bigcup_{\forall t' \in \phi} Free(t')$	t est paramétré par plusieurs types représentés par ϕ
$Free(t)$	=	$[]$	t n'est pas une variable de type et n'est pas paramétré
$Free(t_1 \texttt{ -> } t_2)$	=	$Free(t_1) \cup Free(t_2)$	t_1 et t_2 sont des types

• **La généralisation d'un type** t **dans l'environnement** E est notée $Generalize(E,t)$.

Voici sa définition

$Generalize(E,t) = \forall(\alpha_1,...,\alpha_n).t$ où $\{\alpha_1 ... \alpha_n\} = Free(t) \backslash Free(E)$

• **Une instance du schéma de type** σ est notée $Instanciate(\sigma)$.

Voici sa définition.

$Instanciate(\forall \phi.t) = t[\forall \alpha_i \in \phi, \alpha_i \leftarrow t_i]$

Les variables de ϕ apparaissant dans t sont remplacées par des variables fraîches et la quantification disparaît.

• **Déterminer si une expression est expansive** permet de déterminer la possibilité de généraliser le type d'une variable lors de sa définition, selon la *value restriction* de Wright [87].

À propos de la *value restriction*
À cause des structures mutables, on ne peut pas généraliser toutes les expressions (au niveau du **let**).
Voici un exemple classique qui illustre le problème :

```
type 'a ref = { mutable contents : 'a }
let ref x = { contents = x }
let f a = let tmp = ref a in tmp.contents <- 4; tmp.contents <- true
```

La définition de `f` est rejetée, car `tmp.contents` ne doit pas pouvoir recevoir des nombres et des booléens. On observe alors que lors de la définition de `tmp`, si on avait généralisé le type, on n'empêcherait pas les deux affectations d'être invalides du point de vue des types.
Alors on choisit de ne pas généraliser certaines catégories d'expressions. Plus précisément, on n'autorise la généralisation que des expressions qui sont syntaxiquement des valeurs (au sens de Wright [87]), c'est-à-dire les constantes, les variables et les abstractions. Par exemple, les applications ne sont syntaxiquement pas des valeurs puisqu'il faut les évaluer pour connaître leurs valeurs de retour (et on n'est même pas sûr d'obtenir une valeur dans le cas général, puisqu'on ne peut pas décider de la terminaison.)
Pour relâcher les contraintes de la *value restriction*, J. Garrigue [31] résume la situation avant de donner sa proposition.

$Expansive(e)$ rend $\top$ si e est expansif, $\bot$ si non.

Voici sa définition inductive.

$Expansive(\texttt{x}) = \bot$
$Expansive(\texttt{c}) = \bot$
$Expansive(\texttt{e}_1\ \texttt{e}_2) = \top$
$Expansive(\texttt{fun x -> e}) = \bot$
$Expansive(\texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3) = Expansive(\texttt{e}_2) \vee Expansive(\texttt{e}_3)$
$Expansive(\texttt{let x = e}_1\ \texttt{in e}_2) = Expansive(\texttt{e}_1) \vee Expansive(\texttt{e}_2)$
$Expansive(\texttt{let rec x}_1\ \texttt{= fun x}_2\ \texttt{-> e}_1\ \texttt{in e}_2) = Expansive(\texttt{e}_2)$
$Expansive(\texttt{C(e)}) = Expansive(\texttt{e})$
$Expansive(\texttt{e}_1\ \texttt{; ... ; e}_n) = Expansive(\texttt{e}_n)$
$Expansive(\texttt{\{ f}_1\ \texttt{= e}_1\ \texttt{; ... ; f}_n\ \texttt{= e}_n\ \texttt{\}}) =$
$(\exists \texttt{f} \in (\texttt{f}_1, ..., \texttt{f}_n), \text{mutable}(\texttt{f})) \vee Expansive(\texttt{e}_1) \vee ... \vee Expansive(\texttt{e}_n)$
$Expansive(\texttt{e . f}) = Expansive(\texttt{e})$
$Expansive(\texttt{e}_1\ \texttt{. f <- e}_2) = \bot$
$Expansive(\texttt{for x = e}_1\ \texttt{to e}_2\ \texttt{do e}_3\ \texttt{done}) = \bot$
$Expansive(\texttt{while e}_1\ \texttt{do e}_2\ \texttt{done}) = \bot$
$Expansive(\texttt{e}_1\ \texttt{\&\& e}_2) = \bot$
$Expansive(\texttt{e}_1\ \texttt{|| e}_2) = \bot$
$Expansive(\texttt{not e}) = \bot$
$Expansive(\texttt{(e)}) = Expansive(\texttt{e})$
$Expansive(\texttt{match e with p}_1\ \texttt{-> e}_1\ \texttt{| ... | p}_n\ \texttt{-> e}_n) = Expansive(\texttt{e}_1) \vee ... \vee Expansive(\texttt{e}_n)$
$Expansive(\texttt{raise e}) = \bot$
$Expansive(\texttt{try e}_1\ \texttt{with x -> e}_2) = Expansive(\texttt{e}_1) \vee Expansive(\texttt{e}_2)$

A.2.2 Environnements initiaux

Voici les valeurs initiales des environnements :
$\mathbb{E}$ = [(+,**int->int->int**) ;(-,**int->int->int**) ;(*,**int->int->int**) ;(/,**int->int->int**) ;]
$\mathbb{T}$ = [(**int**,**int**) ; (**bool**,**bool**) ; (**exn**,**exn**) ;]
$\mathbb{C}$ = [(`Match_failure`, ($\emptyset \Rightarrow$ **exn**)) ; (`Division_by_zero`, ($\emptyset \Rightarrow$ **exn**))]
$\mathbb{R}$ = []

A.2.3 Les définitions de types

Les définitions de types enrichissent les environnements $\mathbb{T}$, $\mathbb{C}$ et $\mathbb{R}$ de fait retournent un triplet de nouveaux environnements $\mathbb{T}$'$\Big/\mathbb{C}$'$\Big/\mathbb{R}$'.

Définition d'un type somme (ou type variant)

Considérons la définition de type suivante

$$\textbf{type}\ \phi\ t = C_1\ |\ \ldots\ |\ C_{n-1}\ |\ C_n\ \textbf{of}\ t_n\ |\ \ldots\ |\ C_m\ \textbf{of}\ t_m$$

Les types sommes permettent de définir des constructeurs constants (sans argument) et non constants (avec un argument). Ces constructeurs sont unis par le fait d'être du même type.

Si

- (1) le type t n'existe pas déjà dans l'environnement de typage $\mathbb{T}$,
- (2) les paramètres de types (ϕ) sont uniques,
- (3) dans l'environnement $\mathbb{T}$ enrichi de l'existence de t, chacun des $t_n \ldots t_m$ existe,
- et (4) l'union des ensembles des variables de types de $t_n \ldots t_m$ est égal à l'ensemble des variables de types de t,

alors on obtient

- (5) l'environnement $\mathbb{T}$ enrichi de t,
- (6) ainsi que l'environnement $\mathbb{C}$ enrichi des informations de type associées aux constructeurs déclarés : les constructeurs $C_1 \ldots C_{n-1}$ sont de type t, les constructeurs $C_n \ldots C_m$ sont de type t et leurs arguments sont de types $t_n \ldots t_m$.

Remarques

- (1) *Cette interdiction est arbitraire, mais également appliquée en OCaml.*
- (3) L'enrichissement de l'environnement des définitions avec t permet de déclarer des types récursifs, c'est-à-dire des types qui peuvent apparaître dans leurs propres définitions.
- (4) L'union permet de vérifier qu'il n'y a pas de variable libre de type et que tous les paramètres de types du type défini sont utiles. *Contrairement à OCaml, nous choisissons de ne pas autoriser pour* mCaml *les paramètres « fantômes », c'est-à-dire les paramètres qui ne sont pas présents dans la définition du type. Ils seraient faciles à autoriser, mais nous éviterons d'introduire des types fantômes dans un potentiel cadre de développement de logiciels critiques.*

En notation formelle, voici la règle de typage obtenue :

Type-Constructor

$$\frac{\mathbb{T}(t) = \bot \qquad \forall t \in \phi.\exists! t \in \phi \qquad (t, \forall\phi.\phi\ t)\lhd\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_n \quad \ldots \quad (t, \forall\phi.\phi\ t)\lhd\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_m \qquad (Vars(t_n) \cup \ldots \cup Vars(t_m)) \equiv \phi}{\begin{array}{c}\mathbb{T},\mathbb{C},\mathbb{R} \vdash \ \texttt{type}\ \phi\ t = C_1 \mid \ldots \mid C_{n-1} \mid C_n\ \texttt{of}\ t_n \mid \ldots \mid C_m\ \texttt{of}\ t_m \\ \rightsquigarrow (t, \forall\phi.\phi\ t)\lhd\mathbb{T} \Big/ \begin{array}{l}(C_1, \forall\phi.\emptyset \Rightarrow \phi\ t)\lhd\ldots\lhd(C_{n-1}, \forall\phi.\emptyset \Rightarrow \phi\ t) \\ \lhd(C_n, \forall\phi.t_n \Rightarrow \phi\ t)\lhd\ldots\lhd(C_m, \forall\phi.t_m \Rightarrow \phi\ t)\lhd\mathbb{C}\end{array} \Big/ \mathbb{R}\end{array}}$$

Définition d'un type enregistrement

Considérons la définition de type suivante

$$\begin{array}{l}\texttt{type}\ \phi\ t = \texttt{\{}\quad f_1 \texttt{ : } t_1 \texttt{ ; } \ldots \texttt{ ; } f_n \texttt{ : } t_n \texttt{ ;} \\ \qquad\qquad\quad \texttt{mutable}\ f_m \texttt{ : } t_m \texttt{ ; } \ldots \texttt{ ; mutable}\ f_z \texttt{ : } t_z \texttt{ \}}\end{array}$$

Un enregistrement permet de combiner plusieurs valeurs en une seule valeur. Ces valeurs peuvent ensuite être accédées en lecture via les champs qui les contiennent. Les champs déclarés comme étant mutables peuvent être affectés avec de nouvelles valeurs.

Pour simplifier la notation, on définit la macro suivante qui est en fait la définition du type simplement sous une autre forme (mais sans les informations de mutabilité, qui seront enregistrées ailleurs).

$$\text{soit } T \triangleq \forall\phi.((type, \phi\ t)\lhd(f_1, t_1)\lhd\ \ldots\ \lhd(f_n, t_n)\lhd(f_m, t_m)\lhd\ \ldots\ \lhd(f_z, t_z)\lhd[])$$

T est un ensemble quantifié de paires associant un nom à un type. Le premier élément $(type, t)$ ne sert qu'à l'instantiation du type (car il faut instancier ensemble toutes les variables de types de la définition lorsqu'on en crée une valeur). T peut être utilisé de la même manière que les environnements, ainsi par exemple, $T(type)$ est le nom complet du type et $T(f_n)$ est le type du champ f_n.

Si l'environnement $\mathbb{T}$ enrichi de t permet d'affirmer que $t_1, \ldots, t_n, t_m, \ldots, t_z$ sont des types, et que l'union des ensembles des variables de types de $t_1, \ldots, t_n, t_m, \ldots, t_z$ est égal à l'ensemble des variables de types de t, **alors** on rend l'environnement $\mathbb{T}$ enrichi de (t, T) ainsi que l'environnement $\mathbb{C}$ dans lequel on associe chacun des champs à son type d'origine (pour pouvoir le retrouver), sans l'information de mutabilité et avec l'information de mutabilité. Ainsi un champ mutable apparaît deux fois dans l'environnement $\mathbb{C}$, une fois sous sa forme mutable et une fois sans information de mutabilité. Les champs n'apparaissant que sans information de mutabilité ne sont alors pas mutables.

Type-Record

$$\frac{\begin{array}{c}\text{let } T \triangleq \forall\phi.((type, \phi\ t)\lhd(f_1, t_1)\lhd\ldots\lhd(f_n, t_n)\lhd(f_m, t_m)\lhd\ldots\lhd(f_z, t_z)\lhd[]) \\ (t, T)\lhd\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_1 \quad \ldots \quad (t, T)\lhd\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_n \qquad (t, T)\lhd\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_m \quad \ldots \quad (t, T)\lhd\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_z \\ \vdash (Vars(t_1) \cup \ldots \cup Vars(t_n) \cup Vars(t_m) \cup \ldots \cup Vars(t_z)) \equiv \phi\end{array}}{\begin{array}{c}\mathbb{T},\mathbb{C},\mathbb{R} \vdash \ \texttt{type}\ \phi\ t = \texttt{\{}\quad f_1 \texttt{ :} t_1 \texttt{ ; } \ldots \texttt{ ; } f_n \texttt{ :} t_n \texttt{ ;} \\ \qquad\qquad\quad \texttt{mutable}\ f_m \texttt{ :} t_m \texttt{ ; } \ldots \texttt{ ; mutable}\ f_z \texttt{ :} t_z \texttt{ \}} \\ \rightsquigarrow (t, T)\lhd\mathbb{T} \Big/ \mathbb{C} \Big/ \begin{array}{l}(f_1, t)\lhd\ldots\lhd(f_n, t)\lhd(f_m, t)\lhd\ldots\lhd(f_z, t) \\ \lhd(mutable\ f_m, t)\lhd\ldots\lhd(mutable\ f_z, t)\lhd F\end{array}\end{array}}$$

A.2.4 Le typage des définitions

Le typage des définitions requiert l'ensemble des environnements de typage, et enrichit l'environnement de typage des expressions qui est noté $\mathbb{E}$.

LET-GEN
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash Expansive(e) = \bot}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{let}\ x = e) \rightsquigarrow (x, Generalize(\mathbb{E}, t)) \lhd \mathbb{E}}$$

LET
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash Expansive(e) = \top}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{let}\ x = e) \rightsquigarrow (x, t) \lhd \mathbb{E}}$$

REC-GEN
$$\frac{(x_1, t_1 \texttt{ -> } t_2) \lhd (x_2, t_1) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t_2}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{let rec}\ x_1 = \texttt{fun}\ x_2 \texttt{ -> } e) \rightsquigarrow (x_1, Generalize(\mathbb{E}, t_1 \texttt{ -> } t_2)) \lhd \mathbb{E}}$$

A.2.5 Le typage des expressions

Le typage des expressions associe un type à une expression. Pour faire, on introduit un environnement de typage des expressions, que nous notons $\mathbb{E}$, en plus des environnements précédemment utilisés. Cet environnement $\mathbb{E}$ associe des noms de variables à des schémas de type.

Variables Pour avoir le type d'une variable, il faut récupérer le schéma de type associé à la variable dans l'environnement de typage des expressions, et en prendre une instance. Par exemple, si x est la fonction identité définie par `let x = fun` $\texttt{x}_1$ `->` $\texttt{x}_1$, son schéma de type pourra être $\forall\alpha.\alpha$ `->`α. Son instance sera alors le type mCaml α `->`α (sous-entendu $\exists\alpha.\alpha$ `->`α).

VAR-INST
$$\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash x : Instanciate(\mathbb{E}(x))$$

Nombres entiers Soit n un nombre entier. Alors son type est **int**.

INTEGER
$$\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash n : \texttt{int}$$

Booléens Les deux booléens **true** et **false** sont de type **bool**.

TRUE
$$\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \texttt{true} : \texttt{bool}$$

FALSE
$$\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \texttt{false} : \texttt{bool}$$

Constructeurs

Constructeurs constants Un constructeur constant est un constructeur de type variant sans paramètre. Soit C un constructeur. Le nom sans paramètres de son type peut être retrouvé dans l'environnement $\mathbb{C}$ à l'aide de $\mathbb{C}(C)$. Ensuite $\mathbb{T}$ permet d'obtenir le nom complet (avec les paramètres éventuels de type) avec $\mathbb{T}(\mathbb{C}(C))$. C'est ce type instancié qui est le type donné au constructeur. L'instantiation est nécessaire car le type est éventuellement paramétré. Le type de C est donc $Instanciate(\mathbb{T}(\mathbb{C}(C)))$.

$$\frac{\text{CONSTANT-CONSTRUCTOR-INST}\quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\emptyset \Rightarrow t) \triangleq Instanciate(\mathbb{C}(C))}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash C : t}$$

Constructeurs non-constants Le typage d'un constructeur C avec une expression e en paramètre demande au constructeur d'attendre effectivement un argument d'un type compatible avec celui de e. On récupère les informations de type de C à l'aide de $\mathbb{C}(C)$, et on doit obtenir un couple (t_σ, t_{σ_e}) dont le premier élément indique le nom non paramétré du type d'appartenance de C et dont le second élément indique le type de l'argument attendu. Ensuite, $\mathbb{T}(t_\sigma)$ permet de retrouver le nom complet du type à qui le constructeur appartient. L'instantiation se fait alors en concordance avec le type de l'argument, on obtient ainsi $(t, t_e) \triangleq Instanciate(\mathbb{T}(t_\sigma), t_{\sigma e})$. Enfin, t_e est le type de e, et t est le type de $C\,(e)$.

$$\frac{\text{CONSTRUCTOR-INST}\quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (t_e \Rightarrow t) \triangleq Instanciate(\mathbb{C}(C)) \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t_e}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash C\,(e) : t}$$

Enregistrements Considérons l'expression **{** f_1 **=** e_1 **;** … **;** f_n **=** e_n **}** qui est un enregistrement et nommons-la e. Nous commençons par vérifier que chacun des champs d'enregistrement est associé au même type t_ρ dans l'environnement $\mathbb{R}$. Ensuite, nous prenons récupérons une instance t du type complet correspondant à t_ρ qui est $\mathbb{T}(t_\rho)$, ce qui donne $t \triangleq Instanciate(\mathbb{T}(t_\rho))$. Le type t contient des informations et peut être utilisé comme un environnement pour récupérer les informations dont nous avons besoin (*cf.* la définition d'un type enregistrement plus haut). Le type t contient en effet l'ensemble des champs de l'enregistrement avec leurs types associés instanciés ainsi que le nom complet instancié du type t_ρ qui est alors $t(type)$. On vérifie alors que chacune des expressions a le type du champ auquel elle a été liée. Ensuite on vérifie l'exhaustivité des champs (on fait l'appel pour que personne ne manque) et l'unicité des champs (on ne veut pas qu'un champ apparaisse plusieurs fois). Ainsi, pour chacun des champs f de $\mathbb{T}(t_\rho)\backslash(type,\ t_\rho)$ [1] (donc là il ne reste plus que les champs associés à leurs types), il doit exister un unique champ f' (associé à un quelconque type) tel que $f = f'$. Enfin, le type mCaml de e est alors $t(type)$, mais on mettra directement t dans l'environnement de typage des expressions pour pouvoir disposer ultérieurement des informations sur les types instanciés des champs de l'enregistrement [2].

1. on prend $\mathbb{T}(t_\rho)$ plutôt que t pour pouvoir facilement exprimer la privation du premier champ $(type,\ t_\rho)$, car le premier champ de t n'est pas $(type,\ t_\rho)$ mais une instance. En effet, nous n'utilisons que les noms des champs de l'enregistrement.

2. Une autre façon de faire est d'introduire un environnement supplémentaire pour associer le type $t(type)$ à t, mais cela risque d'alourdir la notation en introduisant des environnements à éléments mutables.

Record-inst
$$\frac{\text{let } e \triangleq \texttt{\{}\ f_1 = e_1\ \texttt{;}\ \ldots\ \texttt{;}\ f_n = e_n\ \texttt{\}} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_\rho \triangleq \mathbb{R}(f_1) \quad \ldots \quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \mathbb{R}(f_1) = \ldots = \mathbb{R}(f_n) \\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \forall (f,\_) \in (t \backslash (type,\ t_\rho)),\ \exists !(f',\_) \in e,\ f = f' \\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t \triangleq Instanciate(\mathbb{T}(t_\rho)) \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t(f_1) \quad \ldots \quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_n : t(f_n)}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t}$$

Accès en lecture aux champs d'enregistrements Si e est de type t et $t(f)$ est valide (ce qui veut dire que t est un type qui contient le champ f), alors $e\texttt{.}f$ est de type $t(f)$ (c'est-à-dire le type associé à f dans t). *L'accès à un champ inexistant ou mal typé ne doit pas arriver.*

Record-Access
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t(f)}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e\texttt{.}f : t(f)}$$

Accès en écriture aux champs d'enregistrements Pour l'expression $(e_1\texttt{.}f\ \texttt{<-}\ e_2)$, si e_1 est de type t_1, f est bien un champ mutable du type t_1, et e_2 est bien du type du champ f, alors l'expression est de type **unit**.

Record-Modify
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t_1 \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \mathbb{R}(mutable\ f) \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_1(f) = t_2 \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t_2}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (e_1\texttt{.}f\ \texttt{<-}\ e_2) : \texttt{unit}}$$

λ-abstraction Pour l'expression ($\texttt{fun}\ x\ \texttt{->}\ e$), si x est de type t_1, e est de type t_2, alors l'expression est de type $(t_1\ \texttt{->}\ t_2)$.

Abstraction
$$\frac{(x : t_1) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t_2}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{fun}\ x\ \texttt{->}\ e) : (t_1\ \texttt{->}\ t_2)}$$

Application Pour l'expression $(e_1\ e_2)$, si e_1 est de type $t_1\ \texttt{->} t_2$ et e_2 est de type t_1, alors l'expression est de type t_2.

Application
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t_1\ \texttt{->}\ t_2 \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t_1}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (e_1\ e_2) : t_2}$$

Liaison locale Les liaisons sont les endroits où se décide la généralisation du type d'une expression. Si l'expression est expansive, alors on ne généralise pas, sinon on généralise. En fait, une expression est expansive lorsque sa généralisation peut potentiellement entraîner une erreur de typage. Par exemple, l'application est expansive notamment parce que si elle retourne une valeur mutable, il ne vaut mieux pas qu'elle soit affectable par la suite avec des valeurs de types différents.

Let-in-gen
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t_1 \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash Expansive(e_1) = \bot \qquad (x, Generalize(\mathbb{E}, t_1)) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t_2}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : t_2}$$

Let-in
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t_1 \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash Expansive(e_1) = \top \qquad (x,t_1) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t_2}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : t_2}$$

Rec-in-gen
$$\frac{(x_1, t_2\texttt{->}t_1) \lhd (x_2,t_2) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t_1 \qquad (x_1, Generalize(\mathbb{E}, t_2\texttt{->}t_1)) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{let rec}\ x_1 = \texttt{fun}\ x_2\ \texttt{->}\ e_1\ \texttt{in}\ e_2) : t}$$

Valeur du type unit

Unit
$$\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \texttt{()} : \texttt{unit}$$

Séquence Pour la séquence e_1 `;` e_2, on choisit d'obliger le type de e_1 à être de type **`unit`**. Si e_1 n'est pas de type **`unit`**, il suffira pour le programmeur de le transtyper.

Sequence
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : \texttt{unit} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (e_1\ \texttt{;}\ e_2) : t}$$

Conditionnelle La conditionnelle oblige la condition à être de type **`bool`** et ses deux branches à être de même type.

Conditional
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : \texttt{bool} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_3 : t}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : t}$$

Conjonction et disjonction séquentielles La conjonction et la disjonction ne s'appliquent qu'à des expressions booléennes.

Sequential-Conjunction
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : \texttt{bool} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : \texttt{bool}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (e_1\ \texttt{\&\&}\ e_2) : \texttt{bool}}$$

Sequential-Disjunction
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : \texttt{bool} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : \texttt{bool}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (e_1\ \texttt{||}\ e_2) : \texttt{bool}}$$

Négation booléenne La négation booléenne ne s'applique qu'aux booléens.

Negation
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : \texttt{bool}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \texttt{not}(e) : \texttt{bool}}$$

Boucle itérative bornée On a choisit d'obliger le corps de la boucle à être de type **`unit`**.

For
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : \texttt{int} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : \texttt{int} \qquad (x : \texttt{int}) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_3 : \texttt{unit}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done}) : \texttt{unit}}$$

Boucle itérative non bornée On a choisi d'obliger le corps de la boucle à être de type **unit**.

$$\textsc{While}\ \frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : \textbf{bool} \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : \textbf{unit}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\textbf{while}\ e_1\ \textbf{do}\ e_2\ \textbf{done}) : \textbf{unit}}$$

Les définitions des exceptions enrichissent l'environnement de typage des constructeurs. On obtient donc en retour un nouvel environnement de typage des constructeurs.

$$\textsc{Exception}_1\ \mathbb{T},\mathbb{C} \vdash \textbf{exception}\ C \rightsquigarrow (C, \textbf{exn}) \lhd \mathbb{C} \qquad \textsc{Exception}_2\ \frac{\mathbb{T},\mathbb{C} \vdash \mathbb{T}(t)}{\mathbb{T},\mathbb{C} \vdash \textbf{exception}\ C\ \textbf{of}\ t \rightsquigarrow (C, (\textbf{exn}, t)) \lhd \mathbb{C}}$$

Typage des expressions utilisant explicitement les exceptions

$$\textsc{Try}\ \frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t \qquad (x, \textbf{exn}) \lhd \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_2 : t}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\textbf{try}\ e_1\ \textbf{with}\ x\ \textbf{->}\ e_2) : t}$$

$$\textsc{Raise}\ \frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : \textbf{exn}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \textbf{raise}\ e : Instanciate(\forall\alpha.\alpha)}$$

Typage du filtrage par motifs

Le typage des motifs nécessite des règles de typage particulières pour les filtres-motifs du fait qu'ils enrichissent l'environnement et qu'ils peuvent être imbriqués. Voici donc les règles de typage des filtres.

Une variable est associée à un type inconnu et cette association est rendue dans l'environnement résultant. Une constante est simplement associée à son type.

$$\textsc{pat-Var}\ \frac{t = Instanciate(\forall\alpha.\alpha)}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash x : t \Big/ (x,t) \lhd \mathbb{E}} \qquad \textsc{pat-const-Integer}\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash n : \textbf{int} \Big/ \mathbb{E} \qquad \textsc{pat-const-Unit}\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \textbf{()} : \textbf{unit} \Big/ \mathbb{E}$$

$$\textsc{pat-const-True}\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \textbf{true} : \textbf{bool} \Big/ \mathbb{E} \qquad \textsc{pat-const-False}\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \textbf{false} : \textbf{bool} \Big/ \mathbb{E}$$

$$\textsc{pat-const-Constructor}\ \frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (\emptyset \Rightarrow t) \triangleq Instanciate(\mathbb{C}(C))}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash C : t \Big/ \mathbb{E}}$$

Comme le constructeur non constant contient un autre motif, c'est l'autre motif qui se charge d'éventuellement enrichir l'environnement. On vérifie bien sûr que le type du constructeur est valide, y compris pour son argument.

PAT-CONSTRUCTOR
$$\frac{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash (t_a \Rightarrow t) \triangleq Instanciate(\mathbb{C}(C)) \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash p : t_a \Big| \mathbb{E}'}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash C(p) : t \Big| \mathbb{E}'}$$

Le filtrage des enregistrements contraint tous les champs à appartenir à la même définition de type et à être associés à des motifs du bon type. Un champ ne peut pas apparaître plusieurs fois mais des champs peuvent être omis.

PAT-RECORD
$$\frac{\begin{array}{c}\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_\rho \triangleq \mathbb{R}(f_1) \quad \ldots \quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t_\rho \triangleq \mathbb{R}(f_n) \qquad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash t \triangleq Instanciate(\mathbb{T}(t_\rho)) \\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash p_1 : t(f_1) \Big| \mathbb{T}_1 \quad \ldots \quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash p_n : t(f_n) \Big| \mathbb{E}_n \qquad (\mathbb{E}_1 \cap \ldots \cap \mathbb{E}_n) = \mathbb{E}\end{array}}{\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash \vdash \texttt{\{}\ f_1 = p_1\ \texttt{;}\ \ldots\ \texttt{;}\ f_n = p_n\ \texttt{\}} : t \Big| (\mathbb{E}_1 \cup \ldots \cup \mathbb{E}_n)}$$

Le typage du filtrage dans son ensemble Chaque filtre des branches doit être de même type que le type de l'expression dont la valeur est filtrée. Chaque expression des branches doit être de même type.

MATCH
$$\frac{\begin{array}{c}\mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash e : t' \\ \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash p_1 : t' \Big| \mathbb{E}_1 \quad \ldots \quad \mathbb{E},\mathbb{T},\mathbb{C},\mathbb{R} \vdash p_n : t' \Big| \mathbb{E}_n \qquad \mathbb{E}_1,\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_1 : t \quad \ldots \quad \mathbb{E}_n,\mathbb{T},\mathbb{C},\mathbb{R} \vdash e_n : t\end{array}}{\mathbb{E} \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) : t}$$

A.3 Les levées d'exceptions dans la sémantique opérationnelle

Cette section est une partie déportée de la sémantique opérationnelle présentée dans le chapitre III.

Les règles suivantes sont exhaustives pour toutes les expressions du langage susceptibles de lever une exception (par exemple, l'évaluation d'une constante ne peut pas lever une exception). Les trois premières règles sont les mêmes que celles données à titre d'exemple dans la section III.1.2.4 (page 51).

APPLICATION-EXCEPTION
$$\frac{\mathcal{E},\mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E},\mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_1}$$

APPLICATION-EXCEPTION
$$\frac{\mathcal{E},\mathcal{M}_0 \vdash e_1 \rightsquigarrow \Big\langle x, e_3, E_1 \Big\rangle \Big| \mathcal{M}_1 \qquad \mathcal{E},\mathcal{M}_1 \vdash e_2 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E},\mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}$$

Application-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \left\langle x, e_3, E_1 \right\rangle \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big| \mathcal{M}_2 \qquad (x, v_2) \lhd \mathcal{E}, \mathcal{M}_2 \vdash e_3 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_3}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_3}$$

Application-rec-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \left\langle x_1, x_2, e_3, E_1 \right\rangle \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}$$

Application-rec-Exception
$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \left\langle x_1, x_2, e_3, E_1 \right\rangle \Big| \mathcal{M}_1 \\ \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v_2 \Big| \mathcal{M}_2 \qquad (x_1, \left\langle x_1, x_2, e_3, E_1 \right\rangle) \lhd (x_2, v_2) \lhd \mathcal{E}, \mathcal{M}_2 \vdash e_3 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_3\end{array}}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_3}$$

Conditional-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}$$

Conditional-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{true} \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}$$

Conditional-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{false} \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_3 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}$$

Let-in-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}$$

Let-in-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big| \mathcal{M}_1 \qquad (x, v_1) \lhd \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}$$

Rec-in-Exception
$$\frac{(x_1, \left\langle x_1, x_2, e_1, E \right\rangle) \lhd \mathcal{E}, \mathcal{M}_0 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\textbf{let rec}\ x_1 = \textbf{fun}\ x_2\ \texttt{->}\ e_1\ \textbf{in}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}$$

Sequential-Conjunction-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ \textbf{\&\&}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_1}$$

Sequential-Conjunction-Exception
$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{true} \Big| \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ \textbf{\&\&}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big| \mathcal{M}_2}$$

Sequential-Disjunction-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ \texttt{||}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

Sequential-Disjunction-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{false} \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\ \texttt{||}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_2}$$

Negation-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{not(}e\textbf{)} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

Sequence-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{(}e_1\ \texttt{;}\ e_2\textbf{)} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

Sequence-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_{n-1} \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_n}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{(}e_1\ \texttt{;}\ e_2\textbf{)} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_n}$$

Record-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{\{}f_1 \textbf{=} e_1\ \texttt{;}\ \ldots\ \texttt{;}\ f_n \textbf{=} e_n\ \textbf{\}} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

Record-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \quad \ldots \quad \mathcal{E}, \mathcal{M}_{n-2} \vdash e_{n-1} \rightsquigarrow v_{n-1} \Big/ \mathcal{M}_{n-1} \qquad \mathcal{E}, \mathcal{M}_{n-1} \vdash e_n \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_n \qquad n \leq m}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{\{}f_1 \textbf{=} e_1\ \texttt{;}\ \ldots\ \texttt{;}\ f_n \textbf{=} e_m\ \textbf{\}} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_n}$$

Record-Access-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash e\textbf{.}f \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

Record-Modify-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\textbf{.}f\texttt{<-}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

Record-Modify-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v_1 \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash (e_1\textbf{.}f\texttt{<-}\ e_2) \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_2}$$

Constructor-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash C\textbf{(}e\textbf{)} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

For-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{for}\ x \textbf{=} e_1\ \textbf{to}\ e_2\ \textbf{do}\ e_3\ \textbf{done} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_1}$$

For-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v1 \Big/ \mathcal{M}_1 \qquad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash \textbf{for}\ x \textbf{=} e_1\ \textbf{to}\ e_2\ \textbf{do}\ e_3\ \textbf{done} \rightsquigarrow \textbf{Exception}\ v \Big/ \mathcal{M}_2}$$

For-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v1 \Big| \mathcal{M}_1 \quad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v2 \Big| \mathcal{M}_2 \quad \mathcal{E}, \mathcal{M}_2 \vdash v1 > v2 \rightsquigarrow \texttt{false} \Big| \mathcal{M}_2 \quad (x, v_1) \lhd \mathcal{E}, \mathcal{M}_2 \vdash e3 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_3}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_3}$$

For-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow v1 \Big| \mathcal{M}_1 \quad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow v2 \Big| \mathcal{M}_2 \quad \mathcal{E}, \mathcal{M}_2 \vdash v1 > v2 \rightsquigarrow \texttt{false} \Big| \mathcal{M}_2 \quad (x, v_1) \lhd \mathcal{E}, \mathcal{M}_2 \vdash e3 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_3 \quad \mathcal{E}, \mathcal{M}_3 \vdash \texttt{for}\ x = v_1 + 1\ \texttt{to}\ v_2\ \texttt{do}\ e_3\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_4}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_4}$$

While-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_1}$$

While-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{true} \Big| \mathcal{M}_1 \quad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}$$

While-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e_1 \rightsquigarrow \texttt{true} \Big| \mathcal{M}_1 \quad \mathcal{E}, \mathcal{M}_1 \vdash e_2 \rightsquigarrow \texttt{()} \Big| \mathcal{M}_2 \quad \mathcal{E}, \mathcal{M}_2 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_3}{\mathcal{E}, \mathcal{M}_0 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_3}$$

Match-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_1}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}$$

Match-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1 \quad \dfrac{\mathcal{E} \vdash PAT(v, p_1) \rightsquigarrow \mathcal{E}' \quad \mathcal{E}' \neq [\,]}{\mathcal{E}', \mathcal{M}_1 \vdash e_1 \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}$$

Match-Exception

$$\frac{\mathcal{E}, \mathcal{M}_0 \vdash e \rightsquigarrow v \Big| \mathcal{M}_1 \quad \mathcal{E} \vdash PAT(v, p_1) \rightsquigarrow [\,] \quad \ldots \quad \mathcal{E} \vdash PAT(v, p_{k-1}) \rightsquigarrow [\,] \quad \mathcal{E} \vdash PAT(v, p_k) \rightsquigarrow \mathcal{E}' \quad \mathcal{E}' \neq [\,] \quad \mathcal{E}', \mathcal{M}_1 \vdash e_k \rightsquigarrow \texttt{Exception}\ v \quad k \leq n}{\mathcal{E}, \mathcal{M}_0 \vdash (\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n) \rightsquigarrow \texttt{Exception}\ v \Big| \mathcal{M}_2}$$

ANNEXE B

Notes explicatives

B.1 À propos de la ZAM, machine virtuelle OCaml

Les caractéristiques techniques de la ZAM décrites dans cette section sont extraites des trois sources suivantes :

1. *le rapport de Leroy sur l'expérience avec la machine ZAM de « première génération » [50].*
2. *le fichier d'implantation en langage C de la ZAM de « seconde génération » fournie dans la distribution officielle du langage OCaml :* `ocaml-3.12.0/byterun/interp.c`.
3. *le document de Clerc sur les instructions de la machine de seconde génération [17].*

B.1.1 Historique

La machine virtuelle OCaml (ou *OCaml virtual machine* en anglais) est aussi appelée ZAM en raison de ses origines. ZAM est l'acronyme de *Zinc Abstract Machine*, tandis que Zinc est donné au projet d'implantation d'une nouvelle version plus légère et plus performante de CAML[50]. La ZAM est la machine virtuelle de Caml Light, le langage qui succède à CAML[21] qui était basé sur la « machine abstraite catégorique » plus connue sous le nom de CAM pour « Categorical Abstract Machine » [19, 20], ce qui explique (en partie) le nom du langage.

OCaml, successeur de Caml Light, a apporté la capacité de produire des exécutables pour divers machines matérielles courantes, comme X86 ou PowerPC, tout en gardant la machine virtuelle de Caml Light. Deux intérêts immédiats justifient le fait de garder une machine virtuelle. La première est que la machine virtuelle est implantée en langage C, ce qui la rend assez facile à porter sur diverses architectures : il suffit que l'implantation de la ZAM compile pour obtenir une machine virtuelle OCaml. La seconde est que la compilation est plus rapide, car plus simple du fait qu'elle fait moins de travail d'optimisations, ce qui peut être utile pour certains programmes.

En comparant les instructions de la machine virtuelle de Caml Light avec celle d'OCaml, on constate que les différences fondamentales sont l'ajout d'instructions pour le paradigme de programmation par objets apporté par OCaml. La machine virtuelle OCaml est parfois appelée ZAM2, du fait de ces différences. *Dans cette thèse, « ZAM » désigne en général la machine virtuelle OCaml.*

B.1.2 Caractéristiques générales

La machine virtuelle OCaml est une machine à pile qui possède 146 instructions spécifiques. Environ 60% d'entre elles sont des instructions pouvent être simulées par les autres instructions. Par exemple, exécuter l'instruction PUSHACC7 est équivalent à exécuter à la suite les deux instructions

PUSH et ACC7. Avoir davantage d'instructions permet de réduire le travail de la machine virtuelle (au lieu de lire deux instructions, donc passer deux fois par la boucle d'interprétation, elle n'en lit qu'une seule, ce qui économise quelques cycles machine) mais aussi de réduire la taille du code machine (on a une sorte d'algorithme simple de compression par factorisation). Ceci est dit sous l'hypothèse que le nombre d'instructions est inférieur à 256, ce qui permet de coder son numéro sur un octet. Toutefois, pour des raisons d'alignement mémoire, on peut remarquer que pour les binaires produits par le compilateur `ocamlc` les instructions sont codées sur 4 octets plutôt qu'un seul, ce qui indique que 3 de ces 4 octets valent toujours 0.

La ZAM possède sept registres :

- ACCU : c'est en quelque sorte le sommet de la pile. S'il n'existait pas, les valeurs qu'il contient seraient simplement au sommet de la pile, et on adapterait les instructions en conséquence. L'intérêt d'avoir un registre est de garantir, lorsque le compilateur C le permet puisque la première et dominante implantation est en langage C, que ce sommet de pile est dans un registre du processeur physique, ce qui permet d'y accéder plus rapidement que s'il se situait dans la mémoire vive.
  - 115 instructions lisent ou écrivent dans ce registre.
  - 96 instructions lisent ou écrivent dans la pile.
- PC : le pointeur de code qui indique quelle instruction doit être interprétée.
- SP : le pointeur de pile, indiquant le niveau courant de la pile.
- TRAPSP : le pointeur vers le rattrapeur d'exception le plus haut dans la pile. Il permet de dépiler jusqu'au rattrapeur d'exception le plus haut en temps constant. Lors de la pose d'un nouveau rattrapeur d'exception, l'ancien est sauvegardé sur la pile.
- EXTRAARGS : le compteur d'arguments en attente d'application. Lors d'une application, il se peut que le nombre d'arguments soit supérieur au nombre d'arguments attendu par la fonction invoquée.
  Exemple :

  ```
  let id1 x = x
  let id2 x = x
  let r = id1 id2 42
  ```

  La fonction identitée (`id1`) n'attend qu'un seul argument mais est invoquée avec deux arguments (`id1 id2 42`). Cela est valide puisque `id1 id2` rendra la fonction `id2` qui attend un argument et donc peut bien recevoir le `42`.
- ENV : un pointeur destiné à pointer sur l'environnement courant (représenté par un tableau de valeurs).
  Lorsqu'on applique une fermeture, l'environnement courant devient celui de la fermeture, ce qui permet d'évaluer le code du corps de la fonction que la fermeture a capturée.
- GLOBALDATA : le pointeur vers le tableau des valeurs globales.

Les particularités de la ZAM qui ne se trouvent pas sur toutes les machines sont essentiellement le système d'application de fonctions et de représentation des fermetures ainsi que les quelques instructions servant au modèle objets du langage OCaml.

La représentation des valeurs est uniforme. Chaque valeur est soit un entier, soit un pointeur vers un bloc de données.

Les entiers sont distingués par le bit de poids le plus faible : s'il est à 1, c'est que ce n'est pas une adresse mémoire puisque tous les mots font un nombre pair d'octets.

Un bloc de données possède un entête permettant de savoir sa taille et ce qu'il contient (fermeture, constructeur de type somme, chaîne de caractères, nombre flottant, etc.)

La machine possède une instruction (`ISINT`) dédiée à la distinction « entier versus pointeur ».

Diverses implantations de la ZAM Il existe un certain nombre d'implantations de la ZAM, en voici quelques exemples par ordre chronologique croissant de naissance :

- en C, `ocamlrun` : L'originale fournie dans la distribution standard d'OCaml [52], par l'INRIA, est certainement la plus diffusée et utilisée. Il s'agit donc de la machine de référence. À ce jour, elle est maintenue et la distribution OCaml continue d'évoluer, mais les spécifications de la machine sont assez stables (c'est surtout le langage OCaml qui évolue).
- en OCaml, *dynlink* : Développé à la fin des années 1990, un module de chargement dynamique de code (pour permettre de charger et exécuter du *bytecode* OCaml au sein d'un programme compilé vers du code natif) contient une machine virtuelle OCaml développée en OCaml. Ce module fait partie du projet GwML [1] de Le Fessant.
- en Java, *Cadmium* : Developpé et distribué par Clerc [16], Cadmium est un projet qui permet de faire interagir OCaml et Java. La distribution comprend notamment un compilateur d'OCaml pour la JVM ainsi qu'une ZAM implantée en Java.
- en JavaScript, *O'Browser* : Developpé par Canou [11] au sein du projet Ocsigen visant à programmer le web en OCaml, O'Browser fournit une ZAM implantée en JavaScript afin de pouvoir programmer les manipulation sur le DOM en OCaml.
- en OCaml, *Zamcov* : Développé initialement au sein du projet Couverture, le système Zamcov comprend des outils pour la couverture structurelle de code OCaml. Zamcov fournit une ZAM implantée en OCaml, ce qui permet à l'ensemble des outils du projet Zamcov d'être implanté en un seul langage.
- en assembleur PIC18, *OCAPIC* : Développé par Vaugon [82] dans le but de pouvoir programmer les micro-contrôleurs PIC18 en OCaml, le projet OCAPIC fournit un ensemble d'outils pour réduire la taille du bytecode ainsi qu'une ZAM en assembleur PIC18. La particularité de cette machine est de fonctionner avec des mots 16 bits.

1. `http://web.archive.org/web/20061208050513/http://pauillac.inria.fr/~lefessan/src/ocamldev/11_gwml.tar.gz`

B.2 Appels terminaux et appels récursifs terminaux

B.2.1 Appels terminaux

Dans la définition d'une fonction, si la dernière action effectuée est un appel à une fonction, on appelle cet appel un « appel terminal » du fait de sa position.

Dans la définition de la fonction `f` qui se trouve ci-après, on remarque que :

- L'appel à `g` est terminal, mais l'appel à `h` ne l'est pas.
- `42` est en position terminale mais ce n'est pas un appel. En effet, rien n'oblige à ce que la dernière action dans une fonction soit un appel.
- Plusieurs expressions peuvent être en position terminale dans une fonction.

```
let f = fun c x ->
  if c then
    g (h x)
  else
    42
```

Enfin, on remarque que, dans certains cas, il est impossible de connaître statiquement la fonction appelée. Par exemple, dans le code suivant, la fonction appelée en position terminale est inconnue au niveau du site d'appel.

```
let f = fun t x ->
  let g = t.contents in
    g x
```

B.2.2 Appels récursifs terminaux

Un appel est terminal et récursif lorsqu'il se trouve en position terminale et qu'il est un appel à la fonction courante.

Plus généralement, un appel est terminal et récursif lorsqu'il se trouve en position terminale et qu'il est un appel à une fonction G faisant appel à la fonction courante F. Si G = F, cela revient à la défintion précédente. Si non, G et F sont des fonctions mutuellement récursives.

B.2.3 Optimisation des appels terminaux

Dans le cas général des appels de fonctions, on enregistre l'adresse de retour pour qu'on puisse reprendre où on se trouve une fois que l'appel a rendu son résultat.

Dans le cas particulier des appels qui sont en position terminale, on remarque deux choses :

1. Lorsqu'on reviendra, on se contentera de rendre le résultat qu'on viendra de récupérer, donc l'espace utilisé par la fonction courante sur la pile sera inutilisé (et de toutes façons supprimé). Donc autant récupérer l'espace utilisé sur la pile avant d'effectuer l'appel, surtout si la pile n'a pas une grande capacité.
2. L'effort de revenir au site d'appel pourrait être évité, car on peut factoriser l'effort du travail de retour du résultat.

Lorsqu'un appel est récursif, en particulier si c'est une implantation d'un algorithme récursif, on aura une potentielle grande consommation de la pile (dépendant linéairement du nombre d'appels effectué).

L'optimisation des appels récursifs terminaux consiste donc à récupérer immédiatement l'espace dans la pile occupé par la fonction courante pour que les boucles implantées à l'aide d'appels récursifs terminaux ne consomment pas plus de pile que les boucles itératives (**`for`**, **`while`**). *Les appels récursifs non terminaux ne sont pas concernés par l'optimisation.*

Cette optimisation peut dans le principe être appliquée à l'ensemble des appels terminaux, y compris lorsqu'ils ne sont pas récursifs. Cependant, mettre l'optimisation en pratique dépend de la cible de compilation. Par exemple, certaines machines n'ont pas d'instruction spécifiques pour les appels terminaux et n'autorisent pas non plus les sauts n'importe où dans le programme, comme c'est le cas pour la machine virtuelle Java (JVM) [54]. En revanche, la ZAM possède des instructions spécifiques pour les appels terminaux récursifs ou non récursifs, sauf pour les appels externes qui n'utilisent cependant pas la même pile d'exécution (détaillé en sections B.2.4 et B.2.5).

> **À propos de la JVM et des appels terminaux**
> Selon les spécifications de la JVM, il n'y a aucune instruction spécifique pour les appels terminaux. On regarde alors du côté des sauts directs pour lesquels il existe deux instructions `goto` et `goto_w`. Comme la seconde est capable d'encoder directement la première (seule les valeurs admissibles pour l'argument change entre les deux instructions), on peut ne considérer que la seconde. La contrainte de ces sauts directs est qu'ils ne sont autorisés qu'au sein d'une même méthode[54, chapitre 6], c'est-à-dire qu'on ne peut pas faire un saut vers le code d'une autre méthode. En conséquence, cela peut faire penser à compiler plusieurs fonctions vers une même méthode afin de pouvoir sauter de l'une à l'autre directement. Cependant, la taille des méthodes est bornée à 65536 octets[54, chapitre 4], ce qui fait qu'il est généralement impossible de compiler l'ensemble d'un programme dans une seule méthode [a]. On peut toutefois compiler les fonctions mutuellement récursives dont le code n'est pas trop volumineux au sein d'une même méthode afin de pouvoir bénéficier d'une consommation de pile constante spécifiquement pour ces fonctions.
>
> *a.* La limite à 65536 octets pourra éventuellement être revue à la hausse, d'après que c'est indiqué dans les spécifications. Pour Java SE 7 en 2012, cette limite est toujours la même qu'en 1996, année de la première édition des spécifications de la JVM qui peut être récupérée à l'URL suivante : `http://web.archive.org/web/20070216100425/http://java.sun.com/docs/books/vmspec/download/vmspec.html.tar.gz`.

B.2.4 Les appels terminaux avec la ZAM

Les appels terminaux (non externes) sont tous compilés de la même manière, en utilisant l'instruction **`APPTERM`** et ses variantes (**`APPTERM`**1, **`APPTERM`**2 et **`APPTERM`**3).

`APPTERM` est une instruction qui prend deux arguments N et S, où N est le nombre d'arguments se trouvant sur le haut de la pile, et S est la place occupée par la fonction courante sur la pile, ce qui permet de récupérer l'ensemble de la place occupée par la fonction courante (*i.e.*, on récupère tous les objets de la fonction courante sauf les arguments qui sont simplement déplacés du haut vers le bas, ce qui fait S-N niveaux de pile récupérés). Enfin, le pointeur de code est affecté avec le pointeur de code de la fermeture se trouvant dans le registre **`ACCU`**. Ainsi, l'instruction exécutée juste après est la première instruction de la fonction appelée.

L'instruction **`APPTERM`**1 appliquée à S est équivalente à l'instruction **`APPTERM`** appliquée à N et S si N = 1. Les deux autres instructions **`APPTERM`**2 et **`APPTERM`**3 sont définies suivant le même schéma.

On remarque que non seulement cette instruction n'ajoute rien sur la pile, mais en plus elle en collecte éventuellement un peu.

Les sauts directs avec la ZAM S'il n'y a pas d'instruction « *goto* » dans les instructions de la ZAM, il y a en revanche une instruction BRANCH. Elle permet de faire des sauts directs relatifs en additionnant un entier relatif au pointeur de code.

B.2.5 Les appels externes avec la ZAM

L'instruction CCALL et ses variantes permettent d'invoquer des fonctions externes, c'est-à-dire des fonctions dont le code n'est pas fait d'instructions de la ZAM. Pour la ZAM officielle, les appels externes sont des appels à des procédures en langage C spécifiquement écrites selon les conventions de l'interface des fonctions externes (aussi appelée FFI, pour *Foreign Function Interface*).

Avant l'appel, le premier argument se trouve dans le registre ACCU, les autres arguments éventuels se trouvent sur la pile d'exécution de la machine ZAM. L'invocation de la fonction externe utilisera les arguments qui lui sont passés et affectera son résultat au registre ACCU. Les éventuels arguments se trouvant sur la pile sont supprimés à la fin de l'évaluation de l'instruction CCALL (ou variante).

ANNEXE C

Notes d'implantation

C.1 Implantation de la génération des traces des conditions et des décisions par réécriture du code source

Le listing C.1 présente une implantation en mCaml de l'environnement d'exécution du programme pour l'émission de traces des conditions et décisions. À noter que pour un programmeur habitué à OCaml, le choix des constructions peut paraître un peu surprenant, mais autrement on a là un programme qui compile en OCaml et qui est conforme à la grammaire de notre langage mCaml.
Voici un tableau de correspondance entre les notations formelles et l'implantation du listing C.1.

notation formelle	code mCaml
newVect(n)	`new_vect n`
`update( ⟦ e ⟧ , c )`	`update <e.loc_start> <e.loc_end> ⟦ e ⟧ c`
`updateD( ⟦ e ⟧ , c , d )`	`update_decision <e.loc_start> <e.loc_end> ⟦ e ⟧ c d`
`updateCD ( ⟦ e ⟧ , c1 , c2, d )`	`update <e.loc_start> <e.loc_end> (update_decision <e.loc_start> <e.loc_end> ⟦ e ⟧ c1 d) c2`
$t_{[n]}$	`get_cell t n`
$t_{s,n}$	`subtrace s n t`
$\emptyset$	`{ start = 0; size = 0; cells = Empty }`

où <e.loc_start> est le numéro du premier caractère de l'expression e et <e.loc_end> est le numéro du dernier ; ces éléments sont remplacés par des entiers littéraux pendant la compilation.

Cette implantation proposée à titre d'exemple conceptuel utilise un environnement pour enregistrer l'ensemble des traces d'exécution. À la fin l'exécution, l'environnement pourra être sérialisé dans un fichier pour analyse ultérieure. La sérialisation pourra se faire en OCaml à l'aide du module « Marshal » [1] ou bien simplement en imprimant les données selon un autre format comme un format textuel ou XML.

Pour éviter de retenir l'ensemble des traces dans la mémoire à l'exécution, on pourra les compresser. Une manière est de stocker les traces dans des structures implantant des ensembles qui éliminent automatiquement les doublons, car si la quantité de traces différentes est bornée, la quantité de traces pouvant être générée n'est bornée que par le temps d'exécution du programme. Une autre manière est de sérialiser à la volée les traces pour les stocker dans un fichier au lieu de les stocker en mémoire. Et en combinant les deux approches, on peut stocker dans un fichier en supprimant les doublons.

On remarque aussi que certains vecteurs de conditions/décisions ne peuvent pas être complets. Pour eux, il suffit de les considérer comme fixés à la fin de l'évaluation de la décision. Cette instrumen-

1. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html

Listing C.1 – Gestion des vecteurs de traces des conditions et décisions en mCaml

```
type ('a, 'b) pair = { fst: 'a ; snd: 'b }
type 'a list = Empty | Cons of ('a , 'a list) pair
type cell_value = Unknown | True | False
type location = { loc_start: int ; loc_end: int }
type cell = { mutable location: location; mutable cell_value: cell_value }
type trace = { start: int ; size: int ; cells: cell list }
type environment = { mutable env_data: trace list }
let environment = { env_data = Empty }
let allocation data =
  environment.env_data <- Cons { fst = data; snd = environment.env_data }
let rec new_cells n =
  if n = 0 then
    Empty
  else
    Cons{fst = { location = { loc_start = 0 ; loc_end = 0 };
                 cell_value = Unknown };
         snd = new_cells(n - 1) }
let new_vect n =
  let t = { start = 0; size = n ; cells = new_cells n } in
    allocation t;
    t
let rec subcells start length cells =
  match cells with
  | Empty -> Empty
  | Cons{fst = hd; snd = tl} ->
     if start = 0 then
       Cons{fst = hd; snd = subcells 0 (length - 1) tl}
     else
       subcells (start - 1) length tl
let subtrace start size trace =
  if size = 0 then
    { start = 0; size = 0; cells = Empty }
  else
    { start = start; size = size; cells = subcells start size trace.cells }
let rec nth list n =
  match list with
  | Cons{fst=hd; snd=tl} ->
     if n = 0 then hd else nth tl (n - 1)
let get_cell trace n = nth trace.cells n
let update expr_start expr_end v cell =
  cell.cell_value <- v;
  cell.location <- { loc_start = expr_start ; loc_end = expr_end };
  v
let update_decision dec_start dec_end v cell trace =
  update dec_start dec_end v cell; (* flush trace; *)
  v
```

tation casse les appels terminaux qui rendent des booléens, mais de toutes façons les appels terminaux sont déjà rétrogradés par l'instrumentation pour la couverture structurelle des expressions.

Enfin, on pourra remarquer que ce code ne tient pas compte par exemple des potentiels dépassement des bornes des vecteurs de conditions. Mais comme il s'agit d'un code dont les fonctions ne sont destinées qu'à être invoquées par du code généré, il paraît pertinent de ne pas le surcharger inutilement avec des gestions d'erreurs qui ne peuvent pas être rencontrées.

C.2 Génération de tests pour MC/DC

La génération automatique des tests peut servir d'aide au développement. Voici un programme qui génère, à partir d'une expression booléenne composée par uniquement par les opérateurs séquentiels **&&** et **||**, un jeu de tests permettant de satisfaire le critère MC/DC, sous réserve que chaque test se termine bien (*i.e.*, sans lever d'exception).

```
type boolexpr =
    | Bool
    | And of boolexpr * boolexpr
    | Or of boolexpr * boolexpr
    | Not of boolexpr

type mcdc_mem = { res : bool ; test : bool list}

let rec gen_mcdc_table =
  let rec split_t = function
    | [] -> raise (Invalid_argument "split_t")
    | ({ res = true; test = _ } as hd)::tl -> hd, tl
    | hd::tl -> match split_t tl with l,r -> l, (hd::r)
  in
  let rec split_f = function
    | [] -> raise (Invalid_argument "split_t")
    | ({ res = false; test = _ } as hd)::tl -> hd, tl
    | hd::tl -> match split_f tl with l,r -> l, hd::r
  in
  function
  | Not boolexpr ->
     let l = (gen_mcdc_table boolexpr) in
       List.map (* "Not" only affects the result of the expression. *)
         (fun e -> {e with res= not e.res})
         l
  | Bool ->
      [
       { res = true; test = [true] };
       { res = false; test = [false] };
      ]
  | (And (e1, e2) | Or (e1, e2)) as eB ->
      let is_and =
        match eB with And _ -> true | _ -> false
      in
        (fun split -> (* split has to be a function which extracts
                         a particular element from the list.
                         let a, b = (split l) : a is a vector which
                          should lead to result true if operator is "And"
                          and false if operator is "Or". b is the rest
                          of list l. *)
           let ge1 = gen_mcdc_table e1 in
```

```
42            let ge2 = gen_mcdc_table e2 in
43            let { res = _ ; test = left_test }, r =
44              (* Extracts one test which returns true,
45                 returns it along with the rest.
46                 This means left_test is a test vector
47                 which expected result is true. *)
48                 split ge1
49            in
50            let hd_ge2 =
51                (* It's far better to take a full usefull test vector *)
52              fst (split ge2)
53            in
54              ((* For "And" (resp. "Or", left part is true (resp. false)),
55                  so right part is actually usefull.
56                  This implies that result of test is result of right part. *)
57                List.map
58                  (fun {res = x; test = test} ->
59                     {res = x; test = left_test @ test})
60                  ge2)
61                  @
62                  ((* All left tests that have not been treated must be kept. They
63                      are combined with exactly one arbitrary member of right part. *)
64                  List.map
65                    (fun {res = x; test = test} ->
66                       {res = x; test = test @ hd_ge2.test })
67                    r)
68          )
69         (if is_and then split_t else split_f)
70
71  let _ =
72    gen_mcdc_table (And(Or(And(Bool,Bool),Bool),Bool))
73
74
75
76  type iexpr =
77    | BOOL of bool
78    | AND of iexpr * iexpr
79    | OR of iexpr * iexpr
80    | NOT of iexpr
81    | JOKER
82
83  let rec eval = function
84      | AND (a, b) -> eval a && eval b
85      | OR (a, b) -> eval a || eval b
86      | NOT e -> not (eval e)
87      | BOOL b -> b
88      | JOKER -> assert false
89
90  let acceptable_tests_with_seqop expr =
91    let rec gen = function
92      | Not e ->
93          let et = gen e in
94            List.map (fun x -> NOT x) et
95      | Bool -> [BOOL false; BOOL true]
96      | And (a, b) ->
97          let at = gen a
98          and bt = gen b in
99            List.flatten
```

```
100             (List.map (fun be -> List.map (fun ae -> AND(ae,be)) at) bt)
101     | Or (a, b) ->
102         let at = gen a
103         and bt = gen b in
104           List.flatten
105             (List.map (fun be -> List.map (fun ae -> OR(ae,be)) at) bt)
106   and filter_one = function
107     | AND (a, b) ->
108         if eval a
109         then AND(filter_one a, filter_one b)
110         else AND(filter_one a, place_jokers b)
111     | OR (a, b) ->
112         if eval a
113         then OR(filter_one a, place_jokers b)
114         else OR(filter_one a, filter_one b)
115     | NOT a ->
116         NOT (filter_one a)
117     | BOOL b ->
118         BOOL b
119     | JOKER -> assert false
120   and place_jokers = function
121     | AND (a, b) ->
122         AND(place_jokers a, place_jokers b)
123     | OR (a, b) ->
124         OR(place_jokers a, place_jokers b)
125     | NOT a ->
126         NOT(place_jokers a)
127     | BOOL _ ->
128         JOKER
129     | JOKER ->
130         assert false
131   and filter l = List.map filter_one l
132   and unique = function (* on sorted lists *)
133     | [] -> []
134     | a::b::tl -> if a = b then unique(b::tl) else a::(unique (b::tl))
135     | [e] -> [e]
136   in unique (List.sort compare (filter (gen expr)))
137
138 let _ =
139   let l =
140     acceptable_tests_with_seqop (And(Or(And(Bool,Bool),Bool),Bool))
141   in
142     List.map (fun e -> e, eval e) l
143
144
145 let rec size =  function
146   | And(a,b) -> size a + size b
147   | Or(a,b) -> size a + size b
148   | Not e -> size e
149   | Bool -> 1
150
151 let _ =
152
153   let l =
154     [
155       Or(And(Or(Or(Or(Or(Or(Bool,Bool),Bool),Bool),Bool),Bool),Bool),Bool);
156       Or(And(Or(Bool,Bool),Bool),Bool);
157       Or(Bool,
```

```
158           Not(And(Bool,
159               Or(Bool,
160                  And(Bool,
161                      Or(Bool,
162                         Or(Bool,Bool)))))));
163     ]
164   in
165     List.map
166       (fun e ->
167        size e,
168        List.length (List.map (fun e -> e, eval e) (acceptable_tests_with_seqop e))
169     )
170       l
```