Magistère de Mathématiques Fondamentales & Appliquées
et d'Informatique de la Région Parisienne

École normale supérieure

Mémoire de Magistère

Vincent Simonet

Septembre 2001

Vincent Simonet
INRIA Rocquencourt (Projet Cristal) *et* École normale supérieure
Vincent.Simonet@inria.fr
http://cristal.inria.fr/~simonet/

Introduction

Ce mémoire de magistère reprend différents documents que j'ai rédigé depuis mon entrée à l'École normale supérieure, organisés comme suit.

Le rapport débute par une introduction générale rappelant mon cursus au MMFAI et présentant mon domaine de recherche actuel.

La deuxième partie est constituée de mon mémoire de DEA, réalisé sous la direction de François Pottier au projet *Cristal* (INRIA Rocquencourt). Ce rapport débute par une brève présentation générale du domaine de l'*analyse de flots d'information*. Il s'intéresse ensuite à la conception et à l'étude d'un système de types pour un langage ML doté de traits impératifs (exceptions et références).

Vient ensuite la version longue d'un article (en anglais) écrit avec François Pottier et soumis à la conférence *Principles Of Programming Languages* (Portland, Oregon, Janvier 2002). Cet article reprend et améliore les résultats obtenu lors de mon stage de DEA. La présentation est cependant largement différente.

Enfin, la quatrième partie inclut deux documents de travail (rédigés en anglais). Le premier s'intéresse à une analyse fine de flots d'information dans un langage avec sommes (ce qui permet ensuite de traiter le mécanisme d'exception par codage). Le second décrit un algorithme de résolution et de simplification des contraintes engendrées par les systèmes de types précédents.

En annexe, le rapport du stage de maîtrise que j'ai effectué au *Laboratoire d'Informatique Algorithmique : Fondements et Applications* (Université Paris VII) sous la direction de Paul Gastin.

On trouvera également en ligne :

La description et le code source d'un petit compilateur réalisé en première année. Il s'intéresse au langage ML avec *let* polymorphe, récursion et références.
`http://cristal.inria.fr/magistere/compil/`

Une brève présentation d'un micro-processeur réalisé avec Antoine Kremer accompagnée du code source en Jazz (un langage permettant de décrire des circuits asynchrones).
`http://cristal.inria.fr/magistere/micro/`

Première partie

Avant-propos

Résumé du cursus au MMFAI

1998–1999 Première année du MMFAI

Licence d'Informatique
Algorithmique et programmation, *Jacques Stern.*
Langages de programmation et compilation, *Patrick Cousot.*
Théorie de l'information, *Jean Vuillemin.*
Langages formels, calculabilité, complexité et analyse d'algorithmes, *Paul Gastin.*

Licence de Mathématiques
Algèbre 1, *Arnaud Beauville.*
Intégration et probabilités, *Francis Comets.*
Logique, *Alain Louveau.*

Maîtrise d'Informatique
Calcul Formel, *Daniel Lazard.*
Géométrie algorithmique, *Michel Pocchiola.*
Programmation Java, *François Bourdoncle et Patrice Bertin.*
Sémantique des langages de programmation, *Patrick Cousot.*

Stage
« Comparaison d'outils de vérification » sous la direction de Paul Gastin (LIAFA)

1999-2000 DEA Programmation : Sémantique, Preuves et Langages

Cours de tronc commun
Termes en logique du premier ordre, *Claude Marché et Jean-Pierre Jouannaud.*
Lambda-calcul, *Jean-Jacques Lévy.*
Preuves constructives, *Gilles Dowek.*
Sémantiques des programmes séquentiels et concurrents, *R. Di Cosmo et D. Sangiorgi.*
Typage et programmation, *Xavier Leroy.*

Cours d'options
Concurrence et Mobilité, *Georges Gonthier.*
Programmation par contraintes, *Philippe Codognet et François Fages.*
Sous-typage et langages à objets, *Giuseppe Castagna et François Pottier.*
Types, catégories et domaines, *Pierre-Louis Curien et Giuseppe Longo.*
Logique Linéaire, *Roberto Di Cosmo et Delia Kesner.*
Fondements de l'interprétation abstraite, *Patrick Cousot.*
Analyse statique par Interpretation abstraite, *R. Cousot, A. Deutsch et A. Venet.*
Modèles finis, *Serge Abiteboul et Stephane Grumbach.*

2000-2001 Stage long

« Inférence de flots d'information pour ML »
sous la direction de François Pottier (INRIA Rocquencourt, Projet Cristal)

Thème de recherche

Flots d'information

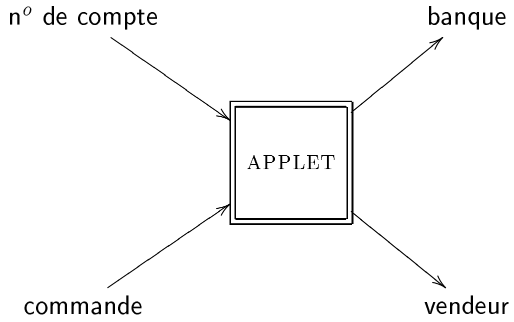
Supposons qu'un système informatique contienne des données confidentielles, pour des raisons militaires, commerciales ou personnelles. Comment en empêcher un mauvais usage ? Il nous faut les protéger non seulement contre l'ensemble des *individus* qui ont accès au système, mais également contre les *programmes* que nous-mêmes utilisons, puisque ceux-ci pourraient contenir — intentionnellement ou non — des fragments de code qui permettront une diffusion, partielle ou totale, de ces données.
On a donc cherché, depuis plusieurs décennies, des moyens de contrôler la dissémination d'information effectuée par un programme. La solution la plus simple, inacceptable, consiste à utiliser un système mono-utilisateur, isolé du monde extérieur. Une seconde solution, couramment répandue aujourd'hui, consiste à utiliser un système de *droits d'accès*, qui demande aux principaux (individus ou programmes) de s'identifier à l'aide de mots de passe. Cependant, une fois un tel droit d'accès accordé, le détenteur des données n'a plus aucun contrôle sur leur dissémination ; il doit donc avoir *confiance* en la correction (et en les bonnes intentions) du programme auquel il accorde cette permission. Ce système est donc dangereux, car cette confiance peut être accordée à tort, et restrictif, car on peut parfois souhaiter confier des données à une entité en qui on n'a pas confiance *a priori* [Lam73] — un scénario très en vogue à l'heure du commerce électronique.

Une solution prometteuse consiste à analyser préalablement le programme auquel on souhaite conférer certains droits, de façon à déterminer quel usage il en fera. Si l'analyse statique garantit que le programme se comportera de façon acceptable, alors celui-ci peut être exécuté sans plus aucune restriction ; sinon, son exécution est refusée. Au lieu de devoir faire toute confiance à l'auteur du programme, il nous suffit donc d'avoir confiance en la correction de l'analyse, laquelle peut être prouvée formellement. Ce procédé est généralement nommé analyse de *flots d'information*.
Les premiers pas dans cette direction datent de la fin des années 1970 [DD77, AR80]. Bien qu'intéressants, ces travaux ne proposent encore que des techniques d'analyse manuelles et dont la correction n'est pas prouvée. Durant les quinze ans qui suivent, on assiste à quelques progrès, mais il faut attendre le milieu des années 1990 pour voir cette analyse formulée en termes de *typage*.

Analyse par typage

Depuis quelques années, on sait réaliser une analyse pouvablement correcte pour des programmes exprimés dans des langages impératifs [VS97] ou fonctionnels [ØP97, HR98, ABHR99, PC00] simples, à l'aide de systèmes de types annotés par des *niveaux d'information*.

Ce type d'analyse consiste généralement à vérifier qu'un programme satisfait certaines propriétés de confidentialité ou d'intégrité vis-à-vis des données qu'il manipule, en étudiant les dépendances entre ses entrées et ses sorties : par exemple, si une entrée est considérée secrète (ou non fiable), on généralement que des sorties publiques n'en dépendent pas, de façon à pouvoir les publier (ou les considérer comme fiables). Contrairement à de nombreuses propriétés de sécurité, l'absence de dépendance n'est pas une propriété de sûreté. Elle est habituellement énoncée en termes de *non-interférence* : on se donne deux exécutions du programme à partir de valeurs d'entrée distinctes, et on compare les valeurs de sortie. Si celles-ci sont distinctes, alors il y a dépendance.

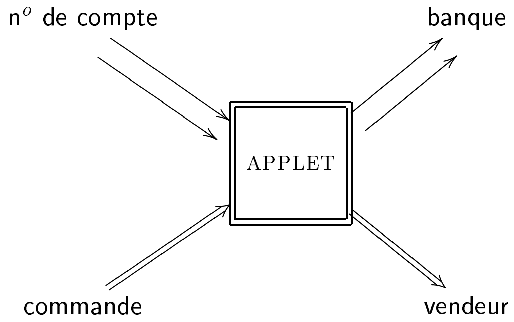


$$\text{compte}^H \times \text{commande}^L \to \text{banque}^H \times \text{vendeur}^L$$

$$(\forall\alpha\beta\gamma\delta)\ [\alpha \sqcup \beta \leq \gamma, \beta \leq \delta]\ \text{compte}^\alpha \times \text{commande}^\beta \to \text{banque}^\gamma \times \text{vendeur}^\delta$$

Cette figure décrit un cas simple de flots d'information. Une application de commerce électronique recueille des données auprès d'un utilisateur : une commande accompagnée de coordonnées bancaires. Elle le transmet ensuite d'une part à la banque (pour qu'elle enregistre le paiement) et d'autre part au vendeur (pour qu'il traite la commande). On souhaite alors typiquement s'assurer que le vendeur n'a pas accès à la donnée secrète : le numéro de compte du client. Le type brut $\text{compte}^H \times \text{commande}^L \to \text{banque}^H \times \text{vendeur}^L$ *traduit cette spécification : le numéro de compte est marqué H (secret) alors que les données transmises au vendeur sont marquées L (public). En fait, on préférera donner à une telle application un schéma contraint polymorphe tel que* $(\forall\alpha\beta\gamma\delta)\ [\alpha \sqcup \beta \leq \gamma, \beta \leq \delta]\ \text{compte}^\alpha \times \text{commande}^\beta \to \text{banque}^\gamma \times \text{vendeur}^\delta$ *et dont le type brut précédent est une instance (on a* $L \leq H$*). Les niveaux associés à chaque entrée et sortie sont variables. Les flots d'information sont traduits sous forme de contraintes d'ordre entre celles-ci : la banque a accès aussi bien au numéro de compte qu'à la commande elle-même (*$\alpha \sqcup \beta \leq \gamma$*) alors que le vendeur est limité à cette dernière (*$\beta \leq \delta$*).*

Fig. 1: *Exemple de flots d'information*



Ce schéma illustre la notion de non-interférence : on veut s'assurer que le vendeur ne peut pas apprendre le numéro de compte du client. Pour cela, on s'intéresse à plusieurs exécutions du programme pour différentes valeurs du numéro de banque. Si les données transmises au vendeur sont invariantes alors on est assuré que le numéro de banque est bien gardé secret par l'applet.

Fig. 2: *Non-interférence*

Au cours de mon stage de DEA, je me suis intéressé à la conception et à l'étude d'un tel système généralisant les résultats précédents à l'ensemble de ML, un langage fonctionnel doté de traits impératifs (références), d'un mécanisme d'exceptions et de « primitives » (opérations arithmétiques, comparaisons polymorphes, hachage, marshalling...). Sa correction, exprimée par un énoncé de *non-interférence*, est ramenée à la preuve d'une simple propriété de *subject-reduction* pour un langage non standard dans lequel on peut raisonner sur les points communs entre deux exécutions distinctes d'un programme.

Ce système de types « à la HM(X) » [SMZ99], présente trois caractéristiques notables : il possède *polymorphisme*, *sous-typage* et permet l'*inférence* des types.
L'introduction de sous-typage est naturelle dans la mesure où les niveaux d'information sont représentés par un treillis. Elle permet de considérer une expression ayant un niveau ℓ comme ayant n'importe quel niveau ℓ' plus grand que ℓ. Cela revient par exemple à autoriser l'utilisation d'une donnée « publique » là où une expression « secrète » est attendue.
Le polymorphisme intervient de manière orthogonale. Il apparaît particulièrement essentiel dans une analyse de flots : on souhaite en effet pouvoir typer une fonction indépendamment des niveaux effectifs des arguments qui pourront varier suivant les différentes applications. La propriété de sécurité est alors exprimée sous forme de contraintes d'ordre entre des niveaux d'information variables.
Enfin, l'existence d'un algorithme d'inférence efficace permettant de calculer les types et de vérifier la validité d'un programme automatiquement est un avantage majeur pour un tel système. On peut supposer que dans un projet réel, l'analyse de sécurité n'intervienne que dans un second temps. L'inférence permet de séparer en grande partie cette vérification du reste du processus de vérification dans la mesure où elle ne nécessite pas de modifier l'ensemble du code. Les annotations de sécurité devraient pouvoir se limiter à quelques spécifications données sous forme d'interfaces.

Projets

Algorithme d'inférence L'implantation d'un système d'inférence pour un système de types avec contraintes de sous-typage tel que le système précédent nécessite principalement la conception d'algorithmes permettant d'une part de *résoudre* des ensembles de contraintes (c'est-à-dire de tester la satisfiabilité d'un ensemble de contraintes) et d'autre part de les *simplifier* (en les réécrivant sous une forme équivalente plus « simple »). La simplification est nécessaire aussi bien pour être en mesure d'afficher des résultats lisibles par l'utilisateur que pour des raisons d'efficacité.
La résolution des contraintes en présence de simples contraintes de sous-typage atomique est décidable et bien comprise [Reh97]. Cependant, notre système fait appel à des contraintes particulières (gardes), présentant quelques similitudes avec des contraintes conditionnelles. Notre approche consiste à partir des techniques de résolution présentées par François Pottier dans sa thèse [Pot98] pour les adapter au cas particulier du sous-typage atomique puis les étendre à nos nouvelles formes de contraintes. J'ai réalisé une première implantation d'une bibliothèque générique de résolution et de simplification de contraintes dans le cadre du sous-typage atomique en *Objective Caml* et m'intéresse actuellement à la formalisation et à la preuve des algorithmes mis en œuvre.
Enfin, la comparaison de schémas de types clos (nécessaire pour vérifier qu'une implantation est conforme à son interface) reste encore une question ouverte.

Intégration au compilateur Caml Un des premiers objectifs de mon travail de thèse est d'implanter cette analyse de sécurité en tant qu'extension du compilateur *Caml*, dans le but de réaliser un outil réaliste. Outre d'éventuelles questions directement relatives à l'analyse de flots d'information, il faudra résoudre plusieurs problèmes liés à l'emploi d'un système d'inférence de types doté de sous-typage : efficacité de l'algorithme de résolution des contraintes, lisibilité des types inférés, clarté des diagnostics d'erreur, légèreté de la notation fournie à l'utilisateur, etc.
Une fois cette première implantation réalisée, une étude de cas pourra être réalisée de façon à illustrer l'expressivité du système, déterminer pour quelle gamme d'applications elle est suffisante et, le cas échéant, quelles améliorations théoriques seraient souhaitables. Il sera particulièrement intéressant de déterminer dans quelle mesure des programmes existants, écrits sans souci particulier de sécurité, devront être modifiés pour être acceptés par l'analyseur. Notre but sera bien sûr de rendre cette tâche aussi simple que possible.

Extensions et améliorations Enfin, plusieurs extensions et améliorations du système de type proposé peuvent d'ores et déjà être envisagées. D'autres devraient être suggérées par l'utilisation effective de l'analyseur.

Le traitement des types sommes est particulièrement intéressant, non seulement parce qu'il permet des analyses fines mais aussi car il peut servir d'intermédiaire pour le typage des exceptions. Une première proposition est décrite dans la quatrième partie de ce rapport. Elle fait appel à des contraintes d'une forme originale impliquant des rangées bidimensionnelles.

On peut également souhaiter ajouter de nouvelles fonctionnalités au langage étudié, telles que les *threads* ou des primitives cryptographiques, de déclassification, etc.

Références

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999. `http://pa.bell-labs.com/~abadi/Papers/flowpopl.ps`.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1) :56–76, January 1980.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7) :504–513, July 1977.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus : Programming with secrecy and integrity. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998. `http://cm.bell-labs.com/cm/cs/who/nch/slam.ps`.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the Association for Computing Machinery*, 16(10) :613–615, October 1973. `http://research.microsoft.com/lampson/11-Confinement/WebPage.html`.

[ØP97] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6) :557–591, November 1997. `http://www.cs.purdue.edu/homes/palsberg/paper/jfp97.ps.gz`.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, Montréal, Canada, September 2000. `http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz`.

[Pot98] François Pottier. *Synthèse de types en présence de sous-typage : de la théorie à la pratique.* PhD thesis, Université Paris 7, July 1998. `http://pauillac.inria.fr/~fpottier/publis/these-fpottier.ps.gz`.

[Reh97] Jakob Rehof. Minimal typings in atomic subtyping. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 278–291, Paris, France, January 1997. `http://research.microsoft.com/~rehof/popl97.ps`.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. `http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz`.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214 :607–621, April 1997. `http://www.cs.nps.navy.mil/people/faculty/volpano/papers/tapsoft97.ps.Z`.

Deuxième partie

Inférence de flots d'information pour ML

(Mémoire de DEA)

Résumé

L'analyse de flots d'information consiste à déterminer, à l'aide d'un système de types annotés par des niveaux d'information, si un programme satisfait certaines propriétés de confidentialité ou d'intégrité vis à vis des données. Ce mémoire s'intéresse à la conception et à l'étude d'un tel système pour un langage de type ML doté de traits impératifs : références et exceptions. Le système proposé comporte sous-typage et polymorphisme. Sa correction est exprimée par la preuve d'une propriété de non-interférence.

Dans la première partie du rapport, on présente le domaine de l'analyse de flots d'information et en particulier les principaux problèmes soulevés par un langage aux traits impératifs. La seconde partie constitue le cœur de l'ouvrage : on donne un système de types pour un langage ML doté de références et d'exceptions puis on prouve une propriété de non-interférence. Enfin, dans la troisième partie, on s'intéresse aux problèmes soulevés par l'inférence de types.

Table des matières

Chapitre **1**

Flots d'information

1 Flots d'information et analyse de sécurité

1.1 Présentation générale

La gestion d'une grande quantité de données sur des systèmes d'information nécessite d'effectuer un contrôle sur les accès à ces données. On peut en effet souhaiter garantir deux propriétés : d'une part le *secret*, c'est-à-dire l'impossibilité pour certains utilisateurs de prendre connaissance (de manière directe ou indirecte) de certaines informations ; et d'autre part l'*intégrité*, c'est-à-dire l'impossibilité pour certains agents de modifier ou corrompre certaines données.
Une solution usuelle consiste en la mise en place d'un système de droits d'accès, comme dans certains systèmes d'exploitation : à chaque fichier est associée une annotation indiquant quels utilisateurs peuvent lire ou écrire son contenu. Ce contrôle peut s'avérer insuffisant : les données doivent être protégées contre les divulgations et corruptions (volontaires ou accidentelles) causées non seulement par les utilisateurs, mais aussi par les programmes eux-mêmes. Les systèmes basés sur la notion de droits nécessitent une confiance inconditionnelle en les programmes qui ont accès aux données sensibles.
Autoriser l'accès aux données à des programmes auxquels on ne peut accorder une telle confiance nécessite d'analyser leur code pour s'assurer qu'ils respectent quelques règles de sécurité. Ce processus est appelé *analyse de flots d'information*. Il peut faire appel à des techniques diverses : vérifications effectuées au moment de l'exécution, lors de la compilation ou même faisant intervenir une preuve manuelle.

Niveaux d'information et non-interférence Avant de proposer une telle analyse, il nous faut énoncer la propriété que l'on souhaite vérifier sur les programmes étudiés. Cette propriété doit être suffisamment générale pour s'adapter à une large variété de situations et de politiques de sécurité.
Nous devons d'abord formaliser la notion de *niveaux d'information*. Suivant le formalisme de [BL75], nous représentons ces niveaux par les éléments d'un treillis $\mathcal{L}$. Les niveaux sont ordonnés des moins secrets vers les plus secrets. Cette formalisation ouvre un large éventail de possibilités. Par exemple, le treillis peut être l'ensemble des parties d'un ensemble dont les éléments représentent des utilisateurs. Chaque information peut alors être annotée par la liste des personnes ayant ou non le droit d'en prendre connaissance.
La propriété que nous allons établir pour les programmes étudiés s'appelle la *non-interférence*. Informellement, cette propriété exprime le fait que tout ou partie des sorties d'un programme ne dépend d'aucune manière de tout ou partie de ses entrées.

Typage, sous-typage et polymorphisme L'utilisation de systèmes de typage pour établir la non-interférence d'un programme a été plusieurs fois proposée. Il s'agit d'une approche naturelle et intéressante : le contrôle des flots d'information peut être totalement effectué avant toute exécution du programme et il n'a aucune répercussion en termes de temps ou d'espace mémoire utilisé lors de l'exécution. Cette idée est confortée par le fait que les langages étudiés disposent en général déjà d'un système de types. L'analyse des flots d'information peut alors souvent se matérialiser par l'ajout d'annotations aux types existants.

Les systèmes de types que nous étudierons auront deux caractéristiques notables : *polymorphisme* et *sous-typage*. L'introduction de sous-typage est naturelle dans la mesure où les niveaux d'information sont représentés par un treillis. Elle nous permettra de considérer une expression ayant un niveau ℓ comme ayant n'importe quel niveau ℓ' plus grand que ℓ. Cela revient par exemple à autoriser l'utilisation d'une expression « publique » là où une expression « secrète » est attendue.
Le polymorphisme intervient de manière orthogonale. Il apparaît particulièrement important dans une analyse de flots : on souhaite en effet pouvoir typer les fonctions indépendamment des niveaux effectifs des arguments qui pourront varier suivant les différentes applications. La propriété de sécurité est alors exprimée sous forme de contraintes d'ordre entre des niveaux d'information variables.

Inférence L'existence d'un algorithme d'inférence efficace est un point important pour un tel système. On peut supposer que dans un projet réel, l'analyse de sécurité n'intervienne que dans un second temps. L'inférence permet de séparer en grande partie cette vérification du reste du processus de vérification dans la mesure où elle ne nécessite pas de modifier l'ensemble du code. Les annotations de sécurité devraient pouvoir se limiter à quelques spécifications données sous forme d'interfaces.

1.2 Travaux précédents

Les premiers travaux s'intéressant à des analyses de flots d'information datent de la fin des années 1970 [DD77, AR80]. Bien qu'intéressants, ils ne proposent cependant que des techniques d'analyse manuelles et dont la correction n'est pas prouvée. Durant les quinze ans qui suivent, on assiste à quelques progrès, mais il faut attendre le milieu des années 1990 pour voir cette analyse formulée en termes de *typage*, que l'on s'intéresse à des langages de programmation impératifs [VS97] ou fonctionnels [ØP97, HR98, ABHR99].
Nous présentons brièvement quelques travaux récents relatifs à des analyses de flots d'information sur des langages présentant des traits fonctionnels ou impératifs. Ces travaux restent essentiellement théoriques dans la mesure où aucune implantation d'échelle raisonnable n'a encore à ce jour été proposée.

Trust in the λ-calculus *(Peter Ørbæk et Jens Palsberg, 1997)* [ØP97] Peter Ørbæk et Jens Palsberg présentent un λ-calcul doté d'opérations explicites permettant de marquer une expression comme étant « de confiance » ou non. Ils proposent ensuite un système de types pour ce calcul qui possède une propriété de *subject reduction*. Ils incluent une primitive « *declassify* » et ne peuvent donc pas prouver de théorème de non-interférence.

A Type-Based Approach to Program Security *(Dennis Volpano et Geoffrey Smith, 1997)* [VS97] Cette étude s'intéresse à un petit langage procédural du premier ordre, avec des variables mutables. Elle présente un système de types qui garantit qu'un programme bien typé satisfait une propriété de non-interférence. Il est accompagné d'un algorithme d'inférence qui produit des types principaux.
La correction du système est prouvée. Cependant le langage reste relativement limité : les procédures ne sont pas des objets de première classe et on est *de fait* limité à un nombre fini de variables.

The SLam Calculus *(Nevin Heintze et Jon G. Riecke, 1998)* [HR98] Le *SLam Calculus* est un λ-calcul typé. En plus de donner des informations habituelles de types, les types du SLam fournissent des informations de sécurité. La première partie de l'article est consacrée à un noyau fonctionnel pour lequel les auteurs citent et prouvent un théorème de non-interférence. Cependant, les types ne sont pas inférés mais doivent être écrits par le programmeur, le système se contentant de vérifier la cohérence de ces annotations.
La seconde partie de l'étude concerne une extension contenant à la fois des références et un calcul de processus. Il n'est cependant pas énoncé de propriété précise de sécurité pour ce deuxième système. De plus, la présence de processus empêche d'effectuer un typage fin des références car elle exige un résultat de non-interférence fort prenant en compte la notion de terminaison.

JFlow *(Andrew C. Myers, 1999)* [Mye99a, Mye99b] Dans sa thèse, Andrew C. Myers a proposé une analyse de flots d'information pour le langage Java. Il s'agit d'un système relativement complet, effectuant principalement une analyse statique par typage, mais possédant également des traits dynamiques.
Les déclarations de variables sont annotées en JFlow par leur niveau de sécurité, ce qui permet au typeur de vérifier que les programmes ne comportent pas de fuite d'information. Cependant, la correction du système proposé n'est pas prouvée et l'implantation n'a pas été publiée à ce jour.

Valeurs

$$
\begin{array}{rll}
v ::= & k & \text{(constantes entières)} \\
 & \mid () & \text{(constante unit)} \\
 & \mid x & \text{(variables)} \\
 & \mid m & \text{(adresses mémoire)} \\
 & \mid \lambda x.e & \text{(abstraction)} \\
 & \mid \mathsf{exn}_\varepsilon\ v & \text{(construction d'exception)}
\end{array}
$$

Expressions

$$
\begin{array}{rll}
e ::= & a & \\
 & \mid E[e] & \\
 & \mid \mathsf{let}\ x = v\ \mathsf{in}\ e & \text{(let polymorphe)} \\
 & \mid v\ v & \text{(application)} \\
 & \mid \mathsf{ref}\ v & \text{(création de référence)} \\
 & \mid v := v & \text{(écriture dans une référence)} \\
 & \mid !\ v & \text{(lecture d'une référence)} \\
 & \mid \mathsf{raise}\ v & \text{(levée d'exception)}
\end{array}
$$

Résultats

$$
\begin{array}{rll}
a ::= & v & \text{(valeur)} \\
 & \mid \mathsf{raised}_\varepsilon\ v & \text{(exception levée)}
\end{array}
$$

Contextes

$$
\begin{array}{rll}
E ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e & \text{(bind)} \\
 & \mid \mathsf{try}_\varepsilon\ [\,]\ \mathsf{with}\ x \succ e & \text{(try)} \\
 & \mid \mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e & \text{(tryall)}
\end{array}
$$

Fig. 1: Syntaxe du langage $\mathrm{ML}_{\mathrm{re}}$

Information Flow Inference For Free *(François Pottier et Sylvain Conchon, 2000)* [PC00] François Pottier et Sylvain Conchon se sont intéressés au λ-calcul étiqueté d'Abadi, Lampson et Lévy [ALL96] étendu avec un « let ». Il s'agit donc d'un noyau ML purement fonctionnel dans lequel les étiquettes représentent les niveaux d'information des sous-expressions.
Leur approche consiste à traduire ce langage étiqueté dans un langage « standard » sans étiquettes, en associant à chaque expression étiquetée un couple dont la première composante représente l'expression elle-même et la seconde composante son étiquette. Grâce à cette traduction, il est possible d'obtenir de manière quasi-immédiate un système de types analysant les flots d'information pour le calcul étiqueté, à partir de tout système de types connu sur le calcul non-étiqueté vérifiant un nombre d'axiomes minimal.

Secure Information Flow and CPS *(Steve Zdancewic et Andrew C. Myers, 2001)* [ZM01] Ce dernier article, qui sera présenté à la conférence ESOP en avril prochain, propose une analyse sur un langage doté de références et de continuations linéaires. Ce langage peut être utilisé comme cible pour d'un compilateur.

2 Flots d'information et programmation impérative

2.1 ML avec références et exceptions

Nous introduisons dans cette section le langage que nous étudions. Il s'agit d'un noyau de ML étendu avec des références et des exceptions. Ce langage est défini par la grammaire de la figure 1. Nous l'appelons $\mathrm{ML}_{\mathrm{re}}$ par la suite.
On dispose d'un ensemble $\mathcal{E}$ de noms d'exceptions ε (il peut être fini ou infini). La construction try_ε permet de rattraper une exception dont le nom est ε alors que tryall rattrape toutes les exceptions.
On ne rappelle pas les règles de réduction qui sont classiques (et dont un sur-ensemble sera présenté en détail dans la partie 2). Nous utilisons le formalisme usuel avec état mémoire, dans le cas de l'appel par valeur qui est naturel dans un langage aux traits impératifs. La seule particularité réside dans la distinction entre déclenchement d'exception (raise) et exception déclenchée ($\mathsf{raised}_\varepsilon$). On a ainsi la règle de réduction

$$\mathsf{raise}\ (\mathsf{exn}_\varepsilon\ v)\ /\ \sigma \rightarrow \mathsf{raised}_\varepsilon\ v\ /\ \sigma \tag{raise}$$

alors que souvent les deux expressions $\mathsf{raise}\ (\mathsf{exn}_\varepsilon\ v)$ et $\mathsf{raised}_\varepsilon\ v$ sont confondues.

Notre syntaxe est inspirée des *formes A-normales* de [FSDF93] : nous imposons dans de nombreuses constructions la présence de valeurs (là où on autorise généralement n'importe quelle expression). Ce choix permet

$$
\begin{array}{llllllll}
(\mathbf{1}) & \begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\\ r := H : 1;\\ !\,r\end{array} &
(\mathbf{2}) & \begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\\ \mathsf{let}\ f = H : (\lambda xy.y)\ \mathsf{in}\\ r := f\,1\,2;\\ !\,r\end{array} &
(\mathbf{3}) & \begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\\ \mathsf{let}\ f = \lambda xy.y\ \mathsf{in}\\ r := f\,(H : 1)\,2;\\ !\,r\end{array} &
(\mathbf{4}) & \begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\\ H : (r := 1);\\ !\,r\end{array}
\end{array}
$$

$$
\begin{array}{llll}
(\mathbf{5}) & \begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\\ \mathsf{let}\ f = \lambda x.(r := x)\ \mathsf{in}\\ \mathsf{let}\ g = \lambda x.()\ \mathsf{in}\\ (H : f)\,1;\\ !\,r\end{array} &
(\mathbf{6}) & \mathsf{tryall}\ H : (\mathsf{raise}\ \mathsf{exn}_\varepsilon\ 0); 0\ \mathsf{with}\ x \succ 1
\end{array}
$$

$$
(\mathbf{7}) \quad \begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\\ \mathsf{tryall}\ H : (\mathsf{raise}\ \mathsf{exn}_\varepsilon\ 0); r := 1\ \mathsf{with}\ x \succ ();\\ !\,r\end{array}
$$

Fig. 2: Exemples d'expressions étiquetées

de bien distinguer d'une part les parties du code qui effectuent une opération (application de fonction, lecture/écriture de l'état mémoire, levée d'exception) et engendrent éventuellement des effets de bord et d'autre part les contextes (bind, try et tryall) qui permettent d'enchaîner les opérations.
Ces contraintes syntaxiques ne limitent pas le pouvoir d'expression du langage. Il est facile de mettre toute expression « usuelle » sous notre forme à condition de fixer un ordre d'évaluation. Ainsi une application $e_1\,e_2$ peut s'écrire $\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ (\mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ x_1\,x_2)$ si on évalue de la gauche vers la droite et $\mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ (\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ x_1\,x_2)$ si on évalue dans l'autre sens.
Notons également la distinction entre les constructions let et bind. Dans notre esprit, bind est un contexte qui permet d'enchaîner deux calculs en liant à un identificateur la valeur produite lors de la réduction de la première expression afin de pouvoir l'utiliser dans la réduction de la seconde expression. À l'inverse, le let est limité aux valeurs. Il ne sert que de « marqueur » pour la généralisation dans le typage. Nous suivons Wright [Wri95], en la limitant aux valeurs afin de ne pas entrer en conflit avec les traits impératifs du langage.

Un contexte peut se comporter de deux manières différentes vis-à-vis d'une réponse. On définit une relation sur les couples contexte – réponse : E rattrape a (notée $E \lll a$) par :

$$
\begin{array}{rlrl}
\mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \lll v & & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \not\lll \mathsf{raised}_\varepsilon\ v & \\
\mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e \not\lll v & & \mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e \lll \mathsf{raised}_\varepsilon\ v & \\
\mathsf{try}_{\varepsilon'}\ [\,]\ \mathsf{with}\ x \succ e \not\lll v & & \mathsf{try}_{\varepsilon'}\ [\,]\ \mathsf{with}\ x \succ e \lll \mathsf{raised}_\varepsilon\ v & \text{ssi } \varepsilon = \varepsilon'
\end{array}
$$

Langage étiqueté Nous représentons les niveaux d'information par les éléments ℓ (ou pc) d'un treillis $\mathcal{L}$. Afin de pouvoir marquer une partie d'une expression comme étant d'un niveau donné, il nous faut ajouter une construction « étiquette » au langage $\mathrm{ML}_{\mathrm{re}}$:

$$
\begin{array}{lll}
a & ::= \ldots \mid \ell : a & \text{(réponses)}\\
v & ::= \ldots \mid \ell : v & \text{(valeurs)}\\
e & ::= \ldots \mid \ell : e & \text{(expressions)}
\end{array}
$$

Nous ne donnons pas de règles de réduction pour ce calcul étiqueté, les étiquettes n'étant pour nous que des annotations de sécurité. Étant donnée une expression étiquetée e, on s'intéresse en revanche à la réduction de l'expression de $\mathrm{ML}_{\mathrm{re}}$, notée $\mathsf{strip}\,e$, obtenue à partir de e en « gommant » les étiquettes.

2.2 Flots directs et flots indirects

L'analyse de flots d'information dans un langage possédant des traits impératifs présente un certain nombre de difficultés particulières. Dans nos exemples, nous considérons que le treillis $\mathcal{L}$ ne comporte que deux niveaux : L pour les données publiques et H pour les données secrètes. On a bien entendu $L \leq H$.

On distingue généralement deux formes de flots d'information : les flots directs et les flots indirects. Les flots directs sont les plus immédiats. Considérons l'exemple (1) de la figure 2. L'étiquette H marque l'entier 1 comme secret. Cet entier est stocké dans une référence r dont le contenu est ensuite lu. Le résultat obtenu doit donc être considéré secret pour garantir l'absence de fuite d'information. De même, dans l'exemple (2), on applique une fonction secrète f aux arguments 1 et 2. Le résultat de cette application doit lui-même être considéré secret puisqu'il divulgue des informations sur f. À l'inverse, pour l'exemple (3), on passe un argument secret à la fonction f, mais cet argument n'est pas utilisé par f. On souhaite pouvoir considérer dans ce cas le résultat de l'application comme public.
Les flots indirects sont liés à l'idée qu'on peut apprendre une information sur une donnée secrète juste en apprenant qu'une expression donnée effectuant des effets de bord a été réduite. Dans l'exemple (4), on écrit dans la référence r sous un contexte secret : la sous-expression $r := 1$ toute entière est secrète. Le contenu de r doit donc lui-même être considéré comme secret. De même, pour l'exemple (5), on applique la fonction f en un point secret du programme : la lecture du contenu de r permet par exemple de savoir que c'est la fonction f et non g qui a été appelée. Il doit donc être considéré secret.

Ces exemples nous amènent à associer un niveau d'information pc au compteur de programme. Cette étiquette, définie à chaque point du programme – c'est-à-dire pour chaque contexte d'évaluation – donne le niveau d'information que l'on est susceptible de révéler en apprenant que ce ce point du programme a été exécuté. Intuitivement, ce niveau est égal à l'union des étiquettes que l'on est amené à traverser pour atteindre de l'extérieur le point considéré.

Considérons enfin quelques exemples relatifs aux exceptions. Dans (6), une exception est déclenchée en un point où le pc est égal à H. Le résultat de l'expression toute entière doit donc être considéré secret puisqu'il est susceptible de révéler que cette exception a été déclenchée. L'exemple (7) montre un cas où la lecture du contenu d'une référence permet indirectement d'apprendre qu'une expression a ou non levé une exception.

Chapitre **2**

Système SB(S) et non-interférence

Dans cette partie, nous donnons un système de typage pour $\mathrm{ML}_{\mathrm{re}}$ avec étiquettes et prouvons une propriété de non-interférence. Pour cela, nous introduisons un *calcul partagé* qui permet de formaliser simplement des propriétés de simulation entre les réductions de deux expressions qui ont en commun une partie de leur arbre syntaxique. Le calcul partagé n'est pas une fin en soi mais sera principalement un *outil* pour nous permettre d'obtenir aisément une propriété de non interférence sur le langage qui nous intéresse, $\mathrm{ML}_{\mathrm{re}}$. Dans la dernière section, nous présentons quelques extensions intéressantes de $\mathrm{ML}_{\mathrm{re}}$ et montrons comment notre système peut alors facilement être adapté.

3 Calcul partagé

3.1 Syntaxe et réduction

Syntaxe Le calcul partagé permet de coder au sein d'un même arbre syntaxique deux expressions ayant des parties communes en prenant ce partage en compte. On obtient le calcul partagé à partir de $\mathrm{ML}_{\mathrm{re}}$ en ajoutant une construction $\langle \ldots \mid \ldots \rangle$ aux classes syntaxiques des réponses, des valeurs et des expressions :

$$
\begin{aligned}
a &::= \ldots \mid \langle a \mid a \rangle && \text{(résultats)}\\
v &::= \ldots \mid \varnothing \mid \langle v \mid v \rangle && \text{(valeurs)}\\
e &::= \ldots \mid \langle e \mid e \rangle && \text{(expressions)}
\end{aligned}
$$

On a également ajouté, pour des raisons techniques, une constante $\varnothing$. Cette constante n'apparaîtra par la suite que dans les états mémoire, pour pouvoir représenter une adresse mémoire qui n'est définie que d'un côté. On ne considère par la suite que des résultats, valeurs et expressions dans lesquels les constructions $\langle \ldots \mid \ldots \rangle$ ne sont pas imbriquées. Une expression est *simple* si elle ne comporte pas de construction $\langle \ldots \mid \ldots \rangle$. Les expressions simples du calcul partagé sont exactement les expressions de $\mathrm{ML}_{\mathrm{re}}$.

Définition 3.1 (Projections) *On associe à chaque expression e du calcul partagé trois expressions simples :*
- $\lfloor e \rfloor_1$ *la « projection gauche » de e obtenue à partir de e en ne conservant que les parties gauches des sous-expressions $\langle \ldots \mid \ldots \rangle$.*
- $\lfloor e \rfloor_2$ *la « projection droite » de e obtenue à partir de e en ne conservant que les parties droites des sous-expressions $\langle \ldots \mid \ldots \rangle$.*

Ces deux constructions sont en fait des homomorphismes sur les constructions du langage, mis à part pour la construction à crochets :

$$\lfloor \langle e_1 \mid e_2 \rangle \rfloor_1 = e_1 \qquad\qquad \lfloor \langle e_1 \mid e_2 \rangle \rfloor_2 = e_2$$

On note également $\lfloor e \rfloor_\varnothing = e$ et on étend les projections aux états mémoire : $\lfloor \sigma \rfloor_i = \{m \mapsto \lfloor \sigma(m) \rfloor_i \mid m \in \mathrm{dom}(\sigma)$ et $\lfloor \sigma(m) \rfloor_i \neq \varnothing\}$.

Les projections permettent de retrouver les deux expressions simples qu'une expression du calcul partagé représente. C'est l'analogue de l'opération strip du calcul étiqueté.
Considérons par exemple les deux expressions étiquetées suivantes :

$$\begin{aligned} e_1 &= \mathsf{let}\ a = H : 1\ \mathsf{in}\ (\lambda x.x)\, a \\ e_2 &= \mathsf{let}\ a = H : 2\ \mathsf{in}\ (\lambda x.x)\, a \end{aligned}$$

Ces deux expressions ont un arbre syntaxique commun, jusqu'au niveau de l'étiquette H. On peut ainsi former une expression e du calcul partagé qui reflète cette similarité : $e = \mathsf{let}\ a = \langle 1 \mid 2 \rangle\ \mathsf{in}\ (\lambda x.x)\, a$. Grâce aux projections, on retrouve les expressions de départ : $\llbracket e \rrbracket_1 = (\mathsf{let}\ a = 1\ \mathsf{in}\ (\lambda x.x)\, a) = \mathrm{strip}\, e_1$ et $\llbracket e \rrbracket_2 = (\mathsf{let}\ a = 2\ \mathsf{in}\ (\lambda x.x)\, a) = \mathrm{strip}\, e_2$.

Substitution Si e est une expression et v une valeur, on note $e[x \Leftarrow v]$ l'expression obtenue en substituant v à x dans e. Les règles de substitution sont usuelles pour toutes les constructions, mis à part pour les crochets :

$$\langle e_1 \mid e_2 \rangle [x \Leftarrow v] = \langle e_1[x \Leftarrow \llbracket v \rrbracket_1] \mid e_2[x \Leftarrow \llbracket v \rrbracket_2] \rangle$$

Réduction Nous définissons une relation de réduction (ainsi que deux relations auxiliaires) sur le calcul partagé. Cette relation, $\xrightarrow{\varnothing}$, est décrite par les règles de la figure 3. Ces règles sont relativement nombreuses, mais elles peuvent être obtenues de manière quasi-systématique à partir de celles du langage de base, $\mathrm{ML}_{\mathrm{re}}$. Le jeu de règles se décompose en deux parties. Tout d'abord les *règles simples* correspondent à peu de choses près aux règles du calcul de départ. L'ajout de la règle (bracket) permet d'appliquer ces règles dans la sous-expression gauche d'un crochet ($i = 1$, $\xrightarrow{1}$), dans la sous-expression droite ($i = 2$, $\xrightarrow{2}$) et bien entendu en dehors des crochets ($i = \varnothing$, $\xrightarrow{\varnothing}$ noté par la suite $\rightarrow$).
Ensuite, les *règles « lift »* permettent de gérer le partage entre les deux expressions tout au long de la réduction : ces règles se contentent de déplacer les crochets à chaque fois qu'ils bloquent la réduction. Ces déplacements vont toujours dans le sens de la perte de partage : on ne repartage jamais deux sous-expressions qui ont été précédemment départagées.

Confluence En introduisant une règle (bracket) nous avons introduit une part de non-déterminisme dans les réductions : nous traitons de manière égale les deux expressions situées sous un crochet. Ainsi, lorsqu'on doit réduire $\langle e_1 \mid e_2 \rangle$, on peut commencer par s'intéresser à e_1 ou e_2 ou même entremêler les réductions de ces deux expressions. Du fait qu'il s'agit toutefois du seul cas non-déterministe, il est facile de montrer que le calcul partagé reste confluent, modulo renommage des adresses mémoires.

Lemme 3.1 (Confluence) *Soient e une expression et σ un état mémoire du calcul partagé. Si $e \,/\, \sigma \rightarrow f_1 \,/\, \tau_1$ et $e \,/\, \sigma \rightarrow f_2 \,/\, \tau_2$ alors il existe e' et σ' ainsi que deux renommages ϕ_1 et ϕ_2 tels que $f_1 \,/\, \tau_1 \rightarrow^{=} \phi_1(e') \,/\, \phi_1(\sigma')$ et $f_2 \,/\, \tau_2 \rightarrow^{=} \phi_2(e') \,/\, \phi_2(\sigma')$.*

3.2 Simulation

Nous allons prouver dans cette partie la *correction* et la *complétude* du calcul partagé. Nous entendons par correction le fait que si une expression du calcul partagé se réduit, alors ses deux projections peuvent effectuer séparément la même réduction. À l'inverse la complétude signifie que si les deux projections d'une expression peuvent se réduire alors l'expression toute entière peut se réduire. Nous aurons ainsi montré que la réduction introduite pour le calcul partagé simule parfaitement la réduction dans le calcul originel.

Définition 3.2 (États bloqués) *Soient e une expression et σ un état mémoire du calcul partagé. On dit que $e \,/\, \sigma$ est un état bloqué si et seulement si $e \,/\, \sigma$ est une forme normale pour $\rightarrow$ (i.e. ne se réduit pas) et e n'est pas une réponse.*

Lemme 3.2 *Soient e une expression et σ un état mémoire du calcul partagé. Si $e \,/\, \sigma$ est un état bloqué alors il existe $i \in \{1, 2\}$ tel que $\llbracket e \rrbracket_i \,/\, \llbracket \sigma \rrbracket_i$ soit un état bloqué.*

Démonstration. Nous procédons par induction sur la forme de l'expression e.

$e = v_1\, v_2$ v_1 n'est pas de la forme $\langle v_{11} \mid v_{12} \rangle$ ni de la forme $\lambda x.f$. On en déduit que $\llbracket v_1 \rrbracket_i$ n'est pas de la forme $\lambda x.f$ pour $i = 1$ et 2 et donc que $\llbracket e \,/\, \sigma \rrbracket_i$ est un état bloqué.

Règles simples : $(i \in \{\varnothing, 1, 2\})$

$$(\lambda x.e)\, v \;/\; \sigma \xrightarrow{i} e[x \Leftarrow v] \;/\; \sigma \qquad (\beta)$$
$$\mathsf{let}\ x = v\ \mathsf{in}\ e \;/\; \sigma \xrightarrow{i} e[x \Leftarrow v] \;/\; \sigma \qquad (\text{let})$$
$$\mathsf{ref}\ v \;/\; \sigma \xrightarrow{i} m \;/\; \sigma \oplus [m \mapsto \text{new}_i\, v] \qquad (\text{ref})$$
$$m := v \;/\; \sigma \xrightarrow{i} () \;/\; \sigma \otimes [m \mapsto \text{update}_i(\sigma(m)/v)] \qquad (\text{assign})$$
$$!\, m \;/\; \sigma \xrightarrow{i} \lfloor\!\lfloor \sigma(m) \rfloor\!\rfloor_i \;/\; \sigma \qquad (\text{deref})$$
$$\mathsf{raise}\ (\mathsf{exn}_\varepsilon\ v) \;/\; \sigma \xrightarrow{i} \mathsf{raised}_\varepsilon\ v \;/\; \sigma \qquad (\text{raise})$$

$$\mathsf{bind}\ x = v\ \mathsf{in}\ e \;/\; \sigma \xrightarrow{i} e[x \Leftarrow v] \;/\; \sigma \qquad (\text{bind})$$
$$\mathsf{try}_\varepsilon\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ e \;/\; \sigma \xrightarrow{i} e[x \Leftarrow v] \;/\; \sigma \qquad (\text{handle})$$
$$\mathsf{tryall}\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ e \;/\; \sigma \xrightarrow{i} e[x \Leftarrow (\mathsf{exn}_\varepsilon\ v)] \;/\; \sigma \qquad (\text{handle-all})$$
$$E[a] \;/\; \sigma \xrightarrow{i} a \;/\; \sigma \qquad \text{si } E \lll \lfloor\!\lfloor a \rfloor\!\rfloor_1 \text{ et } E \lll \lfloor\!\lfloor a \rfloor\!\rfloor_2 \qquad (\text{throw-context})$$

$$\frac{e \;/\; \sigma \xrightarrow{i} e' \;/\; \sigma'}{E[e] \;/\; \sigma \xrightarrow{i} E[e'] \;/\; \sigma'} \qquad (\text{contexte})$$

$$\frac{e_i \;/\; \sigma \xrightarrow{i} e'_i \;/\; \sigma' \qquad e_j = e'_j \qquad \{i, j\} = \{1, 2\}}{\langle e_1 \mid e_2 \rangle \;/\; \sigma \xrightarrow{\varnothing} \langle e'_1 \mid e'_2 \rangle \;/\; \sigma'} \qquad (\text{bracket})$$

Règles « lift » :

$$\langle v_1 \mid v_2 \rangle\, v \;/\; \sigma \xrightarrow{\varnothing} \langle v_1\, \lfloor\!\lfloor v \rfloor\!\rfloor_1 \mid v_2\, \lfloor\!\lfloor v \rfloor\!\rfloor_2 \rangle \;/\; \sigma \qquad (\text{lift-app})$$
$$\langle v_1 \mid v_2 \rangle := v \;/\; \sigma \xrightarrow{\varnothing} \langle v_1 := \lfloor\!\lfloor v \rfloor\!\rfloor_1 \mid v_2 := \lfloor\!\lfloor v \rfloor\!\rfloor_2 \rangle \;/\; \sigma \qquad (\text{lift-assign})$$
$$!\, \langle v_1 \mid v_2 \rangle \;/\; \sigma \xrightarrow{\varnothing} \langle !\, v_1 \mid !\, v_2 \rangle \;/\; \sigma \qquad (\text{lift-deref})$$
$$\mathsf{raise}\ \langle v_1 \mid v_2 \rangle \;/\; \sigma \xrightarrow{\varnothing} \langle \mathsf{raise}\ v_1 \mid \mathsf{raise}\ v_2 \rangle \;/\; \sigma \qquad (\text{lift-raise})$$

$$E[\langle a_1 \mid a_2 \rangle] \;/\; \sigma \xrightarrow{\varnothing} \langle \lfloor\!\lfloor E \rfloor\!\rfloor_1 [a_1] \mid \lfloor\!\lfloor E \rfloor\!\rfloor_2 [a_2] \rangle \;/\; \sigma \qquad (\text{lift-context})$$

Si (bind), (handle), (handle-all) et (throw-context) ne s'appliquent pas

Primitives d'écriture dans l'état mémoire :

$$\text{new}_1\, v = \langle v \mid \varnothing \rangle \qquad \text{update}_1(v_0/v) = \langle v \mid \lfloor\!\lfloor v_0 \rfloor\!\rfloor_2 \rangle$$
$$\text{new}_2\, v = \langle \varnothing \mid v \rangle \qquad \text{update}_2(v_0/v) = \langle \lfloor\!\lfloor v_0 \rfloor\!\rfloor_1 \mid v \rangle$$
$$\text{new}_\varnothing\, v = v \qquad \text{update}_\varnothing(v_0/v) = v$$

Si $m \notin \text{dom}(\sigma)$, $\sigma \oplus [m \mapsto v]$ dénote l'état mémoire qui étend σ et associe v à m. Si $m \in \text{dom}(\sigma)$, $\sigma \otimes [m \mapsto v]$ dénote l'état mémoire qui associe v à m et qui se comporte comme σ pour les autres adresses.

Fig. 3: Réduction du calcul partagé

$e = \mathsf{let}\ x = v\ \mathsf{in}\ e'$ **ou** $e = \mathsf{ref}\ v$ $e \mathbin{/} \sigma$ ne peut être un état bloqué.

$e = v_1 := v_2$ v_1 n'est pas de la forme $\langle v_{11} \mid v_{12}\rangle$ ni de la forme m. On en déduit que $\lfloor\!\lfloor v_1 \rfloor\!\rfloor_i$ n'est pas de la forme m pour $i = 1$ et 2 et donc que $\lfloor\!\lfloor e \mathbin{/} \sigma \rfloor\!\rfloor_i$ est un état bloqué.

$e = !\,v$ v n'est pas de la forme $\langle v_1 \mid v_2\rangle$ ni de la forme m. On en déduit que $\lfloor\!\lfloor v \rfloor\!\rfloor_i$ n'est pas de la forme m pour $i = 1$ et 2 et donc que $\lfloor\!\lfloor e \mathbin{/} \sigma \rfloor\!\rfloor_i$ est un état bloqué.

$e = \mathsf{raise}\ v$ v n'est pas de la forme $\langle v_1 \mid v_2\rangle$ ni de la forme $\mathsf{exn}_\varepsilon\ w$. On en déduit que $\lfloor\!\lfloor v \rfloor\!\rfloor_i$ n'est pas de la forme $\mathsf{exn}_\varepsilon\ w$ pour $i = 1$ et 2 et donc que $\lfloor\!\lfloor e \mathbin{/} \sigma \rfloor\!\rfloor_i$ est un état bloqué.

$e = E[e_1]$ En considérant les règles (let), (throw-context) et (lift-context), on constate que si e_1 est une réponse alors e se réduit. On en déduit que e_1 n'est pas une réponse. On en déduit que $e_1 \mathbin{/} \sigma$ est bloqué et on obtient le résultat par hypothèse d'induction.

$e = \langle e_1 \mid e_2\rangle$ $e \mathbin{/} \sigma$ est un état bloqué. $e_1 \mathbin{/} \sigma$ et $e_2 \mathbin{/} \sigma$ ne se réduisent pas et nécessairement e_i n'est pas une réponse pour $i = 1$ ou 2. On en déduit que $e_i \mathbin{/} \lfloor\!\lfloor \sigma \rfloor\!\rfloor_i$ est un état bloqué, ce qui donne le résultat. □

Lemme 3.3 (Correction) *Soient e, e' deux expressions et σ, σ' deux états mémoire du calcul partagé. Si $e \mathbin{/} \sigma \rightarrow e' \mathbin{/} \sigma'$ alors $\lfloor\!\lfloor e \rfloor\!\rfloor_i \mathbin{/} \lfloor\!\lfloor \sigma \rfloor\!\rfloor_i \rightarrow^{=} \lfloor\!\lfloor e' \rfloor\!\rfloor_i \mathbin{/} \lfloor\!\lfloor \sigma' \rfloor\!\rfloor_i$ pour $i \in \{1, 2\}$.*

Démonstration. On vérifie successivement, en observant chaque règle de réduction, que si $e \mathbin{/} \sigma \xrightarrow{i} e' \mathbin{/} \sigma'$ pour $i \in \{1, 2\}$ alors $e \mathbin{/} \lfloor\!\lfloor \sigma \rfloor\!\rfloor_i \rightarrow e' \mathbin{/} \lfloor\!\lfloor \sigma' \rfloor\!\rfloor_i$ puis que si $e \mathbin{/} \sigma \rightarrow e' \mathbin{/} \sigma'$ alors $\lfloor\!\lfloor e \rfloor\!\rfloor_i \mathbin{/} \lfloor\!\lfloor \sigma \rfloor\!\rfloor_i \rightarrow^{=} \lfloor\!\lfloor e' \rfloor\!\rfloor_i \mathbin{/} \lfloor\!\lfloor \sigma' \rfloor\!\rfloor_i$ pour $i \in \{1, 2\}$. □

Lemme 3.4 (Complétude) *Soient e une expression et σ un état mémoire du calcul partagé. Si $\lfloor\!\lfloor e \rfloor\!\rfloor_i / \lfloor\!\lfloor \sigma \rfloor\!\rfloor_i \rightarrow^* a_i / \tau_i$ (pour $i \in \{1, 2\}$) alors il existe une réponse a et un état mémoire τ du calcul partagé tels que $e/\sigma \rightarrow^* a/\tau$ et $a_i = \phi_i(\lfloor\!\lfloor a \rfloor\!\rfloor_i)$ (pour tout $i \in \{1, 2\}$, où ϕ_i est un remommage des adresses mémoire).*

Démonstration. On note, pour $i \in \{1, 2\}$, $e_i = \lfloor\!\lfloor e \rfloor\!\rfloor_i$ et $\sigma_i = \lfloor\!\lfloor \sigma \rfloor\!\rfloor_i$. En remarquant qu'on ne peut appliquer qu'un nombre fini de fois successivement une règle « lift » pour réduire une expression du calcul partagé, on vérifie grâce au lemme 3.3 que s'il existe une réduction infinie à partir de $e \mathbin{/} \sigma$ alors on peut construire une réduction infinie à partir de $e_1 \mathbin{/} \sigma_1$ ou de $e_2 \mathbin{/} \sigma_2$, ce qui est impossible puisque $e_1 \mathbin{/} \sigma_1$ et $e_2 \mathbin{/} \sigma_2$ se réduisent vers des réponses. On en déduit que $e \mathbin{/} \sigma$ peut se réduire en une forme normale $e' \mathbin{/} \sigma'$.
Procédons par induction sur la longueur de la réduction $e \mathbin{/} \sigma \rightarrow^* e' \mathbin{/} \sigma'$. Si cette dérivation est de longueur nulle, on en déduit que e est une réponse et que $\lfloor\!\lfloor e \rfloor\!\rfloor_i = a_i$ ce qui donne le résultat voulu.
Si $e \mathbin{/} \sigma \rightarrow e_0 \mathbin{/} \sigma_0 \rightarrow^* e' \mathbin{/} \sigma'$ alors $e_i \mathbin{/} \sigma_i \rightarrow \lfloor\!\lfloor e_0 \rfloor\!\rfloor_i \mathbin{/} \lfloor\!\lfloor \sigma_0 \rfloor\!\rfloor_i$ (lemme 3.3). Pour $i \in \{1, 2\}$, il existe un renommage des adresses mémoire ϕ_i tel que $\lfloor\!\lfloor e_0 \rfloor\!\rfloor_i \mathbin{/} \lfloor\!\lfloor \sigma_0 \rfloor\!\rfloor_i \rightarrow^* \phi_i(a_i) \mathbin{/} \phi_i(\tau_i)$ (lemme 3.1). Par hypothèse d'induction, il existe une réponse a et un état mémoire τ du calcul partagé, ainsi que deux renommages ϕ'_1 et ϕ'_2 tels que $e_0 \mathbin{/} \sigma_0 \rightarrow^* a \mathbin{/} \tau$ et $a_i = \phi'_i(\phi_i(a_i))$, ce qui permet de conclure. □

4 Le système SB(S)

SB(S) est un système de types intermédiaire pour le calcul partagé. Ce système, dont la forme est inspirée de [Pot01], est *brut* : il ne comporte pas de notion de variable de type, mais des *types monomorphes* qui appartiennent à un certain ensemble $\mathcal{S}$ et des *types polymorphes* qui sont des sous-ensembles particuliers de $\mathcal{S}$. Notre système est indépendant d'un certain nombre de choix : on se contente d'énoncer quelques propriétés nécessaires sur $\mathcal{S}$ sans spécifier cet ensemble totalement. Ainsi, nos résultats peuvent par exemple s'appliquer dans un cadre avec sous-typage atomique (et donc sans les types $\bot$ et $\top$) ou non-atomique, avec ou sans types récursifs, suivant l'expressivité que l'on souhaite lui donner.

4.1 Algèbre des types

Types monomorphes On suppose donnés deux ensembles : $\mathcal{S}$, un ensemble de *types monomorphes* (dont les éléments seront notés par la suite s) et $\mathcal{A}$, un ensemble d'*alternatives* (dont les éléments seront quant à eux notés a).
On suppose que $\mathcal{S}$ et $\mathcal{A}$ sont ordonnés par deux relations d'ordre notées $\leq_s$ et $\leq_a$ (ou simplement $\leq$ lorsque le contexte lève toute ambiguïté). Ces deux ensembles ne sont pas nécessairement des treillis. On note cependant $s_1 \sqcup s_2$ (resp. $a_1 \sqcup a_2$) tout élément $s \in \mathcal{S}$ (resp. $a \in \mathcal{A}$) tel que $s_1 \leq s$ et $s_2 \leq s$ (resp. $a_1 \leq a$ et $a_2 \leq a$), s'il en existe un.

$$\mathsf{int}^{\ell_1} \leq \mathsf{int}^{\ell_2} \Leftrightarrow \ell_1 \leq \ell_2 \qquad \mathsf{unit}^{\ell_1} \leq \mathsf{unit}^{\ell_2} \Leftrightarrow \ell_1 \leq \ell_2$$

$$(s_1' \xrightarrow{pc_1\ [r_1]} s_1)^{\ell_1} \leq (s_2' \xrightarrow{pc_2\ [r_2]} s_2)^{\ell_2} \Leftrightarrow s_1' \geq s_2',\ pc_1 \geq pc_2,\ r_1 \leq r_2,\ s_1 \leq s_2,\ \ell_1 \leq \ell_2$$

$$(r_1\ \mathsf{exn})^{\ell_1} \leq (r_2\ \mathsf{exn})^{\ell_2} \Leftrightarrow r_1 \leq r_2,\ \ell_1 \leq \ell_2 \qquad (s_1\ \mathsf{ref})^{\ell_1} \leq (s_2\ \mathsf{ref})^{\ell_2} \Leftrightarrow \ell_1 \leq \ell_2,\ s_1 = s_2$$

$$\mathsf{Pre}^{\ell_1} s_1 \leq \mathsf{Pre}^{\ell_2} s_2 \Leftrightarrow \ell_1 \leq \ell_2,\ s_1 \leq s_2$$

Fig. 4: Axiomes sur la relation de sous-typage

Types polymorphes (ou polytypes) Un *polytype* est un ensemble clos de types monomorphes. Les polytypes sont notés par la suite p et leur ensemble $\mathcal{P}$.
Dans notre système, les polytypes n'apparaîtront que dans les environnements. On s'autorisera à écrire s dans un contexte où un polytype est attendu pour dénoter $\{s\}$.

Rangées Une *rangée* (d'alternatives) r indexée par $\Lambda \subseteq \mathcal{E}$ est une application de Λ dans $\mathcal{A}$. Λ est le *domaine* de r. Étant donné $a \in \mathcal{A}$, on note $\partial_\Lambda a$ la rangée constante de valeur a indexée par Λ. Si r' est une rangée indexée par Λ, $a \in \mathcal{A}$ et $\varepsilon \in \mathcal{A} \backslash \Lambda$ alors $r = [\varepsilon_0.a; r']$ est la rangée indexée par $\Lambda \cup \{\varepsilon_0\}$ telle que $r(\varepsilon_0) = a$ et $\forall \varepsilon \in \Lambda,\ r(\varepsilon) = r'(\varepsilon)$. Inversement, si $r = [\varepsilon.a; r']$, on pose $r \backslash \varepsilon = r'$.
On étend la relation d'ordre sur les alternatives aux rangées en posant $r_1 \leq r_2$ si et seulement si $\mathrm{dom}(r_1) = \mathrm{dom}(r_2)$ et pour tout $\varepsilon \in \mathrm{dom}(r_1)$, $r_1(\varepsilon) \leq r_2(\varepsilon)$.

Axiomes Afin de pouvoir écrire les règles de typage, nous devons faire quelques suppositions sur les ensembles $\mathcal{S}$ et $\mathcal{A}$ en exprimant l'existence d'un certain nombre de constructeurs de types.

Axiome 1 (Constructions sur $\mathcal{S}$) L'ensemble $\mathcal{S}$ est clos pour les constructions suivantes :
- **Entiers :** Si $\ell \in \mathcal{L}$ alors $\mathsf{int}^\ell \in \mathcal{S}$.
- **Unit :** Si $\ell \in \mathcal{L}$ alors $\mathsf{unit}^\ell \in \mathcal{S}$.
- **Fonctions :** Si $\ell, pc \in \mathcal{L}$; $s, s' \in \mathcal{S}$ et r est une rangée indexée par $\mathcal{E}$ alors $(s' \xrightarrow{pc\ [r]} s)^\ell \in \mathcal{S}$.
- **Références :** Si $\ell \in \mathcal{L}$ et $s \in \mathcal{S}$ alors $(s\ \mathsf{ref})^\ell \in \mathcal{S}$.
- **Exceptions :** Si $\ell \in \mathcal{L}$ et r est une rangée indexée par $\mathcal{E}$ alors $(r\ \mathsf{exn})^\ell \in \mathcal{S}$.

Axiome 2 (Constructions sur $\mathcal{A}$) L'ensemble $\mathcal{A}$ est clos pour la construction suivante :
- **Présent :** Si $\ell \in \mathcal{L}$ et $s \in \mathcal{S}$ alors $\mathsf{Pre}^\ell s \in \mathcal{A}$.

Dans la pratique, une alternative permet de préciser si une expression est susceptible ou non de déclencher une exception donnée. Pour avoir une expressivité suffisante, il est donc nécessaire qu'une autre forme d'alternative soit disponible, par exemple Abs (pour signifier qu'une expression ne déclenche pas d'exceptions). Cependant, elle n'est pas utile pour prouver la validité du système. Nous ne la précisons donc pas pour l'instant.
On suppose également que les constructions précisées ci-dessus se comportent de manière standard vis-à-vis de la relation de sous-typage.

Axiome 3 (Sous-typage sur les constructions) Les cinq formes de types précisées dans l'axiome 1 sont incomparables entre elles (pour la relation $\leq_s$). On suppose de plus que $\leq_a$ et $\leq_s$ vérifient les relations de la figure 4.

Garde Nous aurons besoin d'exprimer qu'un type s donné est « marqué » par un niveau d'information au moins égal à une étiquette ℓ. Nous introduisons pour cela une relation $\lhd$ entre étiquettes et types : $\ell \lhd s$ (« ℓ garde s »). Nous supposons que cette relation vérifie les deux axiomes suivants.

Axiome 4 (Transitivité) Si $\ell \leq \ell'$, $\ell' \lhd s$ et $s \leq s'$ alors $\ell \lhd s'$

Axiome 5 (Constructions) Soit $s = \mathsf{int}^\ell$ (resp. unit^ℓ, $(s' \xrightarrow{pc\ [r]} s)^\ell$, $(s\ \mathsf{ref})^\ell$, $(r\ \mathsf{exn})^\ell$). On a $\ell_0 \lhd s$ si et seulement si $\ell_0 \leq \ell$.

Niveau des rangées Soit μ une application croissante de l'ensemble des alternatives dans l'ensemble des étiquettes qui vérifie $\mu(\mathsf{Pre}^{\ell} s) = \ell$. On étend μ aux rangées en posant $\mu(r) = \bigsqcup_{\varepsilon \in \mathrm{dom}(r)} \mu(r(\varepsilon))$.

On note $\mu(r_1, r_2)$ pour $\mu(r_1) \sqcup \mu(r_2)$. On note également $\ell \lhd r$ si et seulement si $\forall \varepsilon \in \mathrm{dom}(r)$, $\mathsf{Pre}^{\text{-}} s \leq r(\varepsilon) \Rightarrow \mathsf{Pre}^{\ell} s \leq r(\varepsilon)$.

4.2 Relation de typage

Notre système comporte deux sortes de jugements mutuellement définies, l'une portant sur toutes les expressions et l'autre limitée aux valeurs. Un environnement $\Gamma = X \cup M$ est la réunion de deux applications partielles : X des variables dans les polytypes et M des adresses mémoire dans les monotypes.
Les jugements portant sur des valeurs sont de la forme

$$\Gamma \vdash_L v : s$$

Γ est l'environnement de typage et s un type monomorphe. Le jugement $\Gamma \vdash_L v : s$ signifie donc que « *dans l'environnement* Γ*, la valeur* v *a le type* s ».

Les expressions se réduisant en produisant des effets de bord, nous avons besoin de deux informations complémentaires pour les typer : pc, le niveau d'information lié au compteur de programme (le contexte d'exécution) et r une rangée d'alternatives indexée par $\mathcal{E}$ qui indique les exceptions éventuellement levées lors de la réduction de l'expression. Les jugements portant sur les expressions sont ainsi de la forme

$$pc, \Gamma \vdash_L e : s \ [r]$$

Dans ces deux sortes de jugements, L représente l'ensemble des niveaux d'information considérés secrets, c'est-à-dire l'ensemble des niveaux associés aux sous-expressions apparaissant dans des crochets $\langle \ldots \mid \ldots \rangle$. On le suppose dans cette partie fixé et clos (i.e. si $\ell \in \mathcal{L}$ et $\ell \leq \ell'$ alors $\ell' \in \mathcal{L}$). On ne l'indique plus par la suite sur les jugements pour alléger les notations.
On écrira également $\Gamma \vdash v : p$ comme abréviation de $(\forall s \in p)\ \Gamma \vdash v : s$.

Enfin, pour définir la relation de typage, nous avons besoin d'une relation $.\Downarrow$ qui indique si une expression comporte une exception levée non rattrapée. On a $e\Downarrow$ si et seulement si il existe une valeur v, $\varepsilon \in \mathcal{E}$ et des contextes $E_1, \ldots, E_n$ tels que $e = E_1[\ldots E_n[\mathsf{raised}_\varepsilon\ v] \ldots]$ et pour tout i, E_i ne rattrape pas la réponse $\mathsf{raised}_\varepsilon\ v$.

Les deux relations sont définies par les règles de la figure 5. Nous avons pris pour convention d'inscrire une étoile $*$ à la place des méta-variables libres n'ayant qu'une seule occurrence dans une règle.

États mémoire On dit qu'un état mémoire σ est bien typé dans un environnement $\Gamma = X \cup M$ et on note $\Gamma \vdash \sigma$ si et seulement si $\mathrm{dom}(\sigma) = \mathrm{dom}(M)$ et pour tout $m \in \mathrm{dom}(\sigma)$, $\Gamma \vdash m : M(m)$. On écrit $pc, \Gamma \vdash e \,/\, \sigma : s\ [r]$ pour $pc, \Gamma \vdash e : s\ [r]$ et $\Gamma \vdash \sigma$.

4.3 Visite informelle

Avant de poursuivre notre étude et de prouver la propriété de *subject reduction*, commentons les points importants des règles du système SB(S) présenté sur la figure 5.

Notons tout d'abord que les règles v-Loc, e-Nothing et e-Raised sont des règles « internes » au système dans la mesure où elles ne seront pas utilisées pour typer des expressions écrites par le programmeur. Nous devons cependant les préciser afin de pouvoir énoncer le théorème de *subject reduction*.
La règle e-Value permet quant à elle de considérer un jugement portant sur une valeur comme un jugement d'expression. Une valeur ne peut produire d'effet de bord : le choix du pc et de la rangée r est donc libre.

Compteur de programme Pour typer correctement les opérations qui effectuent des effets de bord, il nous faut propager en chaque point du programme le niveau d'information associé au compteur de programme et qui est noté pc sur les jugements.
Ce niveau est augmenté lorsqu'on traverse des crochets $\langle \ldots \mid \ldots \rangle$ par la règle e-Bracket. Il est alors pollué par n'importe quelle étiquette pc' qui est secrète (c'est-à-dire qui appartient à L). On impose également que

Valeurs :

$$\textsc{v-Int}\quad \Gamma \vdash k : \mathsf{int}^* \qquad \textsc{v-Unit}\quad \Gamma \vdash () : \mathsf{unit}^* \qquad \textsc{v-Nothing}\quad \Gamma \vdash \varnothing : * \qquad \textsc{v-Loc}\quad \Gamma \vdash m : (\Gamma(m)\ \mathsf{ref})^* \qquad \textsc{v-Var}\quad \frac{s \in \Gamma(x)}{\Gamma \vdash x : s}$$

$$\textsc{v-Exn}\quad \frac{\Gamma \vdash v : s}{\Gamma \vdash \mathsf{exn}_\varepsilon\ v : ([\varepsilon.\mathsf{Pre}^* s; *]\ \mathsf{exn})^*} \qquad \textsc{v-Abs}\quad \frac{pc, \Gamma[x \mapsto s'] \vdash e : s\ \ [r]}{\Gamma \vdash \lambda x.e : (s' \xrightarrow{pc\ [r]} s)^*} \qquad \textsc{v-Bracket}\quad \frac{\Gamma \vdash v_1 : s \qquad \Gamma \vdash v_2 : s \qquad (\exists \ell \in L)\ \ell \lhd s}{\Gamma \vdash \langle v_1 \mid v_2 \rangle : s}$$

$$\textsc{v-Sub}\quad \frac{\Gamma \vdash v : s' \qquad s' \leq s}{\Gamma \vdash v : s}$$

Expressions :

$$\textsc{e-Value}\quad \frac{\Gamma \vdash v : s}{*, \Gamma \vdash v : s\ \ [*]} \qquad \textsc{e-App}\quad \frac{\Gamma \vdash v_1 : (s' \xrightarrow{pc \sqcup \ell\ [r]} s)^\ell \qquad \Gamma \vdash v_2 : s' \qquad \ell \lhd s}{pc, \Gamma \vdash v_1\, v_2 : s\ \ [r]} \qquad \textsc{e-Ref}\quad \frac{\Gamma \vdash v : s \qquad pc \lhd s}{pc, \Gamma \vdash \mathsf{ref}\ v : (s\ \mathsf{ref})^*\ \ [*]}$$

$$\textsc{e-Assign}\quad \frac{\Gamma \vdash v_1 : (s\ \mathsf{ref})^\ell \qquad \Gamma \vdash v_2 : s \qquad \ell \sqcup pc \lhd s}{pc, \Gamma \vdash v_1 := v_2 : \mathsf{unit}^\ell\ \ [*]} \qquad \textsc{e-Deref}\quad \frac{\Gamma \vdash v : (s'\ \mathsf{ref})^\ell \qquad s' \leq s \qquad \ell \lhd s}{pc, \Gamma \vdash\ !\, v : s\ \ [*]}$$

$$\textsc{e-Raise}\quad \frac{\Gamma \vdash v : (r\ \mathsf{exn})^\ell \qquad pc \sqcup \ell \lhd r}{pc, \Gamma \vdash \mathsf{raise}\ v : *\ \ [r]} \qquad \textsc{e-Raised}\quad \frac{\Gamma \vdash v : s}{pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : *\ \ [\varepsilon.\mathsf{Pre}^{pc} s; r']} \qquad \textsc{e-Let}\quad \frac{\Gamma \vdash v : p \qquad pc, \Gamma[x \mapsto p] \vdash e : s\ \ [r]}{pc, \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : s\ \ [r]}$$

$$\textsc{e-Bind}\quad \frac{pc, \Gamma \vdash e_1 : s'\ \ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma[x \mapsto s'] \vdash e_2 : s\ \ [r_2]}{pc, \Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : s\ \ [r_1 \sqcup r_2]}$$

$$\textsc{e-Try}\quad \frac{pc, \Gamma \vdash e_1 : s\ \ [\varepsilon.\mathsf{Pre}^\ell s'; r_1'] \qquad pc \sqcup \ell, \Gamma[x \mapsto s'] \vdash e_2 : s\ \ [r_2] \qquad \ell \lhd s}{pc, \Gamma \vdash \mathsf{try}_\varepsilon\ e_1\ \mathsf{with}\ x \succ e_2 : s\ \ [[\varepsilon.*; r_1'] \sqcup r_2]}$$

$$\textsc{e-TryAll}\quad \frac{pc, \Gamma \vdash e_1 : s\ \ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma[x \mapsto (r_1\ \mathsf{exn})^*] \vdash e_2 : s\ \ [r_2] \qquad \mu(r_1) \lhd s}{pc, \Gamma \vdash \mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ e_2 : s\ \ [r_2]}$$

$$\textsc{e-Bracket}\quad \frac{pc \sqcup pc', \Gamma \vdash e_1 : s\ \ [r] \qquad pc \sqcup pc', \Gamma \vdash e_2 : s\ \ [r] \qquad pc' \in L \qquad (pc' \lhd s) \vee e_1\Downarrow \vee e_2\Downarrow}{pc, \Gamma \vdash \langle e_1 \mid e_2 \rangle : s\ \ [r]}$$

$$\textsc{e-Sub}\quad \frac{pc, \Gamma \vdash e : s'\ \ [r'] \qquad s' \leq s \qquad r' \leq r}{pc, \Gamma \vdash e : s\ \ [r]}$$

Fig. 5: Le système SB(S)

le type donné à l'expression à crochets soit pollué par le niveau pc' (prémisse $pc' \lhd s$). La condition $e_1 \Downarrow \vee e_2 \Downarrow$ n'est présente que pour des raisons techniques, afin de conserver la propriété de *subject reduction* sur les étapes intermédiaires d'un calcul.
La propagation du *pc* est immédiate pour toutes les règles mis à part l'application : le corps d'une fonction est en effet exécuté sous le *pc* du point d'appel qui peut être supérieur à celui où la fonction est définie. Pour cela, le corps d'une fonction est typé avec un *pc* « libre », indépendamment de celui du point de définition, et qui est ensuite indiqué sur la flèche du type fonctionnel. Ce *pc* est lié à celui du (ou des) point(s) d'application par la règle E-APP.

Références Les références sont typées par les règles E-REF, E-ASSIGN et E-DEREF. Les types inférés pour une adresse mémoire sont de la forme $(s\ \mathsf{ref})^{\ell}$. Pour garantir l'absence de fuite d'information, il faut que le niveau du type s soit pollué par le *pc* de chaque point où la référence est susceptible d'être modifiée : c'est le rôle de la prémisse $pc \sqcup \ell \lhd s$ de la règle E-ASSIGN. Notons que pour cette règle, l'invariance du constructeur ref est primordiale, on pourrait sinon artificiellement monter le niveau d'une référence pour pouvoir écrire dedans.

Exceptions L'analyse sur les exceptions est effectuée grâce à la rangée r associée à chaque jugement portant sur une expression. Cette rangée comporte un champ par nom d'exception. Ce champ peut indiquer *a priori* trois informations :
- Si l'expression est susceptible ou non de déclencher une exception du nom correspondant,
- Le niveau *pc* du point auquel l'exception peut être déclenchée,
- Le type de la valeur contenue dans l'exception susceptible d'être levée.

Le déclenchement des exceptions est traité par les règles E-RAISE, V-EXN et E-RAISED. Lorsqu'une exception de nom ε est levée avec une valeur de type s à un point du programme de niveau *pc*, il faut que $r(\varepsilon) \geq \mathsf{Pre}^{pc} s$. Leur propagation est assurée par les règles relatives aux contextes. L'expression $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ propage toutes les exceptions levées par e_1 et e_2, la rangée qui lui est associée doit être supérieure à celle associée à chacune des deux sous-expressions. Pour les expressions $\mathsf{try}_{\varepsilon}\ e_1\ \mathsf{with}\ x \succ e_2$ et $\mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ e_2$, on propage la rangée associée à e_2 et éventuellement la partie de la rangée associée à e_1 correspondant aux exceptions non rattrapées. On pollue de plus le type s de l'expression par le niveau de(s) exception(s) rattrapées (m' pour try et $\mu(r_1)$, l'union de tous les niveaux, pour tryall).

4.4 Subject-reduction

On dit que la dérivation d'un jugement $pc, \Gamma \vdash e : s\ [r]$ est *simple* si et seulement si elle ne se termine pas par une instance de la règle E-SUB. De même, la dérivation d'un jugement $\Gamma \vdash : s$ est *simple* si et seulement si elle ne se termine pas par une instance de la règle V-SUB.
Il s'agit des seules règles à ne pas être dirigée par la syntaxe. Le lemme suivant nous permettra de nous ramener systématiquement au cas des dérivations simples dans nos preuves.

Lemme 4.1 (Dérivation simple) *Soit e une expression. Si $pc, \Gamma \vdash e : s\ [r]$ alors il existe une dérivation simple de $pc, \Gamma \vdash e : s'\ [r']$ pour $s' \leq s$ et $r' \leq r$.*
De même, soit v une valeur. Si $\Gamma \vdash v : s$ alors il existe une dérivation simple de $\Gamma \vdash v : s'$ pour $s' \leq s$.

Démonstration. On procède par induction sur la dérivation de $pc, \Gamma \vdash e : s\ [r]$ et par cas suivant la dernière règle appliquée. Tous les cas sont des cas de base, mis à part celui de la règle E-SUB :

Règle E-SUB $pc, \Gamma \vdash e : s\ [r]$ est déduit d'une prémisse de la forme $pc, \Gamma \vdash e : s'\ [r']$ avec $s' \leq s$ et $r' \leq r$. Par hypothèse d'induction, il existe une dérivation simple de $pc, \Gamma \vdash e : s''\ [r'']$ pour $s'' \leq s'$ et $r'' \leq r'$, ce qui permet de conclure.

Le résultat sur les valeurs se montre de la même manière. □

Lemme 4.2 (Diminution du pc) *Soit e une expression du calcul partagé. Si $pc, \Gamma \vdash e : s\ [r]$ et $pc' \leq pc$ alors $pc', \Gamma \vdash e : s\ [r]$.*

Démonstration. Nous allons montrer ce lemme par induction sur la dérivation en procédant par cas suivant la dernière règle appliquée. Nous ne traitons que les cas suivants :

Règle E-APP On a $e = v_1\, v_2$. Les prémisses de la règle sont $\Gamma \vdash v_1 : (s' \xrightarrow{pc \sqcup \ell\ [r]} s)^\ell$ et $\Gamma \vdash v_2 : s'$ avec $\ell \lhd s$. $pc' \leq pc$ donc $pc' \sqcup \ell \leq pc \sqcup \ell$.

Par V-SUB, on obtient $\Gamma \vdash v_1 : (s' \xrightarrow{pc' \sqcup \ell\ [r]} s)^\ell$ puis, par E-APP, $pc', \Gamma \vdash v_1\, v_2 : s\ \ [r]$.

Règle E-ASSIGN On a $e = v_1 := v_2$. Les prémisses de la règle sont $\Gamma \vdash v_1 : (s'\ \mathsf{ref})^\ell$ et $\Gamma \vdash v_2 : s'$ avec $pc \sqcup \ell \lhd s'$ et $s = \mathsf{unit}^\ell$. Or $pc' \leq pc$ donc $pc' \sqcup \ell \lhd s'$. On peut donc appliquer la règle E-ASSIGN pour obtenir $pc', \Gamma \vdash v_1 := v_2 : s\ \ [r]$.

Règle E-RAISE On a $e = \mathsf{raise}\ v$. Les prémisses de la règle sont $\Gamma \vdash v : (r'\ \mathsf{exn})^\ell$ avec $pc \sqcup \ell \lhd r$. Puisque $pc' \leq pc$, on a $pc' \sqcup \ell \lhd r$ et on obtient le jugement $pc', \Gamma \vdash \mathsf{raise}\ v : s\ \ [r]$ en appliquant la règle E-RAISE.

Règle E-RAISED On a $e = \mathsf{raised}_\varepsilon\ v$. Les prémisses de la règle sont $\Gamma \vdash v : s'$ avec $pc \leq \ell$ et $r = [\varepsilon.\mathsf{Pre}^\ell; *]$. Puisque $pc' \leq pc$, on a $pc' \leq \ell$. En appliquant la règle E-RAISED, on obtient $pc', \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ \ [r]$.

Règle E-BRACKET On a $e = \langle e_1 \mid e_2 \rangle$. Les prémisses de la règle sont $pc \sqcup pc'', \Gamma \vdash e_1 : s\ \ [r]$ et $pc \sqcup pc'', \Gamma \vdash e_2 : s\ \ [r]$ avec $(pc'' \lhd s) \vee e_1\Downarrow \vee\, e_2\Downarrow$ et $pc'' \in L$. On a $pc' \leq pc$ donc $pc' \sqcup pc'' \leq pc \sqcup pc''$. On en déduit, par hypothèse d'induction, que $pc' \sqcup pc'', \Gamma \vdash e_1 : s\ \ [r]$ et $pc' \sqcup pc'', \Gamma \vdash e_2 : s\ \ [r]$. On obtient en appliquant la règle E-BRACKET le jugement $pc', \Gamma \vdash e : s\ \ [r]$. □

Lemme 4.3 (Domaine de l'environnement) *Soient e une expression et σ un état mémoire. Soit $\Gamma = X \cup M$ un environnement, X' la restriction de X à $\mathrm{fv}(e)$ et $\Gamma' = X' \cup M$. On a $pc, \Gamma \vdash e \,/\, \sigma : s\ \ [r]$ si et seulement si $pc, \Gamma' \vdash e \,/\, \sigma : s\ \ [r]$.*

Lemme 4.4 (Projections) *Soit e une expression du calcul partagé. Si $pc, \Gamma \vdash e : s\ \ [r]$ alors, pour $i \in \{1, 2\}$ on a $pc, \Gamma \vdash \lfloor e \rfloor_i : s\ \ [r]$. Soit v une valeur du calcul partagé. Si $\Gamma \vdash v : s$ alors, pour $i \in \{1, 2\}$ on a $\Gamma \vdash \lfloor v \rfloor_i : s$.*

Informellement, pour obtenir une dérivation correspondant à $\lfloor e \rfloor_i$ à partir d'une dérivation portant sur e, il suffit de supprimer toutes les occurrences d'une règle E-BRACKET et de conserver pour chacune d'entre elles une des deux branches en fonction de i.

Lemme 4.5 (new, update) *Soit $i = \varnothing, 1$ ou 2. Soit $s \in \mathcal{S}$. Dans les cas $i = 1$ et $i = 2$, on suppose de plus qu'il existe $\ell \in L$ tel que $\ell \lhd s$. Soient v et v' deux valeurs. Si $\Gamma \vdash v : s$ et $\Gamma \vdash v' : s$ alors, pour $i \in \{\varnothing, 1, 2\}$, $\Gamma \vdash \mathrm{update}_i(v'/v) : s$ et $\Gamma \vdash \mathrm{new}_i\, v : s$.*

Lemme 4.6 *Soit $v = \langle v_1 \mid v_2 \rangle$ une valeur du calcul partagé. Si $\Gamma \vdash \langle v_1 \mid v_2 \rangle : s$ alors $\Gamma \vdash v_i : s$ (pour $i \in \{1, 2\}$) et il existe $pc' \in L$ tel que $pc' \lhd s$.*
De plus que si s est de l'une des cinq formes précisées dans l'axiome 1 alors $\ell \in L$.

Démonstration. Il existe une dérivation simple de $\Gamma \vdash \langle v_1 \mid v_2 \rangle : s'$ pour $s' \leq s$ qui se termine par instance de la règle V-BRACKET dont les prémisses sont $\Gamma \vdash v_i : s'$ pour $i \in \{1, 2\}$ avec $pc' \lhd s'$ et $pc' \in L$. On en déduit que $\Gamma \vdash v_i : s'$ (règle V-SUB) et $pc' \lhd s$ (axiome 4). □

Lemme 4.7 (Substitution) *Soit v une valeur close. Si $pc, \Gamma[x \mapsto p] \vdash e : s\ \ [r]$ et pour tout $s' \in p$, $\Gamma \vdash v : s'$ alors $pc, \Gamma \vdash e[x \Leftarrow v] : s\ \ [r]$.*

Démonstration. Soit v une valeur close telle que pour tout $s' \in p$, $\Gamma \vdash v : s'$. On montre par induction sur la dérivation que si $pc, \Gamma[x \mapsto p] \vdash e : s\ \ [r]$ alors $pc, \Gamma \vdash e[x \Leftarrow v] : s\ \ [r]$ et que si w est une valeur et $\Gamma[x \mapsto p] \vdash w : s$ alors $\Gamma \vdash w[x \Leftarrow v] : s$.

Règle V-VAR On a $w = y$. On distingue deux cas. *Si $y \neq x$ alors* $w[x \Leftarrow v] = y$. On a $s \in \Gamma(y)$ donc $s \in \Gamma[x \mapsto p](y)$. On obtient le résultat par la règle V-VAR.

Si $y = x$ alors $w[x \Leftarrow v] = v$. On a $s \in p$. On en déduit que $\Gamma \vdash v : s$ et donc $\Gamma \vdash w[x \Leftarrow v] : s$.

Règle V-ABS e est de la forme $\lambda y.e'$ et s de la forme $(s'' \xrightarrow{pc'\ [r']} s')^\ell$. Le jugement a été obtenu de la prémisse $pc', \Gamma[x \mapsto p][y \mapsto s''] \vdash e' : s'\ \ [r']$. Deux sous cas interviennent suivant que $y = x$ ou que $y \neq x$.

Si $x = y$ alors On a $\lambda y.e' = e[x \Leftarrow v]$ donc $\Gamma \vdash e[x \Leftarrow v] : s$.

Si $x \neq y$ alors $\Gamma[x \mapsto p][y \mapsto s''] = \Gamma[y \mapsto s''][x \mapsto p]$. De plus, v étant close, pour tout $s' \in p$, on a toujours $\Gamma[y \mapsto s''] \vdash v : s'$. Par hypothèse d'induction, on a également $pc', \Gamma[y \mapsto s''] \vdash e'[x \Leftarrow v] : s'\ \ [r']$. Par la règle V-ABS, on obtient $\Gamma \vdash \lambda y.(e'[x \Leftarrow v]) : (s \xrightarrow{pc'\ [r']} s')^\ell$, c'est-à-dire $\Gamma \vdash e[x \Leftarrow v] : s$.

Les autres cas se traitent de manière immédiate en faisant valoir l'hypothèse d'induction. □

Théorème 4.1 (Subject reduction) *Soient e, e' deux expressions closes et σ, σ' deux états mémoire du calcul partagé tels que $e \,/\, \sigma \to e' \,/\, \sigma'$.*
Si $pc, \Gamma \vdash e \,/\, \sigma : s\ [r]$ alors il existe Γ' étendant Γ tel que $pc, \Gamma' \vdash e' \,/\, \sigma' : s\ [r]$.

Démonstration. Nous montrons par induction sur la règle de dérivation appliquée que si $e \,/\, \sigma \xrightarrow{i} e' \,/\, \sigma'$ ($i \in \{\varnothing, 1, 2\}$) et $pc, \Gamma \vdash e \,/\, \sigma : s\ [r]$ alors il existe Γ' tel que $pc, \Gamma' \vdash e' \,/\, \sigma' : s\ [r]$, en supposant également que (*) si $i \neq \varnothing$ alors $pc \in L$.

Le lemme 4.1 nous permet de nous ramener au cas où la dérivation (Δ) de $pc, \Gamma \vdash e : s\ [r]$ est simple. On suppose donc que l'on est dans cette situation et on raisonne par cas suivant la règle de réduction appliquée.

(β) On pose $e = (\lambda x.f)\, v$ et $e' = f[x \Leftarrow v]$. (Δ) se termine par une instance de E-APP de prémisses $\Gamma \vdash \lambda x.f : (s' \xrightarrow{pc \sqcup \ell\ [r]} s)^\ell$ et $\Gamma \vdash v : s'$. On déduit de $\Gamma \vdash \lambda x.f : (s' \xrightarrow{pc \sqcup \ell\ [r]} s)^\ell$ que $pc \sqcup \ell, \Gamma[x \mapsto s''] \vdash f : s\ [r]$ (lemmes 4.1, 4.2 et E-SUB) pour $s'' \geq s'$. Par V-SUB, on a $\Gamma \vdash v : s''$. On en déduit que $pc, \Gamma \vdash f[x \Leftarrow v] : s\ [r]$ (lemmes 4.7 et 4.2).

(let) On pose $e = \mathsf{let}\ x = v\ \mathsf{in}\ f$ et $e' = f[x \Leftarrow v]$. (Δ) se termine par une instance de E-LET de prémisses $(\forall s' \in p)\ \Gamma \vdash v : s'$ et $pc, \Gamma[x \mapsto p] \vdash f : s\ [r]$, on en déduit alors que $pc, \Gamma \vdash f[x \Leftarrow v] : s\ [r]$ (lemme 4.7).

(bind) On pose $e = \mathsf{bind}\ x = v\ \mathsf{in}\ f$ et $e' = f[x \Leftarrow v]$. (Δ) se termine par une instance de E-BIND de prémisses $pc, \Gamma \vdash v : s'\ [r_1]$ et $pc \sqcup \mu(r_1), \Gamma[x \mapsto s'] \vdash f : s\ [r_2]$ avec $r_2 \leq r$. On en déduit que $\Gamma \vdash v : s'$ et $pc, \Gamma[x \mapsto s'] \vdash f : s\ [r_2]$ (lemme 4.2). On a donc $pc, \Gamma \vdash f[x \Leftarrow v] : s\ [r]$ (lemme 4.7 et E-SUB).

(ref) On pose $e = \mathsf{ref}\ v$ et $e' = m$ avec $\sigma' = \sigma \oplus [m \mapsto \mathrm{new}_i\, v]$. (Δ) se termine par une instance de E-REF de prémisse $\Gamma \vdash v : s'$ avec $s = (s'\ \mathsf{ref})^\ell$ et $pc \lhd s'$. En prenant $\Gamma' = \Gamma[m \mapsto s']$, on obtient $\Gamma' \vdash m : s$ et $\Gamma' \vdash \sigma'$ (lemme 4.5). On en conclut $pc, \Gamma' \vdash e' \,/\, \sigma' : s\ [r]$.

(assign) On pose $e = m := v$. (Δ) se termine par une instance de E-ASSIGN de prémisses $\Gamma \vdash m : (s'\ \mathsf{ref})^\ell$ et $\Gamma \vdash v : s'$ avec $pc \sqcup \ell \lhd s'$ et $s = \mathsf{unit}^\ell$. On a $pc, \Gamma \vdash e' : s\ [r]$.
Par ailleurs, $\Gamma \vdash m : (s'\ \mathsf{ref})^\ell$ donc $\Gamma(m) = s'$. Or $\Gamma \vdash \sigma$ donc $\Gamma \vdash \sigma(m) : s'$. On déduit de $pc \lhd s'$ et (*) que $\Gamma \vdash \mathrm{update}_i(\sigma(m)/v) : s'$ (lemme 4.5) et donc que $\Gamma \vdash \sigma'$.

(deref) On pose $e = !\ m$ et $e' = \sigma(m)$. (Δ) se termine par une instance de E-DEREF de prémisse $\Gamma \vdash m : (s'\ \mathsf{ref})^\ell$ avec $s' \leq s$. On en déduit que $\Gamma(m) = s'$. Or $\Gamma \vdash \sigma$ donc $\Gamma \vdash \sigma(m) : s'$. On conclut, en appliquant E-VALUE puis E-SUB, que $pc, \Gamma \vdash e' : s\ [r]$.

(raise) On pose $e = \mathsf{raise}\ (\mathsf{exn}_\varepsilon\ v)$ et $e' = \mathsf{raised}_\varepsilon\ v$. (Δ) se termine par une instance de E-RAISE de prémisse $\Gamma \vdash \mathsf{exn}_\varepsilon\ v : (r\ \mathsf{exn})^\ell$ avec $pc \sqcup \ell \lhd r$.
Il existe une dérivation simple de $\Gamma \vdash \mathsf{exn}_\varepsilon\ v : (r'\ \mathsf{exn})^{\ell'}$ (avec $r' \leq r$ et $\ell' \leq \ell$) qui se termine par une instance de V-EXN. On en déduit que $r(\varepsilon) \geq \mathsf{Pre}^* s'$ avec $\Gamma \vdash v : s'$. Or $pc \lhd r$ donc $r(\varepsilon) \geq \mathsf{Pre}^{pc} s'$. On en déduit, par E-RAISED et E-SUB, que $pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ [r]$.

(throw-context) On pose $e = E[a]$ et $e' = a$.
Si $E = \mathsf{bind}\ x = [\ ]\ \mathsf{in}\ e_0$ *alors* $a = \mathsf{raised}_\varepsilon\ v$ ou $a = \langle \mathsf{raised}_{\varepsilon_1}\ v_1 \mid \mathsf{raised}_{\varepsilon_2}\ v_2 \rangle$. (Δ) se termine par une instance de E-BIND qui admet $pc, \Gamma \vdash a : s'\ [r_1]$ pour prémisse avec $r_1 \leq r$. On en déduit, par E-SUB, que $pc, \Gamma \vdash a : s'\ [r]$ puis que $pc, \Gamma \vdash a : s\ [r]$.

Supposons maintenant $E = \mathsf{try}_\varepsilon\ [\ ]\ \mathsf{with}\ x \succ e_0$. La dérivation de $pc, \Gamma \vdash e : s\ [r]$ se termine par une instance de la règle E-TRY dont une prémisse est $pc, \Gamma \vdash a : s\ [r_1]$. On distingue trois cas suivant la forme de a. Si a est une valeur v, on en déduit que $pc, \Gamma \vdash e' : s\ [r]$. Si $a = \mathsf{raised}_{\varepsilon'}\ v$ avec $\varepsilon' \neq \varepsilon$, on a $r_1(\varepsilon') \leq r(\varepsilon')$ et on en déduit également que $pc, \Gamma \vdash a : s\ [r]$. Si $a = \langle a_1 \mid a_2 \rangle$ et n'est pas une valeur, on a $pc \sqcup pc', \Gamma \vdash a_i : s\ [r_1]$ (pour $i \in \{1, 2\}$) avec $pc' \notin L$. De même que dans les deux cas précédents, on obtient $pc \sqcup pc', \Gamma \vdash a_i : s\ [r]$. Or a n'est pas une valeur donc $a_1 \Downarrow \vee a_2 \Downarrow$. On en déduit, par E-BRACKET, que $pc, \Gamma \vdash e' : s\ [r]$.

(handle) On pose $e = \mathsf{try}_\varepsilon\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ f$ et $e' = f[x \Leftarrow v]$. (Δ) se termine par une instance de E-TRY de prémisses $pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ [\varepsilon.\mathsf{Pre}^\ell s'; r_1']$ et $pc \sqcup \ell, \Gamma[x \mapsto s'] \vdash f : s\ [r_2]$ avec $r_2 \leq r$.
On déduit de $pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ [\varepsilon.\mathsf{Pre}^\ell s'; r_1']$ que $\Gamma \vdash v : s'$. On a donc $pc, \Gamma \vdash f[x \Leftarrow v] : s\ [r_2]$ (lemmes 4.7 et 4.2) puis $pc, \Gamma \vdash e' : s\ [r]$.

(handle-all) On pose $e = \mathsf{tryall}\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ f$ et $e' = f[x \Leftarrow (\mathsf{exn}_\varepsilon\ v)]$. (Δ) se termine par une instance de E-TRYALL de prémisses $pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ [r_1]$ et $pc \sqcup \mu(r_1), \Gamma[x \mapsto (r_1\ \mathsf{exn})^\ell] \vdash f : s\ [r]$.
On déduit de $pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ [r_1]$ que $\Gamma \vdash v : s'$ avec $r_1(\varepsilon) = \mathsf{Pre}^* s'$. On a donc, par V-EXN, que $\Gamma \vdash \mathsf{exn}_\varepsilon\ v : (r_1\ \mathsf{exn})^\ell$. On obtient alors $pc, \Gamma \vdash f[x \Leftarrow \mathsf{exn}_\varepsilon\ v] : s\ [r]$ (lemmes 4.7 et 4.2).

(lift-app) On pose $e = \langle v_1 \mid v_2 \rangle\, v$. (Δ) se termine par une instance de E-APP de prémisses $\Gamma \vdash \langle v_1 \mid v_2 \rangle : (s' \xrightarrow{pc \sqcup \ell\ [r]} s)^\ell$ et $\Gamma \vdash v : s'$ avec $\ell \lhd s$. On a $\Gamma \vdash v_i : (s' \xrightarrow{pc \sqcup \ell\ [r]} s)^\ell$ et $\ell \in L$ (lemme 4.6) ainsi que $\Gamma \vdash \lfloor v \rfloor_i : s'$ ($i \in \{1, 2\}$, lemme 4.4).
En appliquant E-APP, il vient $pc \sqcup \ell, \Gamma \vdash v_i\ \lfloor v \rfloor_i : s\ [r]$. Par E-BRACKET, on obtient $pc, \Gamma \vdash e' : s\ [r]$.

(lift-assign) On pose $e = \langle v_1 \mid v_2 \rangle := v$ et $e' = \langle v_1 := \lfloor\!\lfloor v \rfloor\!\rfloor_1 \mid v_2 := \lfloor\!\lfloor v \rfloor\!\rfloor_2 \rangle$. (Δ) se termine par une instance de E-ASSIGN de prémisses $\Gamma \vdash \langle v_1 \mid v_2 \rangle : (s' \ \mathsf{ref})^\ell$ et $\Gamma \vdash v : s'$ avec $pc \sqcup \ell \lhd s'$ et $s = \mathsf{unit}^\ell$. On a $\Gamma \vdash v_i : (s' \ \mathsf{ref})^\ell$ et $\ell \in L$ (lemme 4.6) ainsi que $\Gamma \vdash \lfloor\!\lfloor v \rfloor\!\rfloor_i : s'$ ($i \in \{1, 2\}$, lemme 4.4).

Par E-ASSIGN, on obtient $pc \sqcup \ell, \Gamma \vdash v_i := \lfloor\!\lfloor v \rfloor\!\rfloor_i : \mathsf{unit}^\ell \ [r]$ puis, par E-BRACKET, $pc, \Gamma \vdash e' : \mathsf{unit}^\ell \ [r]$.

(lift-deref) On pose $e = !\,\langle v_1 \mid v_2 \rangle$ et $e' = \langle !\, v_1 \mid !\, v_2 \rangle$. (Δ) se termine par une instance de la règle E-DEREF de prémisse $\Gamma \vdash \langle v_1 \mid v_2 \rangle : (s' \ \mathsf{ref})^\ell$ avec $s' \leq s$ et $\ell \lhd s$. On a $\Gamma \vdash v_i : (s' \ \mathsf{ref})^\ell$ et $\ell \in L$ (lemme 4.6).

Par E-DEREF, on obtient $pc \sqcup \ell, \Gamma \vdash !\, v_i : s \ [r]$ puis, par E-BRACKET, $pc, \Gamma \vdash \langle !\, v_1 \mid !\, v_2 \rangle : s \ [r]$.

(lift-raise) On pose $e = \mathsf{raise}\ \langle v_1 \mid v_2 \rangle$ et $e' = \langle \mathsf{raise}\ v_1 \mid \mathsf{raise}\ v_2 \rangle$. (Δ) se termine par une instance de E-RAISE de prémisse $\Gamma \vdash \langle v_1 \mid v_2 \rangle : (r \ \mathsf{exn})^\ell$ avec $pc \sqcup \ell \lhd r$. On a $\Gamma \vdash v_i : (r' \ \mathsf{exn})^\ell$ et $\ell \in L$ (lemme 4.6).

Par E-RAISE, on obtient $pc \sqcup \ell, \Gamma \vdash \mathsf{raise}\ v_i : s \ [r]$ puis, par E-BRACKET, $pc, \Gamma \vdash e' : s \ [r]$.

(lift-context) On pose $e = E[\langle a_1 \mid a_2 \rangle]$ et $e' = \langle \lfloor\!\lfloor E \rfloor\!\rfloor_1 [a_1] \mid \lfloor\!\lfloor E \rfloor\!\rfloor_2 [a_2] \rangle$.

Dans le cas où $E = \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e_0$, (Δ) se termine par une instance de E-BIND de prémisses $pc, \Gamma \vdash \langle a_1 \mid a_2 \rangle : s' \ [r_1]$ et $pc \sqcup \mu(r_1), \Gamma[x \mapsto s'] \vdash e_0 : s \ [r_2]$.

On déduit de $pc, \Gamma \vdash \langle a_1 \mid a_2 \rangle : s' \ [r_1]$ que $pc \sqcup pc', \Gamma \vdash a_i : s' \ [r_1]$ avec $pc' \in L$. On a alors, a_1 et a_2 n'étant pas deux valeurs, $\mu(r_1) \geq pc'$ donc $(pc \sqcup pc') \sqcup \mu(r_1), \Gamma[x \mapsto s'] \vdash e_0 : s \ [r_2]$. En appliquant E-BIND puis E-BRACKET (on a $\mathsf{bind}\ x = a_i\ \mathsf{in}\ \lfloor\!\lfloor e_0 \rfloor\!\rfloor_i \Downarrow$ pour $i = 1$ ou 2) on obtient $pc, \Gamma \vdash e : s \ [r]$.

Dans le cas où $E = \mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e_0$, (Δ) se termine par une instance de E-TRYALL de prémisses $pc, \Gamma \vdash \langle a_1 \mid a_2 \rangle : s \ [r_1]$, $pc \sqcup \mu(r_1), \Gamma_x \vdash e_0 : s \ [r]$ avec $\mu(r_1) \lhd s$.

On déduit de $pc, \Gamma \vdash \langle a_1 \mid a_2 \rangle : s \ [r_1]$ que $pc \sqcup pc', \Gamma \vdash a_i : s \ [r_1]$ ($i \in \{1, 2\}$) avec $pc' \in L$. On a alors $\mu(r_1) \geq pc'$ donc $(pc \sqcup pc') \sqcup \mu(r_1), \Gamma_x \vdash e_0 : s \ [r]$. En appliquant E-TRYALL et E-BRACKET, on obtient $pc, \Gamma \vdash e' : s \ [r]$.

Le cas $E = \mathsf{try}_\varepsilon\ [\,]\ \mathsf{with}\ x \succ e_0$ est analogue.

(bracket) On pose $e = \langle e_1 \mid e_2 \rangle$ et $e' = \langle e_1' \mid e_2' \rangle$ avec $e_i \,/\, \sigma \xrightarrow{i} e_i' \,/\, \sigma'$ et $e_j = e_j'$. (Δ) se termine par une instance de E-BRACKET de prémisses $pc \sqcup pc', \Gamma \vdash e_i : s \ [r]$ et $pc \sqcup pc', \Gamma \vdash e_j : s \ [r]$ avec $(pc' \lhd s) \vee e_1 \Downarrow \vee e_2 \Downarrow$ et $pc' \in L$.

On a $e_i \,/\, \sigma \xrightarrow{i} e_i' \,/\, \sigma'$ et, l'ensemble L étant clos, $pc \sqcup pc' \in L$. Il existe donc Γ' étendant Γ et tel que $pc \sqcup pc', \Gamma' \vdash e_i' : s \ [r]$ et $\Gamma' \vdash \sigma'$. De plus si $e_i \Downarrow$ alors $e_i' \Downarrow$ donc $(pc' \lhd s) \vee e_1' \Downarrow \vee e_2' \Downarrow$. On a également $pc \sqcup pc', \Gamma' \vdash e_j : s \ [r]$ (lemme 4.3). On en déduit, par la règle E-BRACKET, que $pc, \Gamma' \vdash e' : s \ [r]$.

(contexte) On pose $e = E[f]$ et $e' = E[f']$ avec $f \,/\, \sigma \xrightarrow{i} f' \,/\, \sigma'$.

Envisageons le cas où $E = \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e_0$. (Δ) se termine par une instance de E-BIND dont les prémisses sont $pc, \Gamma \vdash f : s' \ [r_1]$ et $pc \sqcup \mu(r_1), \Gamma[x \mapsto s'] \vdash e_0 : s \ [r_2]$ avec $r_1 \sqcup r_2 \leq r$.

Par hypothèse d'induction, on sait qu'il existe Γ' étendant Γ tel que $pc, \Gamma' \vdash f' \,/\, \sigma' : s' \ [r_1]$. On a également $pc, \Gamma'[x \mapsto s'] \vdash e_0 : s \ [r_2]$ (lemme 4.3). On en déduit que $pc, \Gamma' \vdash e' \,/\, \sigma' : s \ [r]$

Les autres cas de contextes se traitent d'une manière analogue. □

5 Non-interférence

5.1 Un premier résultat

Nous commençons par énoncer un théorème de non-interférence pour le calcul partagé. Ce résultat est une conséquence immédiate du théorème de *subject-reduction*.

Théorème 5.1 (Non-interférence pour le calcul partagé) *Soit L un ensemble d'étiquettes clos. Soit e une expression du calcul partagé telle que $pc, \emptyset \vdash_L e : \mathsf{int}^\ell \ [r]$ avec $\ell \notin L$. Si $e \,/\, \emptyset \rightarrow^* v$ alors $\lfloor\!\lfloor e \rfloor\!\rfloor_1 \,/\, \emptyset \rightarrow^* v$ et $\lfloor\!\lfloor e \rfloor\!\rfloor_2 \,/\, \emptyset \rightarrow^* v$.*

Démonstration. On a $pc, \emptyset \vdash_L e : \mathsf{int}^\ell \ [r]$ et $e \,/\, \emptyset \rightarrow^* v \,/\, \sigma$. Il existe donc un environnement Γ tel que $pc, \Gamma \vdash_L v : \mathsf{int}^\ell \ [r]$ (théorème 4.1). e est typable dans l'environnement vide donc e est close. v est donc également close. On en déduit que v est de l'une des deux formes suivantes : k et $\langle v_1 \mid v_2 \rangle$.

Or $\ell \notin L$ et L est clos. On en déduit que v ne peut être de la forme $\langle v_1 \mid v_2 \rangle$ et donc que $v = k$. Par le lemme 3.3 on obtient que $\lfloor\!\lfloor e \rfloor\!\rfloor_i \,/\, \emptyset \rightarrow^* v$. □

Nous nous contentons d'énoncer la propriété de non-interférence pour des expressions de type int. Ceci est en effet suffisant : on peut étendre le résultat facilement à toutes les formes de valeurs, par équivalence observationelle.

5.2 Typage du langage étiqueté

Pour énoncer un résultat sur le langage étiqueté, nous devons être en mesure de typer les expressions de ce langage. Nous n'allons cependant pas définir une nouvelle relation de zéro. De même que l'on obtient le calcul étiqueté en remplaçant les crochets par les étiquettes, de même il suffit en effet de remplacer dans SB(S) les règles v-Bracket et e-Bracket par les règles v-Label et e-Label de la figure 6.

$$\text{v-Label}\quad \frac{\Gamma \vdash v : s \qquad \ell \lhd s}{\Gamma \vdash (\ell : v) : s} \qquad\qquad \text{e-Label}\quad \frac{pc \sqcup \ell, \Gamma \vdash e : s\ [r] \qquad \ell \lhd s}{pc, \Gamma \vdash (\ell : e) : s\ [r]}$$

Fig. 6: Règles v-Label et e-Label

Notons que l'ensemble L n'est plus présent sur les jugements : il n'est pas utile d'indiquer les étiquettes secrètes représentées par les crochets puisque les étiquettes sont maintenant directement dans les termes. Ce point est important dans la mesure où il n'est pas nécessaire de fixer l'ensemble des étiquettes secrètes pour pouvoir typer un terme étiqueté : on peut donc typer un terme étiqueté « une fois pour toutes » et non pour chaque ensemble L possible.
Nous n'allons pas montrer de propriété de *subject reduction* sur la relation ainsi définie : il n'a pas été introduit de notion de réduction sur le langage étiqueté. Cependant, cette relation nous permet de typer les termes étiquetés qui sont ceux qui nous intéressent effectivement et d'assurer ensuite que les termes correspondant dans le calcul partagé auront les même types. Nous pourrons ainsi obtenir simplement le résultat de non-interférence pour ML_{re} étiqueté.

Définition 5.1 *Soient v_1 et v_2 deux valeurs closes du calcul étiqueté. On dit que v_2 a au moins les mêmes types que v_1 si et seulement si pour tout $s \in \mathcal{S}$ tel que $\emptyset \vdash v_1 : s$ alors $\emptyset \vdash v_2 : s$.*

Définition 5.2 ($\lessdot$) *Soit L un ensemble d'étiquettes. On dit que e_1* interfère moins que e_2 pour *L et on note $e_1 \lessdot_L e_2$ si on peut obtenir e_1 à partir de e_2 en remplaçant des sous-expressions de e_1 situées immédiatement sous des étiquettes de L et qui sont des valeurs closes par d'autres valeurs closes admettant au moins les mêmes types.*

On a choisi de limiter la substitution aux valeurs closes pour pouvoir donner un énoncé simple : il est en effet nécessaire de considérer un (ou plusieurs) environnement de typage pour pouvoir comparer les types de deux expressions. Cette limitation ne réduit cependant pas la portée des théorèmes qui suivent dans la mesure où on peut se ramener au cas des valeurs closes en ajoutant des λ-abstractions qui permettent en fait, au niveau du typage, de formaliser l'environnement. Par exemple, pour comparer les deux expressions

$$e_1 = H : (r := 1) \qquad\qquad e_2 = H : ()$$

il suffit en fait de considérer

$$e_1' = (H : \lambda x.(x := 1))\, r \qquad\qquad e_2' = (H : \lambda x.())\, r$$

Lemme 5.1 *Soient e_1 et e_2 deux expressions du calcul étiqueté telles que $e_1 \lessdot_L e_2$ et $pc, \Gamma \vdash e_1 : s\ [r]$. On peut construire une expression $e = [\![e_1 \mid e_2]\!]_L$ du calcul partagé telle que $\lfloor\!\lfloor e \rfloor\!\rfloor_i = \text{strip}(e_i)$ et $pc, \Gamma \vdash_L e : s\ [r]$.*

5.3 Théorème de non-interférence

Théorème 5.2 (Non-interférence pour le langage étiqueté) *Soit L un ensemble d'étiquettes clos. Soient e_1 et e_2 deux expressions du calcul étiqueté telles que $e_1 \lessdot_L e_2$ et $pc, \emptyset \vdash e_1 : \mathsf{int}^\ell\ [r]$ avec $\ell \notin L$.*
Si $\text{strip}\, e_i \,/\, \emptyset \rightarrow^* v_i$ (pour $i \in \{1, 2\}$) alors $v_1 = v_2$.

> **Démonstration.** On déduit de $e_1 \lessdot_L e_2$ et $pc, \emptyset \vdash e_1 : \mathsf{int}^\ell\ [r]$ qu'il existe une expression $e = [\![e_1 \mid e_2]\!]_L$ du calcul partagé telle que $\lfloor\!\lfloor e \rfloor\!\rfloor_i = \text{strip}(e_i)$ et $pc, \Gamma \vdash_L e : s\ [r]$ (lemme 5.1).
> On déduit de $\text{strip}\, e_i/\emptyset \rightarrow^* v_i$ et $\lfloor\!\lfloor e \rfloor\!\rfloor_i = \text{strip}\, e_i$ qu'il existe une valeur v telle que $e/\emptyset \rightarrow^* v$ et $v_i = \phi_i(\lfloor\!\lfloor v \rfloor\!\rfloor_i)$ (pour $i \in \{1, 2\}$, lemme 3.4). Par le théorème 5.1, on a $\phi_1(\lfloor\!\lfloor v \rfloor\!\rfloor_1) = \phi_2(\lfloor\!\lfloor v \rfloor\!\rfloor_2) = k$. On en déduit que $v_1 = v_2 = k$. $\square$

Cet énoncé appelle quelques commentaires. Il signifie intuitivement qu'étant donnée une expression e ayant pour type int^ℓ où ℓ est une étiquette non-secrète, la valeur produite par e ne dépend pas des sous-expressions situées sous des étiquettes secrètes puisque l'on peut remplacer ces dernières sans changer le résultat final. Il y a cependant deux limitations par rapport au résultat donné dans [PC00] :

- La terminaison n'est plus assurée : on est obligé de supposer que e_2 se réduit en une valeur pour obtenir le résultat.
- On impose non seulement que les expressions e_1 et e_2 aient même partie publique, mais aussi en fait qu'elles partagent la même dérivation de typage jusqu'aux étiquettes secrètes. Cette contrainte est exprimée par le fait que les sous-expressions substituées doivent être en relation pour $\lessdot_L$.

Ces restrictions sont directement liées à l'ajout de traits impératifs au langage. Si on veut exprimer un résultat de non-interférence dans lequel toute sous-expression située sous une étiquette secrète peut être remplacée par n'importe quelle autre expression, on doit être très restrictif – et donc très grossier – dans le typage. On devrait par exemple considérer le résultat de $\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\ H : ();!\, r$ comme secret puisque l'on pourrait comparer cette expression à $\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\ H : (r := 1);!\, r$... Un tel système présenterait peu d'intérêt dans la mesure où les types inférés monteraient très rapidement au niveau maximal : par exemple, dès lors qu'une sous-expression susceptible d'être réduite est située sous une étiquette secrète alors toutes les références pouvant être lues après doivent être rendues secrètes.

Notre système fait le choix d'une analyse plus fine, au prix de cette limitation du résultat. Celle-ci nous paraît cependant justifiée dans la mesure où on peut toujours :

- Remplacer une valeur brute (i.e. ne comportant pas de λ-abstraction ni de variable libre) telle qu'un entier par une autre valeur brute de même forme.
- Remplacer une expression effectuant des effets de bord (écriture dans le store, levée d'une exception) par une expression n'effectuant pas d'effet de bord (e.g. la constante $()$).

6 Extensions

Nous proposons quelques extensions de notre langage de base, $\mathrm{ML_{re}}$, et par là même du calcul partagé. Elles peuvent être de deux natures : on peut souhaiter ajouter de nouvelles constructions au langage soit pour accroître son expressivité (c'est le cas des sommes ou des paires par exemple) soit pour raffiner le typage de certaines constructions usuelles (comme par exemple le *try-finally* ou le *re-raise*).
Nous verrons que la démarche que nous avons suivie jusqu'à présent pour obtenir le résultat de non-interférence se prête particulièrement bien à ce type d'extensions : après avoir donné la syntaxe des nouvelles constructions ainsi que les règles de réduction et de typage correspondantes, il suffit principalement de prouver *subject reduction* pour les règles de réduction supplémentaires pour conclure.

6.1 Try-finally et re-raise

Nous ajoutons deux nouvelles formes de constructions « try ». Notre but n'est pas d'accroître la richesse sémantique de notre langage. Ces nouvelles formes d'expressions permettent d'obtenir un typage plus fin dans certains cas courants.

$\mathsf{try}\ e_1\ \mathsf{finally}\ e_2$: la réduction de cette construction consiste à évaluer e_1 en un résultat a_1 (valeur ou exception), puis dans les deux cas à évaluer e_2. Si e_2 lève une exception, elle est propagée ; sinon on renvoie a_1. D'un point de vue purement sémantique, cette structure n'est qu'une abréviation de

$$\mathsf{bind}\ x = (\mathsf{tryall}\ e_1\ \mathsf{with}\ y \succ e_2;\ \mathsf{raise}\ y)\ \mathsf{in}\ e_2;\ x$$

Cependant le typage de cette traduction n'est pas suffisamment fin : la règle induite obtenue est

$$\frac{pc, \Gamma \vdash e_1 : s\ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma \vdash e_2 : *\ [r_2] \qquad \mu(r_2) \lhd r_1}{pc, \Gamma \vdash \mathsf{try}\ e_1\ \mathsf{finally}\ e_2 : s\ [r_1 \sqcup r_2]}$$

L'expression e_2 doit être typée avec un pc augmenté de $\mu(r_1)$. Cette augmentation est inutile : e_2 est *toujours* réduite, que e_1 déclenche une exception ou non. Cette pollution est une conséquence de la duplication du code de e_2 dans la traduction : chaque copie est typée séparément sans tenir compte du fait que e_2 est toujours exécutée.

$$\frac{pc, \Gamma \vdash e_1 : s\ [r_1] \qquad pc, \Gamma \vdash e_2 : *\ [r_2] \qquad \mu(r_2) \lhd r_1}{pc, \Gamma \vdash \mathsf{try}\ e_1\ \mathsf{finally}\ e_2 : s\ [r_1 \sqcup r_2]}\ \text{(E-TRYFINALLY)}$$

$$\frac{pc, \Gamma \vdash e_1 : s\ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma[x \mapsto (r_1\ \mathsf{exn})^*] \vdash e_2 : *\ [r_2]}{pc, \Gamma \vdash \mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ e_2\ \mathsf{reraise} : s\ [r_1 \sqcup r_2]}\ \text{(E-TRYRERAISE)}$$

Fig. 7: Règles E-TRYRERAISE et E-TRYFINALLY

$\mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ e_2\ \mathsf{reraise}$: cette expression est, sémantiquement, une abréviation de

$$\mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ (e_2;\ \mathsf{raise}\ x)$$

Une fois encore, la règle de typage induite par cette traduction n'est pas satisfaisante :

$$\frac{pc, \Gamma \vdash e_1 : s\ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma[x \mapsto (r_1\ \mathsf{exn})^*] \vdash e_2 : *\ [r_2] \qquad r_1 \leq r_1' \qquad \mu(r_1) \lhd r_1'}{pc, \Gamma \vdash \mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ e_2\ \mathsf{reraise} : s\ [r_1' \sqcup r_2]}$$

On aimerait pouvoir avoir $r_1 = r_1'$ mais ce n'est pas toujours possible à cause de la contrainte « parasite » $\mu(r_1) \lhd r_1'$.

Syntaxe et règles de réduction Nous étendons la syntaxe de notre langage et ajoutons deux règles de réduction.

$$E \ ::= \ldots \mid \mathsf{try}\ [\,]\ \mathsf{finally}\ e \mid \mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e\ \mathsf{reraise} \qquad \text{(contextes)}$$

$$\mathsf{try}\ [\,]\ \mathsf{finally}\ e \lll a \qquad \mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e\ \mathsf{reraise} \not\lll v \qquad \mathsf{tryall}\ [\,]\ \mathsf{with}\ x \succ e\ \mathsf{reraise} \lll \mathsf{raised}_\varepsilon\ v$$

$$\mathsf{try}\ a\ \mathsf{finally}\ e\ /\ \sigma \xrightarrow{i} \mathsf{bind}\ \_ = e\ \mathsf{in}\ a\ /\ \sigma \qquad \text{(try-finally)}$$

$$\mathsf{tryall}\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ e\ \mathsf{reraise}\ /\ \sigma \xrightarrow{i} \mathsf{bind}\ \_ = e[x \Leftarrow \mathsf{raised}_\varepsilon\ v]\ \mathsf{in}\ (\mathsf{raised}_\varepsilon\ v)\ /\ \sigma \qquad \text{(try-reraise)}$$

Dans ces règles _ dénote n'importe quelle variable libre n'apparaissant pas dans e.

Typage Les règles de la figure 7 permettent de typer finement les deux nouvelles constructions, en s'affranchissant des problèmes évoqués précédemment. Nous devons maintenant prouver la correction de ces règles. Il suffit pour cela d'ajouter les cas correspondant aux nouvelles règles de réduction à la preuve de *subject reduction* :

Démonstration. On conserve toutes les notations employées dans la preuve du théorème 4.1.

(try-finally) On pose $e = \mathsf{try}\ a\ \mathsf{finally}\ e_0$ et $e' = \mathsf{bind}\ \_ = e_0\ \mathsf{in}\ a$. (Δ) se termine par une instance de E-TRYFINALLY de prémisses $pc, \Gamma \vdash a : s\ [r_1]$ et $pc, \Gamma \vdash e_0 : *\ [r_2]$ avec $\mu(r_2) \lhd r_1$ et $r_1 \sqcup r_2 \leq r$.
On déduit de $pc, \Gamma \vdash a : s\ [r_1]$ et $\mu(r_2) \lhd r_1$ que $pc \sqcup \mu(r_2), \Gamma \vdash a : s\ [r_1]$. On obtient, par E-BIND, $pc, \Gamma \vdash e' : s\ [r]$.

(try-reraise) On pose $e = \mathsf{tryall}\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ e\ \mathsf{reraise}$ et $e' = \mathsf{bind}\ \_ = e\ \mathsf{in}\ (\mathsf{raised}_\varepsilon\ v)$. (Δ) se termine par une instance de E-TRYRERAISE de prémisses $pc, \Gamma \vdash \mathsf{raised}_\varepsilon\ v : s\ [r_1]$ et $pc \sqcup \mu(r_1), \Gamma[x \mapsto (r_1\ \mathsf{exn})^*] \vdash e : *\ [r_2]$ avec $r_1 \sqcup r_2 \leq r$. On en déduit que $pc \sqcup \mu(r_1), \Gamma \vdash e[x \Leftarrow \mathsf{raised}_\varepsilon\ v] : *\ [r_2]$ (lemme 4.7). Par E-BIND, on obtient $pc, \Gamma \vdash e' : s\ [r]$.

On vérifie également que la propriété reste vraie pour les règles (throw-context), (lift-context) et (context) qui ont été surchargées par l'ajout de nouveaux contextes. □

Le théorème 4.1 est ainsi toujours valide. On en déduit immédiatement que les théorèmes de non-interférence pour le calcul partagé (théorème 5.1) et pour le calcul étiqueté (théorème 5.2) restent vrais.

6.2 Sommes et paires

Nous ajoutons ici des sommes binaires et des paires à notre langage. Il est très aisé de généraliser notre approche au cas des sommes n-aires et des n-uplets.

Syntaxe et règles de réduction

$$v \ ::= \ldots \mid \mathsf{inj}_j\ v \mid (v,\ v) \qquad (j \in \{1,2\}) \qquad \text{(valeurs)}$$
$$e \ ::= \ldots \mid \mathsf{match}\ v\ \mathsf{with}\ v\ v \mid \mathsf{proj}_j\ v \qquad (j \in \{1,2\}) \qquad \text{(expressions)}$$

$$\mathsf{proj}_j\ (v_1,\ v_2)\ /\ \sigma \xrightarrow{i} v_j\ /\ \sigma \qquad \text{(proj)}$$
$$\mathsf{proj}_j\ \langle v_1 \mid v_2 \rangle\ /\ \sigma \xrightarrow{\varnothing} \langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2 \rangle \qquad \text{(lift-proj)}$$
$$\mathsf{match}\ (\mathsf{inj}_j\ v)\ \mathsf{with}\ v_1\ v_2\ /\ \sigma \xrightarrow{i} v_j\ v\ /\ \sigma \qquad \text{(match)}$$
$$\mathsf{match}\ \langle v_1 \mid v_2 \rangle\ \mathsf{with}\ w_1\ w_2\ /\ \sigma \xrightarrow{\varnothing} \langle \mathsf{match}\ v_i\ \mathsf{with}\ \lfloor\!\lfloor w_1 \rfloor\!\rfloor_i\ \lfloor\!\lfloor w_2 \rfloor\!\rfloor_i \rangle_i\ /\ \sigma \qquad \text{(lift-match)}$$

Typage Pour pouvoir écrire les règles de typage relatives aux sommes et aux paires, nous devons ajouter deux « constructeurs » à notre algèbre de types.

Axiome 6 (Constructions sur $\mathcal{S}$) L'ensemble $\mathcal{S}$ est clos pour les constructions suivantes :
- **Sommes :** Si $s_1, s_2 \in \mathcal{S}$ et $\ell \in \mathcal{L}$ alors $(s_1 + s_2)^\ell \in \mathcal{S}$.
- **Paires :** Si $s_1, s_2 \in \mathcal{S}$ et $\ell \in \mathcal{L}$ alors $(s_1 \times s_2)^\ell \in \mathcal{S}$.

On suppose également que ces constructions vérifient les relations usuelles de sous-typage.
On peut alors compléter notre système par les règles de la figure 8 (on a noté $(s_1 +^1 s_2)^\ell = (s_1 + s_2)^\ell$ et $(s_1 +^2 s_2)^\ell = (s_2 + s_1)^\ell$).

Pour prouver la correction de ce système étendu, il suffit encore une fois de considérer simplement de nouveaux cas dans la preuve de *subject-reduction*.

Démonstration. On conserve toutes les notations employées dans la preuve du théorème 4.1.

(proj) On pose $e = \mathsf{proj}_j\ (v_1,\ v_2)$ et $e' = v_j$. (Δ) se termine par une instance de E-PROJ dont une prémisse est $\Gamma \vdash (v_1,\ v_2) : (s_1 \times s_2)^\ell$ avec $s = s_j$. Il existe une dérivation simple de $\Gamma \vdash (v_1,\ v_2) : (s_1' \times s_2')^{\ell'}$ (avec $s_1' \leq s_1$, $s_2' \leq s_2$ et $\ell' \leq \ell$, lemme 4.1) qui se termine par une instance de V-PAIR de prémisse $\Gamma \vdash v_j : s_j'$. On en déduit, par V-SUB, que $\Gamma \vdash v_j : s$.

(lift-proj) On pose $e = \mathsf{proj}_j\ \langle v_1 \mid v_2 \rangle$ et $e' = \langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2 \rangle$. (Δ) se termine par une instance de E-PROJ de prémisses $\Gamma \vdash \langle v_1 \mid v_2 \rangle : (s_1 \times s_2)^\ell$ et $\ell \lhd s_j$ avec $s = s_j$. On a $\Gamma \vdash v_i : (s_1 \times s_2)^\ell$ (pour $i \in \{1,2\}$) et $\ell \in L$ (lemme 4.6). Par E-PROJ, on en déduit que $\ell \sqcup pc, \Gamma \vdash \mathsf{proj}_j\ v_i : s\ [r]$. Par E-BRACKET on obtient $pc, \Gamma \vdash e' : s\ [r]$.

(match) On pose $e = \mathsf{match}\ (\mathsf{inj}_j\ v)\ \mathsf{with}\ v_1\ v_2$ et $e' = v_j\ v$. (Δ) se termine par une instance de E-MATCH de prémisses $\Gamma \vdash v : (s_1 + s_2)^\ell$ et $\Gamma \vdash v_j : (s_j \xrightarrow{pc \sqcup \ell_j \sqcup \ell\ [r]} s)^{\ell_j}$ avec $\ell_j \lhd s$. On déduit du premier jugement que $\Gamma \vdash \mathsf{inj}_j\ v : s_j$. On obtient, par la règle E-APP, que $pc, \Gamma \vdash v_j\ v : s\ [r]$.

(lift-match) On pose $e = \mathsf{match}\ \langle v_1 \mid v_2 \rangle\ \mathsf{with}\ w_1\ w_2$ et $e' = \langle \mathsf{match}\ w_i\ \mathsf{with}\ \lfloor\!\lfloor v_1 \rfloor\!\rfloor_i\ \lfloor\!\lfloor v_2 \rfloor\!\rfloor_i \rangle_i$. (Δ) se termine par une instance de E-MATCH de prémisses $\Gamma \vdash \langle v_1 \mid v_2 \rangle : (s_1 + s_2)^\ell$ et $\Gamma \vdash w_j : (s_j \xrightarrow{pc \sqcup \ell_j \sqcup \ell\ [r]} s)^{\ell_j}$ (pour $j \in \{1,2\}$) avec $\ell_1 \sqcup \ell_2 \sqcup \ell \lhd s$. On en déduit que $\Gamma \vdash v_i : (s_1 + s_2)^\ell$ et $\ell \in L$ (lemme 4.6). Par E-MATCH, il vient $pc \sqcup \ell, \Gamma \vdash \mathsf{match}\ w_i\ \mathsf{with}\ \lfloor\!\lfloor v_1 \rfloor\!\rfloor_i\ \lfloor\!\lfloor v_2 \rfloor\!\rfloor_i : s\ [r]$. On en déduit, par E-BRACKET, que $pc, \Gamma \vdash e' : s\ [r]$. □

Retour sur les exceptions Le mécanisme d'exception peut être simulé dans un langage avec sommes : une réponse qui est une valeur est codée comme une injection *gauche* et une exception levée comme une injection *droite*. Si on se place dans le cas simple avec une seule sorte d'exception, les contextes bind et try peuvent être traduits à l'aide d'un match : bind $x = e_1$ in e_2 devient match e_1 with $(\lambda x.e_2)$ $(\lambda x.\mathsf{inj}_2\ x)$ et try e_1 with $x \succ e_2$ devient match e_1 with $(\lambda x.\mathsf{inj}_1\ x)$ $(\lambda x.e_2)$.
Nous aurions semble-t-il pu suivre une autre approche : partir d'un langage sans exceptions mais doté de sommes, prouver un résultat de non-interférence pour ce langage et en déduire un résultat similaire pour le

$$\frac{\Gamma \vdash v_1 : s_1 \qquad \Gamma \vdash v_2 : s_2}{\Gamma \vdash (v_1,\ v_2) : (s_1 \times s_2)^*}\ \textsc{v-Pair} \qquad \frac{\Gamma \vdash v : (s_1 \times s_2)^\ell \qquad \ell \lhd s_j}{pc, \Gamma \vdash \mathsf{proj}_j\ v : s_j\ \ [*]}\ \textsc{e-Proj} \qquad \frac{\Gamma \vdash v : s}{\Gamma \vdash \mathsf{inj}_j\ v : (s +^j *)^*}\ \textsc{v-Inj}$$

$$\frac{\Gamma \vdash v : (s_1 + s_2)^\ell \qquad \Gamma \vdash v_1 : (s_1 \xrightarrow{pc \sqcup \ell_1 \sqcup \ell\ [r]} s)^{\ell_1} \qquad \Gamma \vdash v_2 : (s_2 \xrightarrow{pc \sqcup \ell_2 \sqcup \ell\ [r]} s)^{\ell_2} \qquad \ell_1 \sqcup \ell_2 \sqcup \ell \lhd s}{pc, \Gamma \vdash \mathsf{match}\ v\ \mathsf{with}\ v_1\ v_2 : s\ \ [r]}\ \textsc{e-Match}$$

Fig. 8: Règles v-Pair, e-Proj, v-Inj et e-Match

langage avec exceptions en codant ce dernier à l'aide de sommes. Cependant, cette technique ne donne pas un résultat satisfaisant. Les règles de typage obtenues ne sont pas suffisamment fines. La règle obtenue pour la construction bind est ainsi la suivante :

$$\frac{pc, \Gamma \vdash e_1 : (s_1 + s')^{\ell_1} \qquad pc \sqcup \ell_1, \Gamma[x \mapsto s_1] \vdash e_2 : (s_2 + s')^{\ell_2}}{pc, \Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : (s_2 + s')^{\ell_1 \sqcup \ell_2}}$$

Le niveau du résultat est en effet pollué par le niveau associé à l'expression e_1 même si x n'est pas utilisé pour le calcul de e_2 et aucune exception n'est levée.

6.3 Primitives

Les résultats que nous avons énoncés jusqu'à présent ne permettent pas de donner des réponses satisfaisantes à propos d'un certain nombre de primitives disponibles dans les langages réels, telles que les opérations arithmétiques ou les opérations de comparaison (éventuellement polymorphes comme en Caml). Nous allons proposer dans cette section une réponse générale à ce problème en dotant notre langage de primitives unaires (ce qui traite également les opérations n-aires, grâce aux paires).

Définition 6.1 (Valeur brute) *Une valeur de ML_{re} (avec éventuellement sommes et paires) est* brute *si elle ne comporte ni λ-abstraction ni variable libre. Les valeurs brutes sont donc définies par la grammaire suivante :*

$$\bar{v} \ ::= k \mid () \mid \mathsf{exn}_\varepsilon\ \bar{v} \mid \mathsf{inj}_\varepsilon\ \bar{v} \mid (\bar{v},\ \bar{v}) \qquad \text{(valeurs brutes)}$$

Extension syntaxique de $\mathbf{ML_{re}}$ Supposons que l'on munit ML_{re} d'un certain nombre de primitives f appartenant à un ensemble $\mathcal{F}$ en ajoutant une forme d'expression :

$$e \ ::= \ldots \mid f\, e \qquad \text{(expressions)}$$

À chaque $f \in \mathcal{F}$ est associée une application partielle $[f]$ des valeurs brutes dans les réponses closes. Ces interprétations donnent les δ-règles de ML_{re} qui correspondent aux primitives :

$$f\,\bar{v} \,/\, \sigma \to [f](\bar{v}) \,/\, \sigma \qquad (\text{si } \bar{v} \in \mathrm{dom}(f))$$

La forme de ces δ-règles sous-entend que les primitives f n'effectuent pas d'effet de bord *via* l'état mémoire, ni en lecture, ni en écriture.

Extension syntaxique du calcul partagé Dans le calcul partagé, une valeur brute peut comporter des crochets. Nous ajoutons donc deux règles de réduction au calcul partagé de manière à pouvoir réduire les primitives lorsqu'elles sont appliquées à des valeurs brutes comportant des crochets.

$$f\,\bar{v} \,/\, \sigma \xrightarrow{i} [f](\bar{v}) \,/\, \sigma \qquad \text{si } \bar{v} \text{ ne comporte pas de crochets} \qquad (\delta_f)$$
$$f\,\bar{v} \,/\, \sigma \xrightarrow{\varnothing} \langle [f](\lfloor \bar{v} \rfloor_1) \mid [f](\lfloor \bar{v} \rfloor_2) \rangle \,/\, \sigma \qquad \text{sinon} \qquad (\text{lift-}f)$$

On vérifie qu'ainsi étendu le calcul partagé simule toujours le calcul simple, i.e. que les lemmes 3.3 et 3.4 demeurent vrais.

Typage Nous allons montrer comment, si on sait typer les primitives dans le cas du calcul simple, alors on peut étendre ce typage au calcul partagé tout en conservant les théorèmes de *subject reduction* et de non-interférence. On suppose donné, pour chaque primitive f, un ensemble de types flèches $T(f)$ qui vérifie l'axiome suivant.

Axiome 7 Pour toute valeur brute simple $\bar{v} \in \mathrm{dom}(f)$, $\Gamma \vdash \bar{v} : s'$, si $(s' \xrightarrow{pc\ [r]} s)^\ell \in T(f)$ alors $pc, \Gamma \vdash [f](\bar{v}) : s\ [r]$.

Cet axiome nous assure que les primitives sont *bien typées* dans le calcul simple. Plutôt que de définir formellement une relation de typage spécifique aux valeurs brutes, il est plus simple d'utiliser les règles de SB(S) (figure 5), les annotations de sécurité étant alors de fait inutiles (puisque la règle V-BRACKET n'est pas utilisée).

Nous avons également besoin d'une relation particulière entre types et étiquettes $s \blacktriangleleft \ell$. Moralement, elle signifie que toute étiquette présente dans le type s est inférieure ou égale à ℓ. En raison de la façon dont l'ensemble $\mathcal{S}$ a été introduit, nous devons la spécifier par un axiome.

Axiome 8 La relation $\cdot \blacktriangleleft \cdot$ vérifie

$$\begin{array}{rl}
\mathsf{int}^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 & \mathsf{unit}^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \\
(r\ \mathsf{exn})^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \wedge (\forall \varepsilon \in \mathcal{E},\ r(\varepsilon) \blacktriangleleft \ell_0) & (s_1 + s_2)^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \wedge s_1 \blacktriangleleft \ell_0 \wedge s_2 \blacktriangleleft \ell_0 \\
(s_1 \times s_2)^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \wedge s_1 \blacktriangleleft \ell_0 \wedge s_2 \blacktriangleleft \ell_0 & s \leq s' \wedge s' \blacktriangleleft \ell_0 \Rightarrow s \blacktriangleleft \ell_0
\end{array}$$

Nous notons également $s \blacktriangleleft s_0$ si et seulement si il existe ℓ tel que $s \blacktriangleleft \ell$ et $\ell \lhd s_0$; ainsi que $s \blacktriangleleft r_0$ si et seulement si il existe ℓ tel que $s \blacktriangleleft \ell$ et $\ell \lhd r_0$. Nous pouvons adjoindre la règle suivante au système SB(S) pour typer les primitives :

E-PRIMITIVE
$$\frac{(s' \xrightarrow{pc\ [r]} s)^\ell \in T(f) \qquad \Gamma \vdash v : s' \qquad s' \blacktriangleleft s \qquad s' \blacktriangleleft r}{pc, \Gamma \vdash v : s\ [r]}$$

Cette règle est « pessimiste » dans le sens où elle impose que le type du résultat, s, soit pollué par toute étiquette apparaissant dans le type de l'argument s'. En effet, la fonction f n'étant pas connue, le résultat de l'application peut *a priori* dépendre de toute partie de l'argument.

Subject reduction et non-interférence

Lemme 6.1 *Soient $\bar{v}$ une valeur brute du calcul partagé telle que $\Gamma \vdash \bar{v} : s$. Si $\bar{v}$ comporte une sous-expression de la forme $\langle \ldots \mid \ldots \rangle$ et $s \blacktriangleleft \ell$ alors $\ell \in L$.*

Démonstration. On peut se ramener au cas où la dérivation de $\Gamma \vdash \bar{v} : s$. On procède par cas suivant la forme de la valeur v. Tous les cas sont analogues, on ne traite donc que le cas suivant :

Paire La dérivation de $\Gamma \vdash \bar{v} : s$ se termine par une instance de V-PAIR de prémisses $\Gamma \vdash v_1 : s_1$ et $\Gamma \vdash v_2 : s_2$ avec $s = (s_1 \times s_2)^{\ell'}$. On a $(s_1 \times s_2)^{\ell'} \blacktriangleleft \ell$ donc $s_1 \blacktriangleleft \ell$ et $s_2 \blacktriangleleft \ell$.
v comporte une sous-expression à crochets. On en déduit qu'il en est de même de v_1 ou de v_2. Par hypothèse d'induction, on en déduit que $\ell \in L$. □

On peut maintenant vérifier que la propriété de *subject reduction* s'étend aux deux nouvelles règles.

Démonstration. On conserve toutes les notations employées dans la preuve du théorème 4.1.

(δ_f) On pose $e = f\,v$ et $e' = [f](v)$. (Δ) se termine par une instance de E-PRIMITIVE de prémisses $(s' \xrightarrow{pc\ [r]} s)^\ell \in T(f)$ et $\Gamma \vdash v : s'$. Par l'axiome 7, on en déduit que $pc, \Gamma \vdash [f](v) : s\ [r]$.

(δ-lift$_f$) On pose $e = f\,v$ et $e' = \langle [f](\lfloor \bar{v} \rfloor_1) \mid [f](\lfloor \bar{v} \rfloor_2) \rangle$. ($\Delta$) se termine par une instance de E-PRIMITIVE de prémisses $(s' \xrightarrow{pc\ [r]} s)^\ell \in T(f)$, $\Gamma \vdash v : s'$, $s' \blacktriangleleft s$ et $s' \blacktriangleleft r$. On déduit de $\Gamma \vdash v : s'$ que $\Gamma \vdash \lfloor v \rfloor_i : s'$ (pour $i \in \{1, 2\}$, lemme 4.4). On obtient $pc, \Gamma \vdash [f](\lfloor v \rfloor_i) : s\ [r]$ (règle E-PRIMITIVE).
Or v comporte une sous-expression à crochets donc il existe $\ell \in L$ tel que $\ell \lhd s$ et $\ell \lhd r$ (lemme 6.1). $\lfloor v \rfloor_i$ étant une réponse, on en déduit que $pc \sqcup \ell, \Gamma \vdash [f](\lfloor v \rfloor_i) : s\ [r]$. En appliquant E-BRACKET, on en déduit que $pc, \Gamma \vdash e' : s\ [r]$. □

Exemples d'application Ce résultat permet d'obtenir facilement des règles de typage pour les opérations arithmétiques et de comparaison usuelles :

$$\frac{\Gamma \vdash v_1 : \mathsf{int}^\ell \qquad \Gamma \vdash v_2 : \mathsf{int}^\ell}{\Gamma \vdash v_1 \bullet v_2 : \mathsf{int}^\ell} \ \bullet \in \{+, -, \times, /, \ldots\} \qquad \frac{\Gamma \vdash v_1 : s \qquad \Gamma \vdash v_2 : s \qquad s \blacktriangleleft \ell}{\Gamma \vdash v_1 \bullet v_2 : \mathsf{int}^\ell} \ \bullet \in \{=, \leq, \geq, \ldots\}$$

Les relations de comparaison indiquées ci-dessus ne sont pas exactement celles de *Caml* dans la mesure où dans ce langage il est possible de comparer des valeurs contenant des adresses mémoires (et ce sont alors, pour l´égalité structurelle, les contenus des références qui sont comparés). Pour traiter exhaustivement ces relations, il faudrait ajouter m comme forme de valeur brute en *entrée* pour les primitives et autoriser les fonctions $[f]$ à lire la partie du store accessible à partir de la valeur passée en argument.

6.4 Vers une extension systématique

Notre première analyse sur le « noyau » $\mathrm{ML_{re}}$ et les différentes extensions que nous venons d'étudier nous suggèrent de proposer un mécanisme général permettant, à partir de la syntaxe et des règles de réduction d'un calcul simple, d'obtenir les règles pour le calcul partagé correspondant. Les propriétés que l'on attend pour ce dernier sont, bien entendu, la correction (au sens du lemme 3.3) et la complétude (lemme 3.4). Une telle présentation poserait cependant quelques problèmes formels, dans le sens où elle nécessiterait de définir précisément comment la syntaxe et la sémantique du calcul de base sont données.

Nous nous limiterons de ce fait à quelques idées générales. Si on choisit comme nous l'avons fait une syntaxe inspirée des *formes A-normales* (ce qui n'est *a priori* pas limitatif), le langage comporte deux types de constructions : des contextes et des redex simples. Le cas des contextes est probablement le plus aisé. Les règles relatives à ces derniers effectuent en effet une sorte de *pattern matching* sur les différentes formes de réponses. Il suffit donc d'étendre ce dernier aux nouvelles formes de réponses, à l'aide des règles (throw-context) et (lift-context).
Les redex peuvent quant-à-eux être de formes plus variées. Une forme usuelle de redex est $\mathsf{op}\ v$. Nous avons vu que dans ce cas, en plus de la règle habituelle de réduction du redex, il convient d'ajouter une règle (lift-op) de la forme

$$\mathsf{op}\ \langle v_1 \mid v_2 \rangle \ / \ \sigma \xrightarrow{\varnothing} \langle \mathsf{op}\ v_1 \mid \mathsf{op}\ v_2 \rangle$$

pour départager l'opérateur lorsqu'il est appliqué à deux valeurs différentes. Ces opérateurs sont des *destructeurs*, en opposition au *constructeurs*, comme $\mathsf{exn}_\varepsilon\ \_$, qui permettent de constituer de nouvelles formes de réponse. Ceci explique le choix que nous avons fait de distinguer la construction $\mathsf{raise}\ (\mathsf{exn}_\varepsilon\ v)$ de $\mathsf{raised}_\varepsilon\ v$ afin de bien donner son rôle de destructeur à raise et d'obtenir la « bonne » règle *lift*.

Chapitre 3

Système SHM(S) et inférence de types

Le système SB(S) que nous avons présenté dans la partie 2 nous a permis de proposer une preuve relativement simple de la propriété de non-interférence. Cependant, ce système ne peut être la base d'un algorithme d'inférence : pour montrer qu'une valeur admet un certain polytype p, il peut être nécessaire de considérer une infinité de jugements. Nous présentons dans cette partie le système SHM(S) qui est dérivé pour sa forme du système général HM(X) [SMZ99] et montrons comment on peut prouver sa correction à partir des résultats obtenu pour SB(S).

7 Le système SHM(S)

7.1 Algèbre des types et contraintes

Le système SHM(S) est basé sur une logique du premier ordre dont les termes dénotent les éléments de types (i.e. types annotés, alternatives, rangées, niveaux) et les formules représentent les contraintes. Les termes et les formules sont définis par les grammaires de la figure 9. On distingue quatre sortes de termes : les termes de types (notés ς), les termes d'alternatives (θ), les termes de rangées (ρ) et les termes d'étiquettes (λ).
On introduit un système de sortes sur les termes afin de s'assurer que les rangées sont bien formées. On note $\vdash \varsigma$ (resp. $\vdash \theta$) pour indiquer qu'un terme de type (resp. d'alternative) est bien formé. On note $\vdash \rho : X$ pour indiquer qu'un terme de rangée est bien formé et a pour domaine l'ensemble de noms d'exceptions X. Les termes d'étiquettes sont tous bien formés. On note enfin $\vdash C$ pour indiquer qu'une contrainte C est bien formée. Ces jugements sont définis par les règles de la figure 10. On ne considère plus par la suite que des termes et des contraintes bien formés.

Une affectation ϕ est la réunion de quatre applications : une application des termes de types dans $\mathcal{S}$, une application des termes d'alternatives dans $\mathcal{A}$, une application des termes d'étiquettes dans $\mathcal{L}$ et enfin une application des termes de rangées de sorte X dans $(X \to \mathcal{A})$. On étend systématiquement une affectation aux termes par homomorphisme.
Notre logique doit être équipée d'une interprétation, c'est-à-dire d'un prédicat binaire $\cdot \vdash \cdot$ dont le premier argument est une affectation ϕ des variables et le second argument une contrainte C. Cette interprétation doit vérifier les lois suivantes :

$$
\begin{array}{rcl}
\phi \vdash \varsigma_1 \leq \varsigma_2 \Leftrightarrow \phi(\varsigma_1) \leq \phi(\varsigma_2) & \quad & \phi \vdash \lambda \lhd \varsigma \Leftrightarrow \phi(\lambda) \lhd \phi(\varsigma) \\
\phi \vdash \theta_1 \leq \theta_2 \Leftrightarrow \phi(\theta_1) \leq \phi(\theta_2) & & \phi \vdash \lambda \lhd \rho \Leftrightarrow \phi(\lambda) \lhd \phi(\rho) \\
\phi \vdash \rho_1 \leq \rho_2 \Leftrightarrow \phi(\rho_1) \leq \phi(\rho_2) & & \phi \vdash \mathbf{true} \\
\phi \vdash \lambda_1 \leq \lambda_2 \Leftrightarrow \phi(\lambda_1) \leq \phi(\lambda_2) & & \phi \vdash C_1 \wedge C_2 \Leftrightarrow \phi \vdash C_1 \wedge \phi \vdash C_2 \\
\phi \vdash \mu(\rho) \leq \lambda \Leftrightarrow \mu(\phi(\rho)) \leq \phi(\lambda) & & \phi \vdash \exists \bar{\alpha}.C \Leftrightarrow \exists \phi' \ (\phi' \backslash \bar{\alpha} = \phi \backslash \bar{\alpha}) \wedge \phi' \vdash C
\end{array}
$$

Termes :

$$\begin{array}{rll} \tau & ::= \varsigma \mid \theta \mid \rho \mid \lambda & \text{(termes)} \\ \varsigma & ::= \alpha_{\mathcal{S}} \mid \mathsf{int}^{\lambda} \mid \mathsf{unit}^{\lambda} \mid (\varsigma \xrightarrow{\pi\ [\rho]} \varsigma)^{\lambda} \mid (\varsigma\ \mathsf{ref})^{\lambda} \mid (\rho\ \mathsf{exn})^{\lambda} & \text{(termes de types)} \\ \theta & ::= \alpha_{\mathcal{A}} \mid \mathsf{Pre}^{\lambda}\varsigma & \text{(termes d'alternatives)} \\ \rho & ::= \alpha_{\mathcal{R}[\mathrm{X}]} \mid \varepsilon.\theta;\rho & \text{(termes de rangées)} \\ \lambda,\pi & ::= \alpha_{\mathcal{L}} \mid \ell & \text{(termes de niveaux)} \end{array}$$

Formules (contraintes) :

$$\begin{array}{rll} C & ::= \mathbf{true} & \text{(vrai)} \\ & \mid \varsigma \leq \varsigma \mid \theta \leq \theta \mid \rho \leq \rho \mid \lambda \leq \lambda & \text{(sous-typage)} \\ & \mid \lambda \lhd \varsigma & \text{(garde)} \\ & \mid \lambda \lhd \rho & \text{(garde tous)} \\ & \mid \mu(\rho) \leq \lambda & \text{(mu)} \\ & \mid C \wedge C & \text{(conjonction)} \\ & \mid \exists \alpha.C & \text{(quantification)} \end{array}$$

Fig. 9: Termes et formules

Termes :

$$\vdash \alpha_{\mathcal{S}} \qquad \vdash \alpha_{\mathcal{A}} \qquad \vdash \alpha_{\mathcal{R}[\mathrm{X}]} : X \qquad \vdash \mathsf{int}^{\lambda} \qquad \vdash \mathsf{unit}^{\lambda}$$

$$\frac{\vdash \varsigma' \quad \vdash \rho : \mathcal{E} \quad \vdash \varsigma}{\vdash (\varsigma' \xrightarrow{\pi\ [\rho]} \varsigma)^{\lambda}} \qquad \frac{\vdash \varsigma}{\vdash (\varsigma\ \mathsf{ref})^{\lambda}} \qquad \frac{\vdash \rho : \mathcal{E}}{\vdash (\rho\ \mathsf{exn})^{\lambda}} \qquad \frac{\vdash \varsigma}{\vdash \mathsf{Pre}^{\lambda}\varsigma} \qquad \frac{\vdash \theta \quad \vdash \rho : X \quad \varepsilon \notin X}{\vdash \varepsilon.\theta;\rho : X \cup \{\varepsilon\}}$$

Contraintes :

$$\frac{\vdash \varsigma_1 \quad \vdash \varsigma_2}{\vdash \varsigma_1 \leq \varsigma_2} \qquad \frac{\vdash \theta_1 \quad \vdash \theta_2}{\vdash \theta_1 \leq \theta_2} \qquad \frac{\vdash \rho_1 : X \quad \vdash \rho_2 : X}{\vdash \rho_1 \leq \rho_2} \qquad \vdash \lambda_1 \leq \lambda_2 \qquad \frac{\vdash \varsigma}{\vdash \lambda \lhd \varsigma} \qquad \frac{\vdash \rho : X}{\vdash \lambda \lhd \rho}$$

$$\frac{\vdash \rho : X}{\vdash \mu(\rho) \leq \lambda} \qquad \frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1 \wedge C_2} \qquad \frac{\vdash C}{\vdash \exists \alpha. C}$$

Fig. 10: Règles de sortage

Nous écrivons $C \Vdash C'$ si et seulement si C implique C', c'est-à-dire si toute solution ϕ de C est une solution de C'.

Définition 7.1 (Schéma de type) *Un schéma de type σ est un triplet formé d'une liste finie $\bar{\alpha}$ de variables, d'une contrainte C et d'un terme de type ς. On écrit $\sigma = \forall\bar{\alpha}[C].\varsigma$. Les variables de $\bar{\alpha}$ sont* liées *dans σ. Les schémas sont considérés modulo α-conversion. Par abus de notation, une variable de type α seule pourra être vue comme le schéma $\forall\varnothing[\mathbf{true}].\alpha$.*

7.2 Jugements

Dans SHM(S), un environnement Γ est une application qui à une variable (ou un état mémoire) associe un schéma de type. Comme dans SB(S), les jugements sont de deux formes. La première forme est limitée aux valeurs : $\Gamma, C \vdash v : \varsigma$ où Γ est un environnement, C une contrainte, v une valeur et ς un terme de sorte $\mathcal{S}$. Les jugements portant sur les expressions sont quant à eux de la forme : $pc, \Gamma, C \vdash v : \varsigma\ [\rho]$ où pc est un terme de sorte $\mathcal{L}$ et ρ un terme de sorte $\mathcal{R}[\mathcal{E}]$. On sous-entend toujours, dans un jugement, que la contrainte C est satisfiable. Les jugements valides sont définis inductivement par les règles de la figure 11.
Dans ces règles, on s'autorise à noter $\tau_1 \sqcup \tau_2$ là où une variable libre α est attendue avec une prémisse supplémentaire $C \Vdash \tau_1 \leq \alpha_{\mathcal{L}} \wedge \tau_2 \leq \alpha$.
SHM(S) se distingue de SB(S) par l'introduction d'un niveau syntaxique : les monotypes ont été remplacés par des termes, les polytypes par des schémas. De plus, tous les jugements sont paramétrés par une contrainte qui représente une hypothèse sur les variables libres du jugement. Les schémas de types sont introduits par la règle HE-LET (qui comprend elle-même la généralisation) et instanciés directement par la règle HV-VAR. Ceci fait que nous n'avons pas besoin de règles de généralisation ou d'instanciation spécifiques. Ainsi, les jugements portent toujours sur des termes et non des schémas.

7.3 Correction et non-interférence

Nous allons montrer comment on peut prouver la correction de SHM(S) à partir de la correction du système SB(S). Pour cela il suffit de montrer qu'à chaque jugement dérivable dans SHM(S) correspond un jugement de SB(S). Nous ne détaillons pas la démonstration, des preuves analogues sont données précisément dans [Pot01] pour HM(X) et [CP01] pour JOIN(X), un système sur le Join-Calcul.

Sous une affectation des variables donnée, un schéma de type représente en fait l'ensemble des types *bruts* qui sont ses instances, c'est-à-dire en fait un *polytype* de SB(S).

Définition 7.2 (Dénotation d'un schéma et d'un environnement) *La dénotation d'un schéma de type $\sigma = \forall\bar{\alpha}[C].\varsigma$ sous une affectation ϕ (notée $[\![\sigma]\!]_\phi$) est définie comme l'ensemble $\{\phi'(\varsigma) \mid (\phi' \backslash \bar{\alpha} = \phi \backslash \bar{\alpha}) \wedge \phi' \vdash C\}$ (si cet ensemble n'est pas vide, elle n'est pas définie sinon).*
Cette définition est étendue « point par point » aux environnements en posant $[\![\Gamma]\!]_\phi = [\![\cdot]\!]_\phi \circ \Gamma$.

Nous pouvons alors énoncer le théorème de correction.

Théorème 7.1 (Correction) *Si $pc, \Gamma, C \vdash e : \varsigma\ [\rho]$ est dérivable dans SHM(S) et $\phi \vdash C$ alors $\phi(pc), [\![\Gamma]\!]_\phi \vdash e : \phi(\varsigma)\ [\phi(\rho)]$ est dérivable dans SB(S).*

On peut définir de la même manière que pour SB(S) une relation $\cdot \lessdot_L \cdot$. On en déduit un corollaire du théorème 5.2 pour le système SHM(S).

Corollaire 7.2 (Non-interférence) *Soit L un ensemble d'étiquettes clos et ℓ une étiquette n'appartenant pas à L. Soient e_1 et e_2 deux expressions du calcul étiqueté telles que $e_1 \lessdot_L e_2$ et $\pi, \emptyset, C \vdash e_1 : \mathsf{int}^\ell\ [\rho]$. Si $e_i \,/\, \emptyset \to^* v_i$ (pour $i \in \{1,2\}$) alors $v_1 = v_2$.*

8 Implantation

8.1 Instances

Dans les présentations que nous avons données pour les systèmes SB(S) et SHM(S), l'ensemble des types $\mathcal{S}$ n'est pas totalement spécifié. Il nous reste donc à choisir une (ou plusieurs) instance(s) précises des systèmes.

Valeurs :

$$\textsc{hv-Int}\quad \Gamma, C \vdash k : \mathsf{int}^{*} \qquad\qquad \textsc{hv-Unit}\quad \Gamma, C \vdash () : \mathsf{unit}^{*}$$

$$\textsc{hv-Var}\quad \frac{\Gamma(x) = \forall\bar{\alpha}[D].\varsigma \qquad C \Vdash \exists\bar{\alpha}.D}{\Gamma, C \wedge D \vdash x : \varsigma} \qquad\qquad \textsc{hv-Exn}\quad \frac{\Gamma, C \vdash v : \varsigma}{\Gamma, C \vdash \mathsf{exn}_{\varepsilon}\ v : ([\varepsilon.\mathsf{Pre}^{*}\varsigma; *]\ \mathsf{exn})^{*}}$$

$$\textsc{hv-Abs}\quad \frac{\pi, \Gamma[x \mapsto \varsigma'], C \vdash e : \varsigma\ \ [\rho]}{\Gamma, C \vdash \lambda x.e : (\varsigma' \xrightarrow{\pi\ [\rho]} \varsigma)^{*}} \qquad \textsc{hv-Label}\quad \frac{\Gamma, C \vdash v : \varsigma \qquad C \Vdash \ell \lhd \varsigma}{\Gamma, C \vdash (\ell : e) : \varsigma} \qquad \textsc{hv-Sub}\quad \frac{\Gamma, C \vdash v : \varsigma' \qquad C \Vdash \varsigma' \leq \varsigma}{\Gamma, C \vdash v : \varsigma}$$

Expressions :

$$\textsc{he-Value}\quad \frac{\Gamma, C \vdash v : \varsigma}{*, \Gamma, C \vdash v : \varsigma\ \ [*]} \qquad \textsc{he-App}\quad \frac{\Gamma, C \vdash v_1 : (\varsigma' \xrightarrow{\lambda \sqcup \pi\ [\rho]} \varsigma)^{\lambda} \qquad \Gamma, C \vdash v_2 : \varsigma' \qquad C \Vdash \lambda \lhd \varsigma}{\pi, \Gamma, C \vdash v_1\, v_2 : \varsigma\ \ [\rho]} \qquad \textsc{he-Ref}\quad \frac{\Gamma, C \vdash v : \varsigma \qquad C \Vdash \pi \lhd \varsigma}{\pi, \Gamma, C \vdash \mathsf{ref}\ v : (\varsigma\ \mathsf{ref})^{*}\ \ [*]}$$

$$\textsc{he-Assign}\quad \frac{\Gamma, C \vdash v_1 : (\varsigma\ \mathsf{ref})^{\lambda} \qquad \Gamma, C \vdash v_2 : \varsigma \qquad C \Vdash \lambda \sqcup \pi \lhd \varsigma}{\pi, \Gamma, C \vdash v_1 := v_2 : \mathsf{unit}^{\lambda}\ \ [*]} \qquad \textsc{he-Deref}\quad \frac{\Gamma, C \vdash v : (\varsigma'\ \mathsf{ref})^{\lambda} \qquad C \Vdash \varsigma' \leq \varsigma \qquad C \Vdash \lambda \lhd \varsigma}{\pi, \Gamma, C \vdash !\, v : \varsigma\ \ [*]}$$

$$\textsc{he-Raise}\quad \frac{\Gamma, C \vdash v : (\rho\ \mathsf{exn})^{\lambda} \qquad C \Vdash \pi \sqcup \lambda \lhd \rho}{\pi, \Gamma, C \vdash \mathsf{raise}\ v : *\ \ [\rho]} \qquad \textsc{he-Let}\quad \frac{\Gamma, C \wedge D \vdash v : \varsigma' \qquad \pi, \Gamma[x \mapsto \forall\bar{\alpha}[D].\varsigma'], C \vdash e : \varsigma\ \ [\rho] \qquad \mathrm{fv}(C, \Gamma) \cap \bar{\alpha} = \emptyset}{\pi, \Gamma, C \wedge \exists\bar{\alpha}.D \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \varsigma\ \ [\rho]}$$

$$\textsc{he-Bind}\quad \frac{\pi, \Gamma, C \vdash e_1 : \varsigma'\ \ [\rho_1] \qquad \pi \sqcup \mu(\rho_1), \Gamma[x \mapsto \varsigma'], C \vdash e_2 : \varsigma\ \ [\rho_2]}{\pi, \Gamma, C \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \varsigma\ \ [\rho_1 \sqcup \rho_2]}$$

$$\textsc{he-Try}\quad \frac{\pi, \Gamma, C \vdash e_1 : \varsigma\ \ [\varepsilon.\mathsf{Pre}^{\lambda}\varsigma'; \rho_1'] \qquad \pi \sqcup \lambda, \Gamma[x \mapsto \varsigma'], C \vdash e_2 : \varsigma\ \ [\rho_2] \qquad C \Vdash \lambda \lhd \varsigma}{\pi, \Gamma, C \vdash \mathsf{try}_{\varepsilon}\ e_1\ \mathsf{with}\ x \succ e_2 : \varsigma\ \ [\rho_1' \sqcup \rho_2]}$$

$$\textsc{he-TryAll}\quad \frac{\pi, \Gamma, C \vdash e_1 : \varsigma\ \ [\rho_1] \qquad \pi \sqcup \mu(\rho_1), \Gamma[x \mapsto (\rho_1\ \mathsf{exn})^{*}], C \vdash e_2 : \varsigma\ \ [\rho_2] \qquad C \Vdash \mu(\rho_1) \lhd \varsigma}{\pi, \Gamma, C \vdash \mathsf{tryall}\ e_1\ \mathsf{with}\ x \succ e_2 : \varsigma\ \ [\rho_2]}$$

$$\textsc{he-Label}\quad \frac{\pi \sqcup \ell, \Gamma, C \vdash e : \varsigma\ \ [\rho] \qquad C \Vdash \ell \lhd \varsigma}{\pi, \Gamma, C \vdash (\ell : e) : \varsigma\ \ [\rho]} \qquad \textsc{he-Sub}\quad \frac{\pi, \Gamma, C \vdash e : \varsigma'\ \ [\rho'] \qquad C \Vdash \varsigma' \leq \varsigma \qquad C \Vdash \rho' \leq \rho}{\pi, \Gamma, C \vdash e : \varsigma\ \ [\rho]}$$

Fig. 11: Le système SHM(S)

Dans notre idée, ces instances doivent comporter les formes de types prévues par les axiomes 1, 2 et 6, ainsi qu'une alternative Abs comme nous l'avons expliqué dans la section 4.1.
Cependant, certains choix importants restent ouverts. Deux alternatives sont en particulier envisageables :

– **Sous-typage atomique ou non-atomique :** Le choix entre sous-typage atomique et sous-typage non-atomique repose principalement sur la forme donnée au constructeur Abs. Dans le cas du sous-typage atomique, celui-ci doit admettre deux arguments : $\mathsf{Abs}^{\ell} s$. Les alternatives sont alors en fait des triplets de la forme (b, ℓ, s) avec $b \in \{\mathsf{Abs}, \mathsf{Pre}\}$. Dans le cas du sous-typage non-atomique, le constructeur Abs peut être une constante.
– **Présence de types récursifs ou non :** Si on souhaite pouvoir typer tous les programmes acceptés par *Caml*, il est *a priori* nécessaire de doter le système de types récursifs. On peut en effet définir en *Caml* des fonctions récursives qui lèvent une exception dont la fonction elle-même est un argument. Par exemple :

```
exception A of int -> int;;
let rec f x = raise (A f); x + 1;;
```

  Cependant, l'ajout de types récursifs risque d'accroître la complexité du système et de diminuer la lisibilité des informations affichées à l'utilisateur. De plus, les exemples pour lesquels ils sont nécessaires restent relativement pathologiques. Dans les cas usuels, les arguments passés aux constructeurs d'exception sont des valeurs brutes.

8.2 Algorithme d'inférence et résolution des contraintes

L'implantation d'un système tel que SHM(S) comporte principalement deux parties : une première partie chargée d'inférer les types et d'engendrer les contraintes correspondantes ; une seconde partie qui teste la satisfiabilité des contraintes et les simplifie (principalement pour des raisons d'efficacité).
Pour obtenir un algorithme d'inférence pour SHM(S), il suffit de ré-écrire les règles de la figure 11 de telle sorte qu'elles soient dirigées par la syntaxe. Nous ne détaillons pas le système obtenu, un tel procédé étant présenté pour HM(X) dans [SMZ99].
En ce qui concerne la résolution et la simplification des contraintes, un premier problème réside dans la présence dans notre logique de contraintes dont la forme n'est pas « standard » : $\lambda \lhd \varsigma$, $\lambda \lhd \rho$ et $\mu(\rho) \leq \lambda$. On peut cependant coder ces contraintes à l'aide de contraintes d'inégalité et de contraintes conditionnelles :

– On peut facilement se placer dans un cas où les types sont toujours de la forme $s = t^{\lambda}$. On peut donc ramener une contrainte $\lambda_0 \lhd s$ à une contrainte sur des étiquettes $\lambda_0 \leq \lambda$ en unifiant le type s avec t^{λ}.
– Une contrainte $\lambda \lhd r$ peut être codée par une contrainte conditionnelle de la forme $\mathsf{Pre} \leq r \ ?\ \partial(\mathsf{Pre}^{\lambda}\_) \leq r$.
– La forme de contrainte la plus difficile à représenter est $\mu(\rho) \leq \lambda$. Deux approches sont possibles. On peut coder cette contrainte par une conditionnelle de la forme $\mathsf{Pre} \leq \rho \ ?\ \rho \leq \mathsf{Pre}^{\partial\lambda}\_$. On peut également choisir d'adjoindre à chaque rangée ρ une variable d'étiquette λ et d'engendrer des contraintes sur λ simultanément aux contraintes générées sur ρ de telle sorte que l'inégalité $\mu(\rho) \leq \lambda$ soit toujours vraie.

Ces codages des contraintes nous assurent la décidabilité de la satisfiabilité. Des méthodes de résolution et de simplification sur des contraintes des formes utilisées ci-dessus sont détaillées dans [Pot00]. On pourra cependant également envisager de développer un algorithme de résolution spécifique à ces formes de contraintes.

8.3 À propos de l'ordre d'évaluation

La syntaxe que nous avons introduite pour $\mathrm{ML_{re}}$ nous a entre autres permis de nous affranchir du problème de l'ordre d'évaluation (section 2.1). Cependant, on souhaite dans la pratique obtenir des règles pour un calcul sans les contraintes syntaxiques que nous avons imposées (i.e. avec la possibilité de placer une expression en tout lieu).
Nous prenons comme exemple le cas de l'application $e_1\, e_2$. Avec une évaluation de gauche vers la droite, l'expression $e_1\, e_2$ se code $\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ (\mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ x_1\, x_2)$. On obtient ainsi la règle induite suivante (dans le système SB(S)) :

$$\frac{pc, \Gamma \vdash e_1 : (s' \xrightarrow{pc \sqcup \ell \sqcup \mu(r_1, r_2)\ \ [r]} s)^{\ell}\ \ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma \vdash v_2 : s'\ \ [r_2] \qquad \ell \lhd s}{pc, \Gamma \vdash e_1\, e_2 : s\ \ [r \sqcup r_1 \sqcup r_2]}$$

Deux points sont intéressants dans cette règle. Tout d'abord, la rangée d'alternative obtenue pour $e_1\, e_2$ est $r \sqcup r_1 \sqcup r_2$: les exceptions levées par $e_1\, e_2$ sont celles levées par la réduction de e_1 (r_1), celles de e_2 (r_2) et celles levées après l'application de la fonction (r). D'autre part, le *pc* sous lequel l'expression e_2 est typé est augmenté de $\mu(r_1)$: en effet, lorsque e_2 est réduite, on sait que e_1 n'a pas déclenché d'exception.

Si on choisit d'évaluer de la droite vers la gauche, l'expression $e_1\, e_2$ se traduit alors $\mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ (\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ x_1\, x_2)$ et on obtient une pollution inverse pour la règle de typage :

$$\frac{pc \sqcup \mu(r_2), \Gamma \vdash e_1 : (s' \xrightarrow{pc \sqcup \ell \sqcup \mu(r_1, r_2)\ [r]} s)^\ell\ \ [r_1] \qquad pc, \Gamma \vdash v_2 : s'\ \ [r_2] \qquad \ell \lhd s}{pc, \Gamma \vdash e_1\, e_2 : s\ \ [r \sqcup r_1 \sqcup r_2]}$$

Enfin, si l'ordre d'évaluation n'est pas spécifié, on peut choisir d'utiliser une règle symétrique qui comprend les deux précédentes :

$$\frac{pc \sqcup \mu(r_2), \Gamma \vdash e_1 : (s' \xrightarrow{pc \sqcup \ell \sqcup \mu(r_1, r_2)\ [r]} s)^\ell\ \ [r_1] \qquad pc \sqcup \mu(r_1), \Gamma \vdash v_2 : s'\ \ [r_2] \qquad \ell \lhd s}{pc, \Gamma \vdash e_1\, e_2 : s\ \ [r \sqcup r_1 \sqcup r_2]}$$

Cependant, l'ordre d'évaluation doit rester totalement déterminé, pour chaque redex, par le contexte d'évaluation.

Conclusion

Nous avons présenté un système de types avec annotations de sécurité pour le langage ML doté des principales constructions impératives que l'on est en droit d'attendre. L'approche que nous avons suivie pour établir et prouver ce système présente quelques points intéressants plus généraux. Le choix d'une syntaxe inspirée des *formes A-normales* nous a permis de bien séparer les caractéristiques des différentes constructions du langage. Nos règles de typage sont ainsi relativement simples et lisibles, et les redondances sont limitées. La définition d'un système intermédiaire, SB(S), avant de s'intéresser à SHM(S) nous a donné la possibilité de bien distinguer dans notre étude les questions liées à l'analyse de flots d'information des points – plus techniques – relatifs à l'écriture d'un système de types avec sous-typage et polymorphisme. Enfin, le calcul à crochets est un formalisme simple et intuitif pour étudier des propriétés de simulation entre deux expressions et leurs réductions. Il devrait pouvoir être utilisé pour d'autres langages et d'autres études.

La définition et la preuve d'un tel système ne constitue qu'une première étape. Il serait pertinent de s'intéresser à son implantation effective dans un système de programmation d'échelle réelle, tel que *Caml Light*, dans le but de proposer un outil d'analyse réaliste. Outre le travail d'implantation pure, cela nécessitera l'étude d'algorithmes de gestion de contraintes et la résolution de plusieurs problèmes théoriques ou ergonomiques encore ouverts.

Bibliographie

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999. `http://pa.bell-labs.com/~abadi/Papers/flowpopl.ps`.

[ALL96] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 83–91, Philadelphia, Pennsylvania, May 1996. `http://pa.bell-labs.com/~abadi/Papers/make-preprint.ps`.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1) :56–76, January 1980.

[BL75] D. E. Bell and Leonard J. LaPadula. Secure computer systems : Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corp., Bedford, Massachusetts, July 1975. `http://www.mitre.org/resources/centers/infosec/infosec.html`.

[CP01] Sylvain Conchon and François Pottier. JOIN(X) : Constraint-based type inference for the join-calculus. To appear at *ESOP'01*. `http://pauillac.inria.fr/~fpottier/publis/conchon-fpottier-esop01.ps.gz`, April 2001.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7) :504–513, July 1977.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, June 1993. `http://www.cs.rice.edu/CS/PLT/Publications/pldi93-fsdf.ps.gz`.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus : Programming with secrecy and integrity. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998. `http://cm.bell-labs.com/cm/cs/who/nch/slam.ps`.

[Mye99a] Andrew C. Myers. JFlow : practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press. `http://www.cs.cornell.edu/andru/papers/popl99/myers-popl99.ps.gz`.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999. Technical Report MIT/LCS/TR-783. `http://www.cs.cornell.edu/andru/release/tr783.ps.gz`.

[ØP97] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6) :557–591, November 1997. `http://www.cs.purdue.edu/homes/palsberg/paper/jfp97.ps.gz`.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, Montréal, Canada, September 2000. `http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz`.

[Pot00] François Pottier. A versatile constraint-based type inference system. To appear in the *Nordic Journal of Computing*, November 2000. `http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz`.

[Pot01] François Pottier. A semi-syntactic soundness proof for HM(X). Unpublished draft. `http://pauillac.inria.fr/~fpottier/publis/fpottier-hmx.ps.gz`, March 2001.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. `http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz`.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214 :607–621, April 1997. `http://www.cs.nps.navy.mil/people/faculty/volpano/papers/tapsoft97.ps.Z`.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4) :343–356, December 1995.

[ZM01] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming (ESOP'01)*, Lecture Notes in Computer Science. Springer Verlag, April 2001. `http://www.cs.cornell.edu/zdance/lincont.ps`.

Troisième partie

Information Flow Inference for ML

François Pottier et Vincent Simonet

Abstract

This paper presents a type-based information flow analysis for a call-by-value λ-calculus equipped with references, exceptions and let*-polymorphism, which we refer to as Core ML. The type system is constraint-based and has decidable type inference. Its non-interference proof is reasonably light-weight, thanks to the use of a number of orthogonal techniques. First, a syntactic segregation between* values *and* expressions *allows a lighter formulation of the type system. Second, non-interference is reduced to* subject reduction *for a non-standard language extension. Lastly, a* semi-syntactic *approach to type soundness allows dealing with constraint-based polymorphism separately.*

1 Introduction

Information flow analysis consists in statically determining how a program's outputs are related to its inputs, i.e. how the former *depend*, directly or indirectly, on the latter. This allows establishing *secrecy* and *integrity* properties of a program, i.e. proving that some aspects of its behavior convey no information about those of its inputs deemed "secret", or remain independent of those deemed "unreliable". These properties are instances of *non-interference* [GM82]: they state the absence of certain dependencies.
Because information flow analysis is complex and error-prone, it must be automated. During the past few years, several researchers have advocated its formulation as a *type system*. Then, existing type inference techniques provide automation, while type signatures provide concise, formal security specifications.
Our interest is in designing – and proving correct – a type-based information flow analysis for (the kernel of) a realistic sequential programming language. (In the presence of concurrency, the termination of a process is observable by other processes, creating new ways to leak information and requiring more restrictive type systems. Hence, it appears reasonable to first experiment with information flow control in a sequential setting.) To date, most formal results obtained in this area concern extremely reduced programming languages. Several papers address pure λ-calculi [HR98, ABHR99, PC00]. Volpano *et al.* [VSI96, VS97] study a core imperative programming language, where all variables store integers. Standing in sharp contrast, Myers [Mye99a, Mye99b] considers the full Java language, including objects, exceptions, parameterized classes, etc. However, he does not give a formal proof of correctness; indeed, our formal approach uncovered a couple of flaws in his type system (see section 7.3).
In an attempt to bridge the gap between these approaches, we consider a call-by-value λ-calculus equipped with references, exceptions and let-polymorphism, which we refer to as Core ML. (Presentation set aside, it is identical to Wright and Felleisen's Core ML [WF94], except our exception names have global scope and are not first-class values.) Such a calculus can be viewed as the core of the functional programming language Caml-Light [LD+97]. We endow it with a polymorphic, constraint-based type system, called MLIF, which has decidable type inference and guarantees non-interference.
A (monomorphic) treatment of references in a higher-order language can be found in [ZM01]. Exceptions have been studied by Myers [Mye99a, Mye99b] for Java. However, Myers' treatment relies on Java's explicit, monomorphic throws clauses, whereas our type system uses a more flexible, polymorphic effect analysis, giving rise to issues discussed in section 10. The combination of references, exceptions and constrained let-polymorphism, together with our use of a standard subject reduction technique to establish non-interference, is novel. Our treatment of un-annotated tuple types and of polymorphic equality form ancillary contributions.

2 Overview

Type systems are typically used to establish *safety* properties, i.e. prove that a certain invariant holds throughout the execution of a program. Type safety is such a property. However, non-interference [GM82] requires *two* independent program runs, given different inputs, to yield the same output. As a result, its proof is often more delicate.

Abadi *et al.* [ALL96] devised a *labelled* operational semantics of the λ-calculus, where the labels attached to a term indicate how much information it carries. Executing a program under such a semantics amounts to performing a *dynamic* dependency analysis along with the actual computation. Pottier and Conchon [PC00] later showed how *static*, type-based dependency analyses could be systematically derived, and proven safe, from such a labelled semantics.

Unfortunately, in a programming language with side effects, it is possible to leak information through the *absence* of a certain effect. Indeed, consider the program fragment "$\mathsf{if}\, x = 1 \,\mathsf{then}\, y := 1$". If, after executing this statement, y isn't 1, then x cannot be 1 either. Thus, in that case, execution transfers information about x to y, even though no assignment takes place, since the statement $y := 1$ is skipped. It appears difficult for a labelled semantics to account for the effect of code that is *not* executed; so, the approach must be reconsidered.

Direct non-interference proofs, although straightforward for simple programming languages [VSI96], become increasingly complex in richer languages, requiring cumbersome invariants to be manipulated [ZM01]. To avoid this pitfall, we break our proof down into several independent steps. First, we define a special-purpose extension of the language, which allows explicit reasoning about the commonalities and differences between *two* arbitrary program configurations, and prove it adequate in a certain sense. Then, we define a type system for this extended language, and prove that it enjoys a subject reduction property. Lastly, we show that non-interference for the base language is a consequence of these results. In other words, we reduce the initial problem to subject reduction – a safety property – for our special-purpose language. The invariant preserved by reduction is thus expressed in the type system itself, making it easier to reason about.

In keeping with the ML tradition, our type system has let-polymorphism and type inference. In addition to structure, our types describe effects and security levels; as a result, the former allows writing code that is generic with respect to all three. The latter is an indispensable help, because our types are verbose and information flow is often un-intuitive. Because we employ subtyping (and, to a lesser extent, other forms of constraints), our type inference system is constraint-based. Yet, if generalization, instantiation, and constraint manipulation were part of the type system from the outset, our subject reduction proof would be significantly obfuscated. To work around this problem, we adopt a *semi-syntactic* approach [Pot01], which again consists in breaking down the construction into two steps. First, we present a system equipped with an extensional form of polymorphism, whose formal treatment is remarkably un-intrusive. Then, we build a constraint-based system in the style of HM(X) [OSW99], which we prove correct with respect to the former.

We will now proceed as follows. We first present the syntax of Core ML (section 3). Then, we introduce our technical extension of it, which we refer to as "Core ML^2", give an operational semantics to both languages at once, and show how they relate to each other (section 4). Section 5 introduces MLIF$_0$, a type system for Core ML^2, and establishes subject reduction. Combining these results, we obtain a non-interference property for Core ML (section 6). In section 7, we digress and discuss a few language extensions. Culminating our development, section 8 presents MLIF, a constraint-based type system which we prove correct with respect to MLIF$_0$, allowing type inference. Sections 9 and 10 give some examples and conclude.

By lack of space, many proofs are omitted; they can be found in the full version of this paper [PS01].

3 Core ML

Let k range over integers; let x, m, ε range over disjoint denumerable sets of *program variables*, *memory locations*, and *exception names*, respectively. Then, *values*, *outcomes*, *expressions* and *evaluation contexts* are defined as follows:

$$
\begin{array}{rcl}
v & ::= & x \mid \lambda\langle f\rangle x.e \mid k \mid () \mid m \mid \varepsilon\, v \\
o & ::= & v \mid \mathsf{raise}\ (\varepsilon\, v) \\
e & ::= & o \\
 & \mid & v\, v \\
 & \mid & \mathsf{ref}\ v \mid v := v \mid !\, v \\
 & \mid & \mathsf{raise}\ v \\
 & \mid & \mathsf{let}\ x = v\ \mathsf{in}\ e \\
 & \mid & E[e] \\
E & ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \\
 & \mid & [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e \\
 & \mid & [\,]\ \mathsf{handle}\ x \succ e
\end{array}
$$

Our values include variables, λ-abstractions, integers, a unit constant, memory locations, and exceptions. An abstraction $\lambda\langle f\rangle x.e$ may recursively refer to itself through the program variable f. (This is done merely to avoid dealing with recursion separately.) Every exception name ε can be used as a data constructor to build exception values of the form $\varepsilon\, v$. Outcomes, known as *answers* in [WF94], represent inactive computations; they are either values or unhandled exceptions of the form $\mathsf{raise}\ (\varepsilon\, v)$. An expression is an outcome, a so-called *basic expression*, a let construct, or another expression enclosed within an evaluation context.

Basic expressions include function applications as well as instances of four primitive effectful operations, which allow allocating, updating, dereferencing memory cells, and raising exceptions. They are built out of values, rather than out of arbitrary sub-expressions. This syntactic restriction, which is reminiscent of Flanagan *et al.*'s *A-normal forms* [FSDF93], offers a number of advantages. First, it enables a lighter formulation of our type-and-effect system. Indeed, because values have no computational effect, a basic expression's sub-expressions do not contribute to its effect. Furthermore, it allows our system to remain independent of the evaluation strategy, i.e. of the choice of left-to-right vs. right-to-left evaluation order. User programs, expressed in a more liberal syntax, must be translated down into our restricted syntax before they can be analyzed; different evaluation strategies will simply correspond to different translation schemes (see section 5.7).

The let construct $\mathsf{let}\ x = v\ \mathsf{in}\ e$ has the same meaning as the basic expression $(\lambda\langle f\rangle x.e)\, v$ (where f is not free in e). However, as usual in ML [WF94], the let keyword directs the type checker to give x polymorphic type. Following Wright [Wri95], we require the binding to contain a value v, rather than an arbitrary sub-expression, so as to avoid unsoundness in the presence of imperative features. As a result, let constructs do not appear among evaluation contexts.

Evaluation contexts provide glue to combine expressions and specify their evaluation order. The expression $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ evaluates e_1, binds its value (if any) to x, then evaluates e_2. The bind keyword does not request type generalization; it merely expresses sequentiality. Our decision of making let and bind separate constructs emphasizes this distinction. The handle constructs are dual to bind: they specify what happens after the expression under scrutiny raises an exception, rather than after it returns a value.

The meaning of the memory locations which occur in a Core ML expression is given by a *store* μ, i.e. a partial map from memory locations to values. We write $\mu[m \mapsto v]$ and $\mu \oplus [m \mapsto v]$ for the store which maps m to v and otherwise agrees with μ; the latter is defined only if $m \notin \mathrm{dom}(\mu)$.

4 Core ML^2

4.1 Presentation

Non-interference requires reasoning about two programs and proving that they share some sub-terms throughout execution. To make such reasoning easier, we choose to represent them as a *single* term of an extended language, called Core ML^2, rather than as a pair of Core ML terms. The extension is as follows:

$$\begin{array}{rcl} v & ::= & \ldots \mid \langle v \mid v\rangle \mid \mathsf{void} \\ o & ::= & \ldots \mid \langle o \mid o\rangle \\ e & ::= & \ldots \mid \langle e \mid e\rangle \end{array}$$

The new construct $\langle\cdot \mid \cdot\rangle$ is a pairing construct. In other words, the Core ML^2 term $\langle e_1 \mid e_2\rangle$ encodes the pair of Core ML terms (e_1, e_2). It is important to note that it can appear at an arbitrary depth within a term. For instance, assuming v is a Core ML value, the terms $\langle v_1 \mid v_2\rangle\, v$ and $\langle v_1\, v \mid v_2\, v\rangle$ both encode the pair $(v_1\, v, v_2\, v)$. The former, however, is more informative, because it explicitly records the fact that the application node and its argument v are shared, while the latter doesn't. We do not allow nesting $\langle\cdot \mid \cdot\rangle$ constructs.

We need to keep track of sharing not only between expressions, but also between stores. However, distinct stores may have distinct domains. To account for this fact, we introduce a special constant void. By creating bindings of the form $m \mapsto \langle v \mid \mathsf{void}\rangle$ and $m \mapsto \langle \mathsf{void} \mid v\rangle$ in the store, we represent situations where a memory location m is bound within only one of the two Core ML expressions encoded by a Core ML^2 term.

A *configuration* $e \,/_i\, \mu$ is a triple of an expression e, a store μ, and an index $i \in \{\bullet, 1, 2\}$, whose purpose is explained in section 4.2. We write $e \,/\, \mu$ for $e \,/_\bullet\, \mu$.

A configuration $e \,/_i\, \mu$ is *well-formed* if the following conditions hold:

- e does not contain void; furthermore, if $i \in \{1, 2\}$, then e is a Core ML expression;
- for each $m \in \mathrm{dom}(\mu)$, $\mu(m)$ is of the form v, $\langle v \mid \mathsf{void}\rangle$ or $\langle \mathsf{void} \mid v\rangle$, where v does not contain void.

$$
\begin{array}{llr}
\textbf{Basic reductions} & (\lambda\langle f\rangle x.e)\, v \,/_i\, \mu \to e[x \Leftarrow v][f \Leftarrow \lambda\langle f\rangle x.e] \,/_i\, \mu & (\beta) \\
& \mathsf{ref}\ v \,/_i\, \mu \to m \,/_i\, \mu \oplus [m \mapsto \mathrm{new}_i\, v] & (\mathrm{ref}) \\
& m := v \,/_i\, \mu \to () \,/_i\, \mu[m \mapsto \mathrm{update}_i\, \mu(m)\, v] & (\mathrm{assign}) \\
& !\, m \,/_i\, \mu \to \mathrm{read}_i\, \mu(m) \,/_i\, \mu & (\mathrm{deref}) \\
& \mathsf{let}\ x = v\ \mathsf{in}\ e \,/_i\, \mu \to e[x \Leftarrow v] \,/_i\, \mu & (\mathrm{let}) \\
\textbf{Sequencing} & \mathsf{bind}\ x = v\ \mathsf{in}\ e \,/_i\, \mu \to e[x \Leftarrow v] \,/_i\, \mu & (\mathrm{bind}) \\
& \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ \varepsilon\, x \succ e \,/_i\, \mu \to e[x \Leftarrow v] \,/_i\, \mu & (\mathrm{handle}) \\
& \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e \,/_i\, \mu \to e[x \Leftarrow \varepsilon\, v] \,/_i\, \mu & (\mathrm{handle\text{-}all}) \\
& E[o] \,/_i\, \mu \to o \,/_i\, \mu \quad \text{if } \neg(E \text{ handles } \lfloor o \rfloor_1 \vee E \text{ handles } \lfloor o \rfloor_2) & (\mathrm{throw\text{-}context}) \\
\textbf{Lifting} & E[\langle o_1 \mid o_2 \rangle] \,/\, \mu \to \langle \lfloor E \rfloor_1[o_1] \mid \lfloor E \rfloor_2[o_2] \rangle \,/\, \mu \quad \text{if none of the sequencing rules applies} & (\mathrm{lift\text{-}context}) \\
& \langle v_1 \mid v_2 \rangle\, v \,/\, \mu \to \langle v_1\, \lfloor v \rfloor_1 \mid v_2\, \lfloor v \rfloor_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}app}) \\
& \langle v_1 \mid v_2 \rangle := v \,/\, \mu \to \langle v_1 := \lfloor v \rfloor_1 \mid v_2 := \lfloor v \rfloor_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}assign}) \\
& !\, \langle v_1 \mid v_2 \rangle \,/\, \mu \to \langle\, !\, v_1 \mid !\, v_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}deref}) \\
& \mathsf{raise}\ \langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2 \rangle \,/\, \mu \to \langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid \mathsf{raise}\ (\varepsilon_2\, v_2) \rangle \,/\, \mu & (\mathrm{lift\text{-}raise}) \\
\textbf{Reduction under a context} & \dfrac{e \,/_i\, \mu \to e' \,/_i\, \mu'}{E[e] \,/_i\, \mu \to E[e'] \,/_i\, \mu'} & (\mathrm{context}) \\
& \dfrac{e_i \,/_i\, \mu \to e_i' \,/_i\, \mu' \qquad e_j = e_j' \qquad \{i, j\} = \{1, 2\}}{\langle e_1 \mid e_2 \rangle \,/\, \mu \to \langle e_1' \mid e_2' \rangle \,/\, \mu'} & (\mathrm{bracket})
\end{array}
$$

Auxiliary functions

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
\mathrm{new}_\bullet\, v & = & v & \mathrm{update}_\bullet\, v\, v' & = & v' & \mathrm{read}_\bullet\, v & = & v \\
\mathrm{new}_1\, v & = & \langle v \mid \mathsf{void} \rangle & \mathrm{update}_1\, v\, v' & = & \langle v' \mid \lfloor v \rfloor_2 \rangle & \mathrm{read}_1\, v & = & \lfloor v \rfloor_1 \\
\mathrm{new}_2\, v & = & \langle \mathsf{void} \mid v \rangle & \mathrm{update}_2\, v\, v' & = & \langle \lfloor v \rfloor_1 \mid v' \rangle & \mathrm{read}_2\, v & = & \lfloor v \rfloor_2
\end{array}
$$

Figure 1: Operational semantics of Core ML^2

Furthermore, we consider a memory location m to be bound within e and μ according to the following rules:

- if $\mu(m)$ is of the form v, then m is in scope everywhere within e and μ;
- if $\mu(m)$ is of the form $\langle v \mid \mathsf{void} \rangle$ (resp. $\langle \mathsf{void} \mid v \rangle$), then:
  - m is in scope within the left (resp. right) branch of every $\langle \cdot \mid \cdot \rangle$ construct in μ;
  - if $i = \bullet$, then m is in scope within the left (resp. right) branch of every $\langle \cdot \mid \cdot \rangle$ construct in e; if $i = 1$ (resp. $i = 2$), then m is in scope within e.

A configuration $e \,/_i\, \mu$ is *closed* if all occurrences of memory locations in it are in scope. We restrict our attention to well-formed, closed configurations. (These technical notions are preserved by reduction and guarantee that void is used exclusively in store bindings, as described above.) Furthermore, we identify configurations up to consistent renamings of memory locations.
The correspondence between Core ML and Core ML^2 is made explicit by means of two *projection* functions $\lfloor \cdot \rfloor_i$, where i ranges over $\{1, 2\}$. They satisfy $\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i$ and are homomorphisms on other expression forms. They are extended to stores as follows: $\lfloor \mu \rfloor_i$ maps m to $\lfloor \mu(m) \rfloor_i$ if and only if the latter is defined and isn't void. Lastly, the projection of a configuration is defined by $\lfloor e \,/\, \mu \rfloor_i = \lfloor e \rfloor_i \,/\, \lfloor \mu \rfloor_i$.

4.2 Semantics

The small-step operational semantics of Core ML^2 is given in figure 1. The first two groups of reduction rules are those of Core ML, with a few technical twists explained below. The rules in the third group are specific to Core ML^2; they allow discarding sharing information if reduction cannot otherwise take place. The rules in the fourth group allow reduction under a context.

The rules are designed so that the image of any reduction step through a projection function is again a valid reduction step. Reduction may take place outside brackets, causing both projections to perform the same reduction step; inside brackets, letting one projection compute independently, while the other remains stationary; or lift up the bracket boundary, discarding some sharing information, while leaving both projections unchanged.
The capture-free substitution of v for x in e, written $e[x \Leftarrow v]$, is defined in the usual way, except at $\langle \cdot \mid \cdot \rangle$ nodes, where we must use an appropriate projection of v in each branch: $\langle e_1 \mid e_2 \rangle[x \Leftarrow v]$ is $\langle e_1[x \Leftarrow \lfloor v \rfloor_1] \mid e_2[x \Leftarrow \lfloor v \rfloor_2] \rangle$.
We would like the rules in the first two groups to be applicable under any context. However, (ref), (assign) and (deref) need a small amount of contextual information. Indeed, the store must be accessed in a context-dependent manner: operations which take place inside a $\langle \cdot \mid \cdot \rangle$ construct must use or affect only one projection of the store. The index i carried by configurations is used for this purpose. Its value is $\bullet$ when dealing with top-level reduction steps; it is made 1 (resp. 2) by rule (bracket) when reducing within the left (resp. right) branch of a $\langle \cdot \mid \cdot \rangle$ construct. It is used in the auxiliary functions new_i, update_i and read_i to access the store in an appropriate way.
The rules in the second group describe how values and exceptions are bound (i.e. handled) or propagated. We say that E handles o if and only if $E[o]$ is reducible through (bind), (handle) or (handle-all).
The rules in the third group have no computational content: they leave both projections unchanged. Their purpose is to prevent $\langle \cdot \mid \cdot \rangle$ constructs from blocking reduction, which is done by lifting them up, thus causing some sub-terms to be duplicated, but allowing reduction to proceed independently within each branch. For instance, the left-hand expression in (lift-app) is not a β-redex. In its reduct, the application node and the sub-term v are duplicated, allowing two β-redexes to appear. A somewhat analogous rule can be found in Abadi *et al.*'s labelled semantics of the λ-calculus [ALL96]. To understand the significance of the "lift" rules, one must bear in mind that the contents of every $\langle \cdot \mid \cdot \rangle$ construct will be viewed as "secret". By causing new sub-terms to become secret during reduction, these rules actually provide an explicit description of information flow. Our design attempts to discard as little sharing information as possible; indeed, replacing all of these rules with $e \rightarrow \langle \lfloor e \rfloor_1 \mid \lfloor e \rfloor_2 \rangle$, while computationally correct, would cause the type system to view every expression as "secret". Our "lift" rules are not optimal, because there are situations where they discard sharing information which could conceivably be preserved, and because they never re-create sharing information; however, they are precise enough for our purposes, which is to prove a particular type system sound.
We remark that, because of rule (bracket), reductions under a bracket may be interleaved in an arbitrary order, causing non-determinism to arise. However, confluence is preserved, as stated below.

Lemma 4.1 (Confluence) *If $e \,/\, \mu \rightarrow e'_1 \,/\, \mu'_1$ and $e \,/\, \mu \rightarrow e'_2 \,/\, \mu'_2$, then there exists a configuration $e' \,/\, \mu'$ such that $e'_1 \,/\, \mu'_1 \rightarrow^= e' \,/\, \mu'$ and $e'_2 \,/\, \mu'_2 \rightarrow^= e' \,/\, \mu'$.*

The semantics of Core ML can be obtained as a fragment of that of Core ML^2.

4.3 Relating Core ML^2 to Core ML

We now show that Core ML^2 is an appropriate tool to reason simultaneously about the execution of two Core ML programs. This is expressed by two properties. First, as explained above, the image of a valid reduction through projection remains a valid reduction. Conversely, if both projections of a term can be reduced to an outcome, then so can the term itself.

Lemma 4.2 *Let $i \in \{1, 2\}$. If $e \,/_i\, \mu \rightarrow e' \,/_i\, \mu'$, then $e \,/\, \lfloor \mu \rfloor_i \rightarrow e' \,/\, \lfloor \mu' \rfloor_i$.*

Proof. By inspection of (ref), (assign) and (deref). □

Lemma 4.3 (Soundness) *Let $i \in \{1, 2\}$. If $e \,/\, \mu \rightarrow e' \,/\, \mu'$, then $\lfloor e \,/\, \mu \rfloor_i \rightarrow^= \lfloor e' \,/\, \mu' \rfloor_i$.*

Proof. By inspection of the reduction rules and appeal to lemma 4.2. □

A configuration $e \,/_i\, \mu$ is *stuck* if it is irreducible and e isn't an outcome. It is *successful* if e is an outcome. The following lemma will be used in the proof of the completeness property.

Lemma 4.4 (Stuck Configurations) *If $e \,/\, \mu$ is stuck, then $\lfloor e \,/\, \mu \rfloor_i$ is stuck for some $i \in \{1, 2\}$.*

Proof. By induction on the structure of e.

∘ *Case* $e = v_1\,v_2$. Because neither (β) nor (lift-app) is applicable, v_1 cannot be of the form $\langle v_{11} \mid v_{12}\rangle$ or $\lambda\langle f\rangle x.e'$. As a result, for any $i \in \{1,2\}$, $\lfloor v_1\rfloor_i$ cannot be of the form $\lambda\langle f\rangle x.e'$. It follows that $\lfloor e \,/\, \mu\rfloor_i$ is stuck.

∘ *Case* $e = (v_1 := v_2)$, $e = {!}\,v$. Similar to the previous case.

∘ *Case* $e = \mathsf{ref}\ v$, $e = (\mathsf{let}\ x = v\ \mathsf{in}\ e')$. $e \,/\, \mu$ is not stuck.

∘ *Case* $e = \mathsf{raise}\ v$. Because e isn't an outcome, v isn't of the form $\varepsilon\, v'$. Because (lift-raise) isn't applicable, v isn't of the form $\langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2\rangle$. As a result, for some $i \in \{1,2\}$, $\lfloor v\rfloor_i$ cannot be of the form $\varepsilon\, v'$. It follows that $\lfloor e \,/\, \mu\rfloor_i$ is stuck.

∘ *Case* $e = E[e_1]$. By inspection of (bind), (handle), (handle-all) and (throw-context), one determines that if e_1 is an outcome, then $E[e_1]$ is reducible. So, e_1 is not an outcome, which implies that $e_1 \,/\, \mu$ is stuck. By induction hypothesis, $\lfloor e_1 \,/\, \mu\rfloor_i$ is stuck, for some $i \in \{1,2\}$. By inspection of the reduction rules, so is $F[\lfloor e_1\rfloor_i] \,/\, \lfloor\mu\rfloor_i$, for any evaluation context F; in particular, so is $\lfloor E[e_1]\rfloor_i \,/\, \lfloor\mu\rfloor_i$, which is $\lfloor e \,/\, \mu\rfloor_i$.

∘ *Case* $e = \langle e_1 \mid e_2\rangle$. Assume $e \,/\, \mu$ is stuck. By (bracket), both $e_1 \,/_1\, \mu$ and $e_2 \,/_2\, \mu$ are irreducible. Because e isn't an outcome, there exists $i \in \{1,2\}$ such that e_i isn't an outcome. As a result, $e_i \,/_i\, \mu$ is stuck. It follows that $e_i \,/\, \lfloor\mu\rfloor_i$ is stuck as well. □

Lemma 4.5 (Completeness) *Assume $\lfloor e \,/\, \mu\rfloor_i \to^\star o_i \,/\, \mu'_i$ for all $i \in \{1,2\}$. Then, there exists a configuration $o \,/\, \mu'$ such that $e \,/\, \mu \to^\star o \,/\, \mu'$ and, for all $i \in \{1,2\}$, $\lfloor o \,/\, \mu'\rfloor_i = o_i \,/\, \mu'_i$.*

Proof. Let us first establish that $e \,/\, \mu$ does not admit an infinite reduction sequence. To this end, let us first notice that no infinite reduction sequence can consist exclusively of instances of the "lift" reduction rules. (Indeed, each of these rules moves some $\langle\cdot \mid \cdot\rangle$ constructor strictly closer to the term's root.) Furthermore, these are the only rules which leave both projections of a configuration unchanged. In light of this remark, if $e \,/\, \mu$ admits an infinite reduction sequence, then lemma 4.3 yields an infinite reduction sequence out of $\lfloor e \,/\, \mu\rfloor_i$, for some $i \in \{1,2\}$. However, this is impossible, because both $\lfloor e \,/\, \mu\rfloor_1$ and $\lfloor e \,/\, \mu\rfloor_2$ can be reduced to normal forms, and the semantics of the Core ML fragment is deterministic.
As a result, $e \,/\, \mu$ can be reduced to an irreducible configuration. Let us now reason by induction on the number of steps in this reduction sequence.
First, assume the sequence is empty. Then, $e \,/\, \mu$ is irreducible. If $e \,/\, \mu$ is stuck, then so is one of its projections, by lemma 4.4. However, this is impossible, since both $\lfloor e \,/\, \mu\rfloor_1$ and $\lfloor e \,/\, \mu\rfloor_2$ can be reduced to successful configurations. So, $e \,/\, \mu$ must be successful. Then, so must its projections; the result follows.
Next, assume the sequence begins with $e \,/\, \mu \to e_0 \,/\, \mu_0$. By lemma 4.3, $\lfloor e \,/\, \mu\rfloor_i \to \lfloor e_0 \,/\, \mu_0\rfloor_i$ holds for all $i \in \{1,2\}$. By lemma 4.1, this implies $\lfloor e_0 \,/\, \mu_0\rfloor_i \to^\star o_i \,/\, \mu'_i$. Thus, we may apply the induction hypothesis to $e_0 \,/\, \mu_0$, showing that it reduces to some $o \,/\, \mu'$ such that $\lfloor o \,/\, \mu'\rfloor_i = o_i \,/\, \mu'_i$ holds for all $i \in \{1,2\}$. The result follows. □

Our completeness result requires both projections to converge; it is not applicable if one of them diverges. Indeed, define e as $\mathsf{bind}\ x = \langle\Omega \mid 0\rangle\ \mathsf{in}\ 0$, where Ω is a non-terminating expression. Its right projection is $\mathsf{bind}\ x = 0\ \mathsf{in}\ 0$, which reduces to 0; yet, e cannot be reduced to any term whose right projection is 0, because e only reduces to itself. Such a formulation of completeness will naturally lead us to establish a *weak* non-interference result, whereby two programs can be guaranteed to yield the same result only if they both terminate. We choose not to aim at a *strong* non-interference result, because it would make little sense to plug information leaks related to termination without attacking timing leaks in general. Furthermore, that would require a much more restrictive type system.
In essence, the completeness lemma guarantees that we have provided enough "lift" rules to allow reducing all meaningful Core ML^2 expressions. In the next section, each of these rules will add one case to our subject reduction proof, forcing us to ensure that our type system accounts for all possible kinds of information flow.

5 Typing Core ML^2

We now give a type system, called MLIF$_0$, for Core ML^2. It is a *ground* type system: it has no type variables and deals with polymorphism in a simple, abstract way. As a result, it does not describe an algorithm; we will address this issue in section 8.
Throughout the paper, every occurrence of $*$ stands for a distinct anonymous meta-variable of appropriate kind.

5.1 Types

Let $(\mathcal{L}, \leq)$ be a lattice whose elements, denoted by ℓ and pc, represent *security levels*. (Following Denning [Den82], we typically use the meta-variable pc, rather than ℓ, when considering information obtained by observing the value of the "program counter".) *Types*, *rows* and *alternatives* are defined as follows:

$$
\begin{array}{rcl}
t & ::= & \mathsf{unit} \\
 & | & \mathsf{int}^{\ell} \\
 & | & (t \xrightarrow{pc\ [r]} t)^{\ell} \\
 & | & t\ \mathsf{ref}^{\ell} \\
 & | & r\ \mathsf{exn}^{\ell} \\
r & ::= & \{\varepsilon \mapsto a\}_{\varepsilon \in \mathcal{E}} \\
a & ::= & \mathsf{Abs} \\
 & | & \mathsf{Pre}\ pc
\end{array}
$$

A *row* r is an infinite, quasi-constant family of alternatives indexed by $\mathcal{E}$. (A family is *quasi-constant* if all but a finite number of its entries are equal.) We write $(\varepsilon : a; r)$ for the row whose element at index ε is a and whose other elements are given by the sub-row r, which is indexed by $\mathcal{E} \setminus \{\varepsilon\}$. We write $a \in r$ to indicate that a is a member of r's codomain.

Our types are those of ML's type system, decorated with extra annotations of two kinds.

First, we use rows to keep track of exceptions, as in existing type-and-effect systems, such as Pessaux and Leroy's [PL00]. If an exception value has type $r\ \mathsf{exn}^*$, then the row r contains information about the exception's name. Specifically, for every $\varepsilon \in \mathcal{E}$, if $r(\varepsilon)$ is Abs, then the exception's name cannot be ε; if, on the other hand, it is $\mathsf{Pre}\ *$, then the exception may be named ε. Furthermore, function types carry an effect $[r]$. It is also a row, and gives a conservative description of all exceptions possibly raised by executing the function.

Second, we use security levels to keep track of how much information can be obtained by looking up integer values, executing functions, dereferencing memory locations, and handling exceptions. The remainder of this section describes their meaning.

Because there is only one value of type unit, the value of a unit expression yields no information whatsoever. As a result, it would be superfluous for the unit type constructor to carry a security level. Immutable tuple and record types can be dealt with similarly; see section 7.1. Thus, we break the convention set forth in a number of previous papers [HR98, PC00] that all types be of the form $*^{\ell}$. We expect this feature to help reduce verbosity in practice.

The type int^{ℓ} describes integer expressions whose value may reflect information of security level ℓ.

Function types carry two security annotations. The external annotation ℓ represents information about the function's identity. When the function is applied, part of this information may be reflected in its result or in other aspects of the function's behavior (i.e. in its effect); as a result, their security level will be made ℓ or greater. The annotation pc, found above the $\rightarrow$ symbol, tells how much information the function obtains merely by gaining control – indeed, observing that a particular function is called may allow telling which branches were previously taken. pc can be thought of as an extra parameter to the function, and indeed it is contravariant (see section 5.2). To avoid leaking this information, the function will be allowed to write into memory cells, or to raise exceptions, only at level pc or greater. This explains why the annotation pc is sometimes described as a lower bound on the level of the function's effects [HR98].

Reference types carry one annotation ℓ, which represents information about the reference's identity, i.e. about its address. Information about the reference's contents is found within the parameter t.

Exceptions are described by rows, within which every non-Abs entry, of the form $\varepsilon \mapsto \mathsf{Pre}\ pc$, carries an annotation pc, telling how much information will be obtained by observing (i.e. handling) the exception, if it is named ε. We follow Myers [Mye99a, Mye99b] and associate a distinct security level with every exception name, so as to obtain better precision. Our rows are closely related to Myers' sets of path labels X, which map every exception name to either a special constant $\underline{\emptyset}$ or a security level; compare these with our alternatives Abs and $\mathsf{Pre}\ pc$. (See section 10 for further comparison with [Mye99a, Mye99b].)

In addition to a row, exception types also carry an external annotation ℓ. It is, in fact, redundant with the row r. That is, manipulating an exception as a first-class value causes its external level ℓ to increase, leaving the row r unchanged; when the exception is later raised, every non-Abs entry in r is raised to level ℓ or greater. It would be possible to suppress the external annotation, at the cost of some extra implementation complexity. Another reasonable approach would be to restrict the language so that exceptions are no longer first-class values; this would allow us to do away with exn entirely.

$$\mathsf{int}^{\oplus} \qquad (\ominus \xrightarrow{\ominus\ [\oplus]} \oplus)^{\oplus} \qquad \odot\ \mathsf{ref}^{\oplus} \qquad \oplus\ \mathsf{exn}^{\oplus} \qquad \{\varepsilon \mapsto \oplus\}_{\varepsilon \in \mathcal{E}} \qquad \mathsf{Pre}\ \oplus \qquad \mathsf{Abs} \leq \mathsf{Pre}\ *$$

Figure 2: Subtyping

The reader may notice that rows do not record the type of exception arguments, i.e. the constructor Pre has no type parameter. Indeed, as in ML, we make exceptions monomorphic by assuming given a fixed mapping *typexn* from exception names to types. This decision is useful in two ways. First, it should make function types (which include a row) much more compact. Second, it makes our subtyping relation atomic (see section 5.2), which we believe opens the way to simpler and (in practice) more efficient constraint solving techniques.

5.2 Subtyping

We equip types, rows and alternatives with a subtyping relation $\leq$, which extends the partial order $(\mathcal{L}, \leq)$. It is defined by the axioms in figure 2. The axiom $\mathsf{int}^{\oplus}$ is a compact version of the assertion $\mathsf{int}^{\ell_1} \leq \mathsf{int}^{\ell_2} \iff \ell_1 \leq \ell_2$. In other words, it states that int's parameter is covariant. The other axioms are to be understood similarly; $\oplus$, $\ominus$ and $\odot$ represent covariant, contravariant and invariant parameters, respectively. The fifth axiom extends subtyping to rows, point-wise and covariantly.
The last axiom is the only one which relates two constructors of different arities, apparently making the subtyping relation non-atomic. However, it is only superficially so. Indeed, it is possible to give a presentation of the system where the set of alternatives is merely the disjoint union $\{\mathsf{Abs}\} \cup \mathcal{L}$, causing the explicit injection Pre to disappear, because security levels become a subset of alternatives. In this presentation, subtyping is atomic [Reh97]: alternatives form a set of atoms.
The use of subtyping in information flow control is ubiquitous [BL75, Den82, VS97, HR98] and appears essential, because it allows building a *directed* view of the program's information flow graph, yielding better precision than a unification-based analysis.

5.3 Additional notation

A *polytype* s is a non-empty, upward-closed set of types. A *polytype environment* Γ is a partial mapping from program variables to polytypes. $\Gamma[x \mapsto s]$ denotes the environment which maps x to s and agrees with Γ otherwise. A *memory environment* M is a partial mapping from memory locations to types.
We define $\ell \lhd t$ (read: ℓ *guards* t) as follows:

$$\frac{\ell \leq \ell'}{\begin{array}{c}\ell \lhd \mathsf{unit} \qquad \ell \lhd \mathsf{int}^{\ell'} \qquad \ell \lhd (* \xrightarrow{*\ [*]} *)^{\ell'} \\ \ell \lhd *\ \mathsf{ref}^{\ell'} \qquad \ell \lhd *\ \mathsf{exn}^{\ell'}\end{array}}$$

The assertion $\ell \lhd t$ requires t to have security level ℓ or greater, and is used to record a potential information flow. Note that, for any given ℓ and t, there exists a supertype t' of t such that $\ell \lhd t'$ holds. Thus, the presence of $\ell \lhd t$ as a premise typically never prevents the application of a typing rule: indeed, preceding that rule with a subtyping step will satisfy the premise. One exception is E-ASSIGN, where t cannot be promoted to a supertype because it appears as an invariant argument to the ref type constructor. The predicate $\lhd$ has transitive behavior:

Lemma 5.1 *If $\ell' \leq \ell$ and $\ell \lhd t$ and $t \leq t'$ then $\ell' \lhd t'$.*

> **Proof.** It is easy to see that $\ell \lhd t$ is equivalent to $\ell \leq \mathrm{level}(t)$ for some appropriately defined function level. The result follows. $\square$

To every row r, we associate two security levels, defined by $\sqcup r = \sqcup\{pc \mid \mathsf{Pre}\ pc \in r\}$ and $\sqcap r = \sqcap\{pc \mid \mathsf{Pre}\ pc \in r\}$. Note that Abs entries in r do not contribute to these levels.

5.4 Typing judgements

We distinguish two forms of typing judgements: one deals with values only, the other with arbitrary expressions. Because values are normal forms, they have no side effects, so the former look quite simple:

$$\Gamma, M \vdash v : t$$

(We also write $\Gamma, M \vdash v : s$ when $\Gamma, M \vdash v : t$ holds for all $t \in s$.) On the other hand, expressions do produce side effects, so the latter are more elaborate:

$$pc, \Gamma, M \vdash v : t \;\; [r]$$

The pc parameter again tells how much information the expression may acquire by gaining control; it is a lower bound on the level of the expression's effects. Previous works [VS97, HR98] employ a similar parameter. The row r approximates the set of exceptions which the expression may raise.
Two extra judgement forms are employed to type stores: $M \vdash \mu$ and configurations: $\Gamma \vdash e /_i \mu : t \;\; [r]$.
In typing judgements, we omit Γ and M when they are empty; we sometimes omit pc and r when they are unspecified (i.e. when they could be written $*$).
Even though the security lattice $(\mathcal{L}, \leq)$ is arbitrary, it is desirable to establish a simple dichotomy between "low" and "high" security levels. Such a distinction simplifies our proofs; full generality will be recovered in section 6. In the present section, we assume given a fixed, upward-closed set $H \subseteq \mathcal{L}$. We will view levels inside (resp. outside) H as "high" (resp. "low").
Non-interference demands that two expressions which differ only in high-level sub-terms have identical low-level behavior. To achieve this, our type system requires expressions of the form $\langle e_1 \mid e_2 \rangle$ – which we use to encode the differences between two Core ML expressions – to have high-security result and side effects. (See v-Bracket and e-Bracket in figure 3.) This will be our only use of H in this section.

5.5 Typing rules

We now comment on the typing rules, given in figure 3. v-Unit and v-Int assign base types to constants. v-Void allows typing values of the form $\langle v \mid \mathsf{void} \rangle$ or $\langle \mathsf{void} \mid v \rangle$ by pretending void has the same type as v. v-Loc and v-Var assign types to memory locations and to variables by looking up the appropriate environment. Note that $\Gamma(x)$ is a polytype, of which v-Var selects an arbitrary instance. As usual in type-and-effect systems, v-Abs records, on top of the $\rightarrow$ type constructor, information about a function's side effects. v-Exn associates to the exception value $\varepsilon\, v$ a row which maps the name ε to Pre $*$ and leaves other entries unconstrained, allowing them to be Abs. v-Bracket requires the components of a $\langle \cdot \mid \cdot \rangle$ construct to have a common type, which must have "high" security level, i.e. be guarded by some (arbitrary) element of H. v-Sub is standard.
e-Value allows viewing a value as an expression, and reflects the fact that values have no side effect.
e-App governs function application. Because the effect of a function application is exactly the function's latent effect, the security level pc, which should represent a lower bound on the level of the former, must also be a lower bound on the latter's. Because a function's side effects may reveal information about its identity, their level must equal or exceed the function's own security level, namely ℓ. As a result of these remarks, the function's body must run at level $pc \sqcup \ell$. Because the function's result, too, may reveal information about its identity, we require its type to be guarded by ℓ.
e-Ref and e-Assign require $pc \lhd t$ to ensure that pc is indeed a lower bound on the security level of the memory cell that is written. e-Assign and e-Deref require $\ell \lhd t$ to reflect the fact that writing or reading a cell may indirectly reveal information about its identity.
e-Raise requires $pc \leq \sqcap r$, ensuring that pc is a lower bound on the level of every non-Abs entry in the row r. Thus, any code fragment able to observe this expression's side effect will have to run at level pc or greater (see e-Bind, e-Handle and e-HandleAll). The security level ℓ, which reflects additional, exception-name-independent information, is dealt with similarly.
Because the let construct only binds values, e-Let is nearly as simple as in ML. Note that v can be given a polytype s, allowing x to be used at different types within e.
In a binding construct bind $x = e_1$ in e_2, the expression e_2 observes, if it receives control, that no exception was raised by e_1. To account for this information channel, e-Bind typechecks e_2 at a security level augmented with $\sqcup r_1$, the combined level of all exceptions which e_1 can potentially raise. This is a conservative approximation, which works well in the common case where e_1 is statically known never to raise exceptions; see section 10 for details. $r_1 \sqcup r_2$ denotes the least common supertype of r_1 and r_2.

Values

$$\textsc{v-Unit}\quad \Gamma, M \vdash () : \mathsf{unit}$$

$$\textsc{v-Int}\quad \Gamma, M \vdash k : \mathsf{int}^*$$

$$\textsc{v-Void}\quad \Gamma, M \vdash \mathsf{void} : *$$

$$\textsc{v-Loc}\quad \Gamma, M \vdash m : M(m)\ \mathsf{ref}^*$$

$$\textsc{v-Var}\quad \frac{t \in \Gamma(x)}{\Gamma, M \vdash x : t}$$

$$\textsc{v-Abs}\quad \frac{pc, \Gamma[x \mapsto t'][f \mapsto (t' \xrightarrow{pc\ [r]} t)^\ell], M \vdash e : t\ [r]}{\Gamma, M \vdash \lambda\langle f\rangle x.e : (t' \xrightarrow{pc\ [r]} t)^\ell}$$

$$\textsc{v-Exn}\quad \frac{\Gamma, M \vdash v : \mathit{typexn}(\varepsilon)}{\Gamma, M \vdash \varepsilon\, v : (\varepsilon : \mathsf{Pre}\ *; *)\ \mathsf{exn}^*}$$

$$\textsc{v-Bracket}\quad \frac{\Gamma, M \vdash v_1 : t \qquad \Gamma, M \vdash v_2 : t \qquad pc' \in H \qquad pc' \lhd t}{\Gamma, M \vdash \langle v_1 \mid v_2\rangle : t}$$

$$\textsc{v-Sub}\quad \frac{\Gamma, M \vdash v : t' \qquad t' \leq t}{\Gamma, M \vdash v : t}$$

Expressions

$$\textsc{e-Value}\quad \frac{\Gamma, M \vdash v : t}{*, \Gamma, M \vdash v : t\ [*]}$$

$$\textsc{e-App}\quad \frac{\Gamma, M \vdash v_1 : (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell \qquad \Gamma, M \vdash v_2 : t' \qquad \ell \lhd t}{pc, \Gamma, M \vdash v_1\, v_2 : t\ [r]}$$

$$\textsc{e-Ref}\quad \frac{\Gamma, M \vdash v : t \qquad pc \lhd t}{pc, \Gamma, M \vdash \mathsf{ref}\ v : t\ \mathsf{ref}^*\ [*]}$$

$$\textsc{e-Assign}\quad \frac{\Gamma, M \vdash v_1 : t\ \mathsf{ref}^\ell \qquad \Gamma, M \vdash v_2 : t \qquad pc \sqcup \ell \lhd t}{pc, \Gamma, M \vdash v_1 := v_2 : \mathsf{unit}\ [*]}$$

$$\textsc{e-Deref}\quad \frac{\Gamma, M \vdash v : t'\ \mathsf{ref}^\ell \qquad t' \leq t \qquad \ell \lhd t}{pc, \Gamma, M \vdash\ !v : t\ [*]}$$

$$\textsc{e-Raise}\quad \frac{\Gamma, M \vdash v : r\ \mathsf{exn}^\ell \qquad pc \sqcup \ell \leq \sqcap r}{pc, \Gamma, M \vdash \mathsf{raise}\ v : *\ [r]}$$

$$\textsc{e-Let}\quad \frac{\Gamma, M \vdash v : s \qquad pc, \Gamma[x \mapsto s], M \vdash e : t\ [r]}{pc, \Gamma, M \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : t\ [r]}$$

$$\textsc{e-Bind}\quad \frac{pc, \Gamma, M \vdash e_1 : t'\ [r_1] \qquad pc \sqcup (\sqcup r_1), \Gamma[x \mapsto t'], M \vdash e_2 : t\ [r_2]}{pc, \Gamma, M \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : t\ [r_1 \sqcup r_2]}$$

$$\textsc{e-Handle}\quad \frac{pc, \Gamma, M \vdash e_1 : t\ [\varepsilon : \mathsf{Pre}\ pc'; r] \qquad pc \sqcup pc', \Gamma[x \mapsto \mathit{typexn}(\varepsilon)], M \vdash e_2 : t\ [\varepsilon : a; r] \qquad pc' \lhd t}{pc, \Gamma, M \vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : t\ [\varepsilon : a; r]}$$

$$\textsc{e-HandleAll}\quad \frac{pc, \Gamma, M \vdash e_1 : t\ [r_1] \qquad pc \sqcup (\sqcup r_1), \Gamma[x \mapsto r_1\ \mathsf{exn}^*], M \vdash e_2 : t\ [r_2] \qquad (\sqcup r_1) \lhd t}{pc, \Gamma, M \vdash e_1\ \mathsf{handle}\ x \succ e_2 : t\ [r_2]}$$

$$\textsc{e-Bracket}\quad \frac{pc \sqcup pc', \Gamma, M \vdash e_1 : t\ [r] \qquad pc \sqcup pc', \Gamma, M \vdash e_2 : t\ [r] \qquad pc' \in H \qquad (pc' \lhd t) \vee (e_1 \Uparrow) \vee (e_2 \Uparrow)}{pc, \Gamma, M \vdash \langle e_1 \mid e_2\rangle : t\ [r]}$$

$$\textsc{e-Sub}\quad \frac{pc, \Gamma, M \vdash e : t'\ [r'] \qquad t' \leq t \qquad r' \leq r}{pc, \Gamma, M \vdash e : t\ [r]}$$

Configurations

$$\textsc{Store}\quad \frac{\mathrm{dom}(M) = \mathrm{dom}(\mu) \qquad \forall m \in \mathrm{dom}(\mu) \quad M \vdash \mu(m) : M(m)}{M \vdash \mu}$$

$$\textsc{Conf}\quad \frac{pc, \Gamma, M \vdash e : t\ [r] \qquad M \vdash \mu}{\Gamma \vdash e \,/\, \mu : t\ [r]}$$

Figure 3: The type system $\textsc{mlif}_0$

Like E-BIND, E-HANDLE typechecks e_2 at an increased security level, reflecting the fact that, by gaining control, e_2 observes that e_1 raised an exception named ε. The increment is exactly pc', the security level associated with ε in e_1's effect, so the analysis is, in this case, quite accurate. Because the result of the handle construct may also allow determining whether the handler was executed, we require $pc' \lhd t$. E-HANDLEALL is analogous; however, because the construct allows observing any exception, regardless of its name, we again use $\sqcup r_1$ as a conservative approximation of how much information is gained. Myers [Mye99a, Mye99b] performs the same approximation.

As explained earlier, E-BRACKET requires both components of a $\langle \cdot \mid \cdot \rangle$ expression to have a common type, and demands that its side effects and its result be of "high" security level, i.e. guarded by an arbitrary $pc' \in H$. The auxiliary predicate $e\Uparrow$ holds if and only if e is of the form $E_1[\ldots E_n[\mathsf{raise}\ (\varepsilon\, v)]\ldots]$ where $n \geq 0$ and none of the E_i handles $\mathsf{raise}\ (\varepsilon\, v)$. The use of this predicate in E-BRACKET's last premise is technical; it is required for subject reduction to hold.

5.6 Subject reduction

Let us first state a few auxiliary lemmas, whose proofs are straightforward.

Lemma 5.2 (Subsumption) *$pc' \leq pc$ and $pc, \Gamma, M \vdash e : t\ [r]$ imply $pc', \Gamma, M \vdash e : t\ [r]$.*

> **Proof.** By induction on the derivation of $pc, \Gamma, M \vdash e : t\ [r]$. By monotonicity of $\sqcup$, contravariance of $\rightarrow$ with respect to its pc parameter, rule V-SUB, lemma 5.1, and the induction hypothesis, it is easy to check that every premise remains valid when pc decreases. The result follows. ☐

Lemma 5.3 (Projection) *Let $i \in \{1, 2\}$. If $\Gamma, M \vdash v : t$ then $\Gamma, M \vdash \lfloor v \rfloor_i : t$. If $pc, \Gamma, M \vdash e : t\ [r]$ then $pc, \Gamma, M \vdash \lfloor e \rfloor_i : t\ [r]$.*

> **Proof.** By induction on the input derivation. The only case of interest is that of E-BRACKET, where the expression at hand is $\langle e_1 \mid e_2 \rangle$. Then, one of the first two premises is $pc \sqcup pc', \Gamma, M \vdash e_i : t\ [r]$. Lemma 5.2 yields $pc, \Gamma, M \vdash e_i : t\ [r]$, as required. ☐

Lemma 5.4 (Guard) *If $\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t$ then there exists $pc' \in H$ such that $pc' \lhd t$.*

> **Proof.** Thanks to lemma 5.1, we may assume, w.l.o.g., that the derivation of $\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t$ does not end with an instance of V-SUB. Thus, it must end with an instance of V-BRACKET, among whose premises we find $pc' \lhd t$ and $pc' \in H$. ☐

Lemma 5.5 (Store access) *Let i be in $\{\bullet, 1, 2\}$. Assume $\Gamma, M \vdash v : t$ and $\Gamma, M \vdash v' : t$. Then, $\Gamma, M \vdash \mathrm{read}_i\, v : t$ holds. Moreover, if $i \in \{1, 2\}$, assume there exists some $pc' \in H$ such that $pc' \lhd t$. Then, $\Gamma, M \vdash \mathrm{new}_i\, v : t$ and $\Gamma, M \vdash \mathrm{update}_i\, v\, v' : t$ hold.*

> **Proof.** By definition of the functions new, update and read (figure 1), by lemma 5.3, by V-VOID and V-BRACKET. ☐

Lemma 5.6 (Substitution) *Assume $M \vdash v : s$. Then, $\Gamma[x \mapsto s], M \vdash v' : t$ implies $\Gamma, M \vdash v'[x \Leftarrow v] : t$. Also, $pc, \Gamma[x \mapsto s], M \vdash e : t\ [r]$ implies $pc, \Gamma, M \vdash e[x \Leftarrow v] : t\ [r]$.*

> **Proof.** By induction on the input derivation.
>
> ◦ *Case* V-VAR. If v' is x, then the premise is $t \in s$. Thus, the hypothesis $M \vdash v : s$ implies $M \vdash v : t$, and, a fortiori, $\Gamma, M \vdash v : t$. Considering $v'[x \Leftarrow v] = v$, this was the goal. If, on the other hand, v' isn't x, then the result stems from $\Gamma[x \mapsto s](v') = \Gamma(v')$ and $v'[x \Leftarrow v] = v'$.
>
> ◦ *Case* V-ABS. Then, the premise must be of the form $pc', \Gamma[x \mapsto s][y \mapsto t'][f \mapsto t_f], M \vdash e' : t''\ [r']$. Because typing judgements are stable under α-conversion, we will assume, w.l.o.g., that x, f and y are distinct. Then, $\Gamma[x \mapsto s][y \mapsto t'][f \mapsto t_f]$ coincides with $\Gamma[y \mapsto t'][f \mapsto t_f][x \mapsto s]$. We conclude by applying the induction hypothesis, followed by an instance of V-ABS.
>
> ◦ *Case* V-BRACKET. The first premise is of the form $\Gamma[x \mapsto s], M \vdash v'_1 : t$. By lemma 5.3, the hypothesis $M \vdash v : s$ implies $M \vdash \lfloor v \rfloor_1 : s$. Thus, by induction hypothesis, $\Gamma, M \vdash v'_1[x \Leftarrow \lfloor v \rfloor_1] : t$ holds. The second premise is dealt with similarly. By V-BRACKET, we obtain $\Gamma, M \vdash \langle v'_1[x \Leftarrow \lfloor v \rfloor_1] \mid v'_2[x \Leftarrow \lfloor v \rfloor_2] \rangle : t$, which, considering our definition of substitution (section 4.2), was our goal.
>
> Other cases are either immediate or analogous to those above. ☐

Lemma 5.7 (Value) *$pc, \Gamma, M \vdash v : t\ [r]$ implies $\Gamma, M \vdash v : t$.*

Proof. By induction on the proof of $pc, \Gamma, M \vdash v : t\ [r]$.

∘ *Case* E-VALUE. Immediate.

∘ *Case* E-SUB. The result follows from the induction hypothesis and V-SUB.

∘ *Case* E-BRACKET. The predicate $\cdot\Uparrow$ is never true of a value, so $pc' \lhd t$ must hold. The result follows from the induction hypothesis and V-BRACKET. □

We can now state our main lemma:

Lemma 5.8 (Subject reduction) *Let $e /_i \mu \to e' /_i \mu'$. Assume $pc, M \vdash e : t\ [r]$ and $M \vdash \mu$. If $i \in \{1, 2\}$, assume $pc \in H$. Then, there exists a memory environment M', which extends M, such that $pc, M' \vdash e' : t\ [r]$ and $M' \vdash \mu'$.*

Proof. By induction on the derivation of $e /_i \mu \to e' /_i \mu'$. We assume, w.l.o.g., that the derivation of $pc, M \vdash e : t\ [r]$ does not end with an instance of E-SUB. As a result, it must end with an instance of the single syntax-directed rule that matches e's structure.

∘ *Case* (β). e is $(\lambda\langle f\rangle x.e_0)\, v$. Define $\theta = (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell$. By E-APP, we have $M \vdash \lambda\langle f\rangle x.e_0 : \theta$ and $M \vdash v : t'$. The former's derivation must end with an instance of V-ABS, followed by a number of instances of V-SUB. Because $\to$ is contravariant (resp. covariant) in its first and second (resp. third and fourth) parameters, applying lemma 5.2 and E-SUB to V-ABS's premise yields $pc, (x \mapsto t''; f \mapsto \theta'), M \vdash e_0 : t\ [r]$, for some t'' and θ' such that $t' \leq t''$ and $\theta \leq \theta'$. By V-SUB, $M \vdash v : t''$ and $M \vdash \lambda\langle f\rangle x.e_0 : \theta'$ hold. Then, lemma 5.6 yields $pc, M \vdash e_0[x \Leftarrow v][f \Leftarrow \lambda\langle f\rangle x.e_0] : t\ [r]$.

∘ *Case* (ref). e is $\mathsf{ref}\ v$, e' is m and μ' is $\mu \oplus [m \mapsto \mathrm{new}_i\, v]$. By E-REF, we have $M \vdash v : t'$ and $pc \lhd t'$ and $t = t'\ \mathsf{ref}^*$. By lemma 5.5, these imply $M \vdash \mathrm{new}_i\, v : t'$. Define $M' = M[m \mapsto t']$. By STORE, $M \vdash \mu$ yields $\mathrm{dom}(M) = \mathrm{dom}(\mu)$. Because $\mu \oplus [m \mapsto v]$ is defined, m isn't a member of $\mathrm{dom}(\mu)$. So, M' extends M. Because $M'(m) = t'$, V-LOC and E-VALUE yield $pc, M' \vdash e' : t\ [r]$. Lastly, $M \vdash \mu$ and $M \vdash \mathrm{new}_i\, v : t'$ entail $M' \vdash \mu'$.

∘ *Case* (assign). e is $m := v$ and e' is $()$. By E-ASSIGN, we must have $M \vdash m : t'\ \mathsf{ref}^*$ and $M \vdash v : t'$ and $pc \lhd t'$. Furthermore, t must be unit, which implies $pc, M \vdash e' : t\ [r]$. By V-LOC, V-SUB and by invariance of the ref type constructor, $M \vdash m : t'\ \mathsf{ref}^*$ implies $M(m) = t'$. Thus, $M \vdash \mu$ entails $M \vdash \mu(m) : t'$. By lemma 5.5, we have $M \vdash \mathrm{update}_i\, \mu(m)\, v : t'$, which yields $M \vdash \mu'$.

∘ *Case* (deref). e is $!m$. By E-DEREF, we have $M \vdash m : t'\ \mathsf{ref}^*$, where $t' \leq t$. As above, this entails $M \vdash \mu(m) : t'$. By lemma 5.5, $M \vdash \mathrm{read}_i\, \mu(m) : t'$ follows. Conclude with V-SUB and E-VALUE.

∘ *Case* (let). By E-LET and lemma 5.6.

∘ *Case* (bind). e is $\mathsf{bind}\ x = v\ \mathsf{in}\ e_2$ and e' is $e_2[x \Leftarrow v]$. By E-BIND, we have $pc, M \vdash v : t'\ [r_1]$ and $pc \sqcup (\sqcup r_1), (x \mapsto t'), M \vdash e_2 : t\ [r_2]$, where $r_2 \leq r$. By lemma 5.7, the former implies $M \vdash v : t'$. By lemma 5.2, the latter implies $pc, (x \mapsto t'), M \vdash e_2 : t\ [r_2]$. By lemma 5.6 and E-SUB, we obtain $pc, M \vdash e_2[x \Leftarrow v] : t\ [r]$.

∘ *Case* (handle). e is $\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ \varepsilon\, x \succ e_2$ and e' is $e_2[x \Leftarrow v]$. By E-HANDLE, we have $pc, M \vdash \mathsf{raise}\ (\varepsilon\, v) : t\ [\varepsilon : \mathsf{Pre}\ pc'; r']$ and $pc \sqcup pc', (x \mapsto \mathit{typexn}(\varepsilon)), M \vdash e_2 : t\ [\varepsilon : a; r']$, where r equals $(\varepsilon : a; r')$. By E-SUB, E-RAISE, V-SUB and V-EXN, the former yields $M \vdash v : \mathit{typexn}(\varepsilon)$. By lemmas 5.6 and 5.2, this yields $pc, M \vdash e_2[x \Leftarrow v] : t\ [r]$.

∘ *Case* (handle-all). e is $\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e_2$ and e' is $e_2[x \Leftarrow \varepsilon\, v]$. By E-HANDLEALL, $pc, M \vdash \mathsf{raise}\ (\varepsilon\, v) : t\ [r_1]$ and $pc \sqcup (\sqcup r_1), (x \mapsto r_1\ \mathsf{exn}^\ell), M \vdash e_2 : t\ [r]$ hold. By E-SUB, E-RAISE, V-SUB and V-EXN, and by covariance of the exn type constructor, the former yields $M \vdash \varepsilon\, v : r_1\ \mathsf{exn}^\ell$. Lemmas 5.6 and 5.2 yield $pc, M \vdash e_2[x \Leftarrow \varepsilon\, v] : t\ [r]$.

∘ *Case* (throw-context). e is $E[o]$ and e' is o. Several sub-cases arise.

Sub-case$E = \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e_2$. By E-BIND, we must have $pc, M \vdash o : t'\ [r_1]$, where $r_1 \leq r$. Because o must be of the form $\mathsf{raise}\ (\varepsilon\, v)$ or $\langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid \mathsf{raise}\ (\varepsilon_2\, v_2)\rangle$, this judgement must be a consequence of E-RAISE, E-BRACKET and E-SUB. A derivation of identical shape can be built to establish $pc, M \vdash o : t\ [r_1]$. (In the case of E-BRACKET, the fourth premise is satisfied, though its first disjunct may be false, because the other two hold.) The result follows by E-SUB.

Sub-case$E = [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e_2$. By E-HANDLE, we have $pc, M \vdash o : t\ [\varepsilon : \mathsf{Pre}\ *; r']$. o must be of the form v or $\mathsf{raise}\ (\varepsilon'\, v)$ or $\langle v_1 \mid \mathsf{raise}\ (\varepsilon_2\, v_2)\rangle$ or $\langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid v_2\rangle$ or $\langle \mathsf{raise}\ (\varepsilon_1\, v_1) \mid \mathsf{raise}\ (\varepsilon_2\, v_2)\rangle$, where ε', ε_1 and ε_2 are distinct from ε. As a result, a derivation of identical shape can be built to establish $pc, M \vdash o : t\ [\varepsilon : a; r']$, that is, $pc, M \vdash o : t\ [r]$.

Sub-case$E = [\,]\ \mathsf{handle}\ x \succ e_2$. By E-HANDLEALL, $pc, M \vdash o : t\ [r_1]$ holds. Because o must be a value, a derivation of identical shape yields $pc, M \vdash o : t\ [r]$.

∘ *Case* (lift-app). e is $\langle v_1 \mid v_2\rangle\, v$. Let $\theta = (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell$. E-APP's premises are $M \vdash \langle v_1 \mid v_2\rangle : \theta$ and $M \vdash v : t'$ and $\ell \lhd t$. Lemma 5.3 yields $M \vdash v_i : \theta$ and $M \vdash \lfloor v \rfloor_i : t'$, for $i \in \{1,2\}$. Then, E-APP yields $pc \sqcup \ell, M \vdash v_i\, \lfloor v \rfloor_i : t\ [r]$. Furthermore, applying lemma 5.4 to the first premise above and recalling that H is upward-closed yields $\ell \in H$. Because $\ell \lhd t$, E-BRACKET is applicable and yields $pc, M \vdash e' : t\ [r]$.

∘ *Case* (lift-assign). e is $\langle v_1 \mid v_2\rangle := v$. E-ASSIGN's premises are $M \vdash \langle v_1 \mid v_2\rangle : t'\ \mathsf{ref}^\ell$ and $M \vdash v : t'$ and $pc \sqcup \ell \lhd t'$. As above, applying lemma 5.3 and building new instances of E-ASSIGN, we obtain $pc \sqcup \ell, M \vdash v_i := \lfloor v \rfloor_i : t\ [r]$, for $i \in \{1,2\}$. Similarly, lemma 5.4 allows establishing $\ell \in H$. The result follows by E-BRACKET.

∘ *Case* (lift-deref). e is $!\langle v_1 \mid v_2\rangle$. E-DEREF's premises are $M \vdash \langle v_1 \mid v_2\rangle : t'\ \mathsf{ref}^\ell$ and $t' \leq t$ and $\ell \lhd t$. As above, applying lemma 5.3 and building new instances of E-DEREF, we obtain $pc \sqcup \ell, M \vdash\ !v_i : t\ [r]$, for $i \in \{1,2\}$. Similarly, lemma 5.4 yields $\ell \in H$. Lastly, by E-BRACKET, we obtain $pc, M \vdash \langle !v_1 \mid !v_2\rangle : t\ [r]$.

∘ *Case* (lift-raise). e is $\mathsf{raise}\ \langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2\rangle$. E-RAISE's premises are $M \vdash \langle \varepsilon_1\, v_1 \mid \varepsilon_2\, v_2\rangle : r\ \mathsf{exn}^\ell$ and $pc \sqcup \ell \leq \sqcap r$. As above, lemma 5.3 and E-RAISE yield $pc \sqcup \ell, M \vdash \mathsf{raise}\ (\varepsilon_i\, v_i) : t\ [r]$, for $i \in \{1,2\}$, while lemma 5.4 yields $\ell \in H$. The result follows by E-BRACKET, whose fourth premise is satisfied, though $\ell \lhd t$ may be false, because the other disjuncts hold.

∘ *Case* (lift-context). e is $E[\langle o_1 \mid o_2\rangle]$. If E is a bind context, then, because e cannot be reduced by (bind), $\langle o_1 \mid o_2\rangle$ cannot be a value. If, on the other hand, E is a handle context, then, because (throw-context) isn't applicable, E must handle o_1 or o_2. In either case, we conclude that o_i is of the form $\mathsf{raise}\ (\varepsilon\, v)$, for some $i \in \{1,2\}$. Now, e's typing derivation must end with an instance of E-BIND, E-HANDLE or E-HANDLEALL, whose first premise is of the form $pc, M \vdash \langle o_1 \mid o_2\rangle : t'\ [r_1]$. Because $\langle o_1 \mid o_2\rangle$ isn't a value, this must be a consequence of E-SUB and E-BRACKET, which yields $pc \sqcup \ell, M \vdash o_i : t'\ [r_1]$, for some $\ell \in H$ and for $i \in \{1,2\}$. By E-RAISE, V-SUB and V-EXN, this implies $\mathsf{Pre}\ \ell \leq r_1(\varepsilon)$ and $\ell \leq \sqcup r_1$; thus, the security assumption in E-BIND, E-HANDLE or E-HANDLEALL's second premise is greater than or equal to ℓ. As a result, by applying lemma 5.3 to that premise, then building new instances of E-BIND, E-HANDLE or E-HANDLEALL, we obtain $pc \sqcup \ell, M \vdash \lfloor E \rfloor_i[o_i] : t\ [r]$, for $i \in \{1,2\}$. There remains to apply E-BRACKET. If E is a bind context, then $\lfloor E \rfloor_i[o_i]\Uparrow$ holds for some $i \in \{1,2\}$; if, on the other hand, E is a handle context, then $\ell \lhd t$ holds, according to E-HANDLE or E-HANDLEALL's third premise. In either case, E-BRACKET's fourth premise holds.

∘ *Case* (bracket). e is $\langle e_1 \mid e_2\rangle$ and e' is $\langle e_1' \mid e_2'\rangle$. We have $e_i \,/_i\, \mu \to e_i' \,/_i\, \mu'$ and $e_j = e_j'$, where $\{i,j\} = \{1,2\}$. Because $\langle e_1 \mid e_2\rangle$ isn't a value, its typing derivation must end with an instance of E-BRACKET, whose first two premises are $pc \sqcup pc', M \vdash e_i : t\ [r]$ and $pc \sqcup pc', M \vdash e_j : t\ [r]$. Because $pc' \in H$, the induction hypothesis is applicable, yielding a memory environment M', which extends M, such that $pc \sqcup pc', M' \vdash e_i' : t\ [r]$ and $M' \vdash \mu'$. Because M' extends M, $pc \sqcup pc', M' \vdash e_j : t\ [r]$ holds as well. The result follows by E-BRACKET.

∘ *Case* (context). e is $E[e_0]$ and e' is $E[e_0']$, where $e_0 \,/_i\, \mu \to e_0' \,/_i\, \mu'$. Applying the induction hypothesis to E-BIND, E-HANDLE or E-HANDLEALL's first premise yields a version of it with M and e_0 replaced with M' and e_0', where M' extends M and $M' \vdash \mu'$ holds. Because M extends M', the second premise remains valid when the former is replaced with the latter. Build a new instance of E-BIND, E-HANDLE or E-HANDLEALL to conclude. □

The previous lemma entails the following, more abstract statement:

Theorem 5.1 (Subject reduction) *If* $\vdash e \,/\, \mu : t\ [r]$ *and* $e \,/\, \mu \to e' \,/\, \mu'$ *then* $\vdash e' \,/\, \mu' : t\ [r]$.

Proof. By CONF and lemma 5.8. □

We do not establish *progress* (i.e. "no well-typed configuration is stuck"), even though it does hold, because it is unrelated to our concerns.

5.7 On evaluation order

As explained in section 3, our restricted syntax is fully explicit about evaluation order. In practice, it is possible to use more permissive syntax, provided some evaluation strategy is fixed. For instance, if left-to-right evaluation order is chosen, then $e_1\, e_2$ (the application of an expression to another expression) is syntactic sugar for $\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ \mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ x_1\, x_2$. This gives rise to the following derived typing rule:

$$\frac{pc, \Gamma, M \vdash e_1 : (t' \xrightarrow{pc \sqcup \ell \sqcup (\sqcup r_1) \sqcup (\sqcup r_2)\ [r]} t)^\ell\ [r_1] \qquad pc \sqcup (\sqcup r_1), \Gamma, M \vdash e_2 : t'\ [r_2] \qquad \ell \lhd t}{pc, \Gamma, M \vdash e_1\, e_2 : t\ [r \sqcup r_1 \sqcup r_2]}$$

Conversely, under right-to-left evaluation order, $e_1\, e_2$ is encoded as bind $x_2 = e_2$ in bind $x_1 = e_1$ in $x_1\, x_2$, yielding a different derived rule:

$$\frac{pc \sqcup (\sqcup r_2), \Gamma, M \vdash e_1 : (t' \xrightarrow{pc \sqcup \ell \sqcup (\sqcup r_1) \sqcup (\sqcup r_2)\ [r]} t)^\ell\ [r_1] \qquad pc, \Gamma, M \vdash e_2 : t'\ [r_2] \qquad \ell \lhd t}{pc, \Gamma, M \vdash e_1\, e_2 : t\ [r \sqcup r_1 \sqcup r_2]}$$

In either case, the second expression to be evaluated is typechecked at an increased security level, reflecting the fact that, by receiving control, it is able to observe that the expression which was executed first terminated normally.

Caml-Light [LD+97] does not specify its evaluation order. It is possible to give a conservative typing rule which is safe with respect to both left-to-right and right-to-left evaluation orders. Such a rule typechecks e_i under $pc \sqcup (\sqcup r_j)$, for $\{i, j\} = \{1, 2\}$. Because exceptions are annotated with the value of pc at the point where they are raised, and because pc can only increase within sub-expressions, this typically entails $\sqcup r_j \leq \sqcap r_i$. Furthermore, for every row r with at least one non-Abs entry, $\sqcap r \leq \sqcup r$ holds. As a result, if e_i is liable to raise some exception, then all exceptions in r_j must have the same security level. Thus, under-specifying the evaluation order causes an important loss of precision in our analysis. Caml-Light's current implementation uses a right-to-left evaluation strategy; for our purposes, this should be made part of its specification.

6 Non-interference

From here on, the set H is no longer fixed. We introduce it explicitly when needed, writing $\vdash_H$ instead of $\vdash$ in Core ML^2 typing judgements. (This is not necessary for those judgements which involve plain Core ML expressions, because H is used only in V-BRACKET and E-BRACKET.) We write $e \rightarrow^\star o$ if there exists a store μ such that $e \,/\, \varnothing \rightarrow^\star o \,/\, \mu$, where $\varnothing$ is the empty store.

Our type system keeps track of $\langle \cdot \mid \cdot \rangle$ constructs by assigning them "high" security levels (i.e. levels in H). By subject reduction, any expression which may evaluate to such a construct must also carry a "high" annotation. Conversely, no expression with a "low" annotation can evaluate to such a construct, as stated, in the particular case of integers, by the following lemma:

Lemma 6.1 *Let H be an upward-closed subset of $\mathcal{L}$. Let $\ell \notin H$. If $\vdash_H e : \mathsf{int}^\ell$ and $e \rightarrow^\star v$ then $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$.*

Proof. By theorem 5.1 and CONF, there exists a memory environment M such that $M \vdash_H v : \mathsf{int}^\ell\ [*]$ holds. A value of type int^* must be of the form k or $\langle k_1 \mid k_2 \rangle$. If the latter, then, by V-BRACKET or E-BRACKET, there exists $pc' \in H$ such that $pc' \leq \ell$, which implies $\ell \in H$, a contradiction. Thus, we must have $v = k = \lfloor v \rfloor_1 = \lfloor v \rfloor_2$. □

We can now use the correspondence between Core ML and Core ML^2 developed in section 4.3 to reformulate this result in a Core ML setting:

Theorem 6.1 (Non-interference) *Choose $\ell, h \in \mathcal{L}$ such that $h \not\leq \ell$. Let $h \lhd t$. Assume $(x \mapsto t) \vdash e : \mathsf{int}^\ell$, where e is a Core ML expression. If $\vdash v_i : t$ and $e[x \Leftarrow v_i] \rightarrow^\star v'_i$, for $i \in \{1, 2\}$, then $v'_1 = v'_2$.*

Proof. Let $H = \uparrow\{h\}$. Define $v = \langle v_1 \mid v_2 \rangle$. By V-BRACKET, $\vdash_H v : t$ holds. Lemma 5.6 yields $\vdash_H e[x \Leftarrow v] : \mathsf{int}^\ell$. Now, $\lfloor e[x \Leftarrow v] \rfloor_i$ is $e[x \Leftarrow v_i]$, which, by hypothesis, reduces to v'_i. According to lemma 4.5, there exists an outcome o such that $e[x \Leftarrow v] \rightarrow^\star o$ and, for $i \in \{1, 2\}$, $\lfloor o \rfloor_i = v'_i$. Because of the latter, o must be a value. Lastly, $h \not\leq \ell$ yields $\ell \notin H$. The result follows by lemma 6.1. □

In words, h and ℓ are security levels such that information flow from h to ℓ is disallowed by the security lattice. Assuming the hole x has a "high"-level type t, the expression e can be given the "low"-level type int^ℓ. Then, no matter which value (of type t) is placed in the hole, e will compute the same value (that is, if it does produce a value at all).

7 Extensions

In this section, we describe a number of language extensions. Some are standard programming facilities which we have left out so far, namely products, sums, and primitive operations. Others are new language constructs which capture common idioms, so as to make them more amenable to analysis. We omit all proofs in this section; they can be found in [PS01].

Basic reductions	$\mathsf{proj}_j\ (v_1,\ v_2) /_i \mu$	$\rightarrow$	$v_j /_i \mu$	(proj)
	$(\mathsf{inj}_j\ v)\ \mathsf{case}\ v_1\ v_2 /_i \mu$	$\rightarrow$	$v_j\, v /_i \mu$	(case)
Lifting	$\mathsf{proj}_j\ \langle v_1 \mid v_2 \rangle / \mu$	$\rightarrow$	$\langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2 \rangle / \mu$	(lift-proj)
	$\langle v_1 \mid v_2 \rangle\ \mathsf{case}\ v'_1\ v'_2 / \mu$	$\rightarrow$	$\langle v_1\ \mathsf{case}\ \lfloor v'_1 \rfloor_1\ \lfloor v'_2 \rfloor_1 \mid v_2\ \mathsf{case}\ \lfloor v'_1 \rfloor_2\ \lfloor v'_2 \rfloor_2 \rangle / \mu$	(lift-case)

Figure 4: Semantics of products and sums

v-Pair
$$\frac{\Gamma, M \vdash v_1 : t_1 \qquad \Gamma, M \vdash v_2 : t_2}{\Gamma, M \vdash (v_1,\ v_2) : t_1 \times t_2}$$

v-Inj
$$\frac{\Gamma, M \vdash v : t}{\Gamma, M \vdash \mathsf{inj}_j\ v : (t +^j *)^*}$$

e-Proj
$$\frac{\Gamma, M \vdash v : t_1 \times t_2}{*, \Gamma, M \vdash \mathsf{proj}_j\ v : t_j\ [*]}$$

e-Match
$$\frac{\Gamma, M \vdash v : (t_1 + t_2)^\ell \qquad \forall j \in \{1,2\}\quad \Gamma, M \vdash v_j : (t_j \xrightarrow{pc \sqcup \ell_j \sqcup \ell\ [r]} t)^{\ell_j} \qquad \ell \sqcup \ell_1 \sqcup \ell_2 \lhd t}{pc, \Gamma, M \vdash v\ \mathsf{case}\ v_1\ v_2 : t\ [r]}$$

Figure 5: Typing products and sums

7.1 Products and sums

Extending our system with products and sums is straightforward. We extend values and expressions with standard constructs:

$$\begin{array}{lcll} v & ::= & \ldots \mid (v,\ v) \mid \mathsf{inj}_j\ v & j \in \{1,2\} \\ e & ::= & \ldots \mid \mathsf{proj}_j\ v \mid v\ \mathsf{case}\ v\ v & j \in \{1,2\} \end{array}$$

The semantics of Core ML^2 is extended with the reduction rules given in figure 4. Rules (proj) and (case) are standard. (lift-proj) and (lift-case) handle the situation where the desired structure is found under a $\langle \cdot \mid \cdot \rangle$ construct; the brackets are then lifted up, as usual, causing some sub-terms to be duplicated. The grammar of types is extended as follows:

$$t \quad ::= \quad \ldots \mid t \times t \mid (t + t)^\ell$$

Our treatment of sums is similar to that of [HR98]. $\ell \lhd (* + *)^{\ell'}$ is, by definition, equivalent to $\ell \leq \ell'$. Our treatment of products, on the other hand, is slightly innovative. Products carry no security annotation because, in the absence of a physical equality operator, all of the information carried by a tuple is in fact carried by its components. To reflect this, we define $\ell \lhd t_1 \times t_2$ as $\ell \lhd t_1 \wedge \ell \lhd t_2$. The typing rules for products and sums are given in figure 5. In v-Inj, $(t_1 +^j t_2)^\ell$ stands for $(t_j + t_i)^\ell$, where $\{i, j\} = \{1, 2\}$.

7.2 Primitive operations

Practical programming languages usually provide many primitive operations, such as integer arithmetic operators. Some languages, such as Caml-Light [LD+97], provide generic comparison, hashing or marshalling functions. In the following, we present a way of assigning types to such primitive operations, without knowledge of their semantics, i.e. by considering them as "black boxes" which potentially use all of the information content of their arguments.

Semantics Assuming given a set $\mathcal{F}$ of primitive operations f, we extend the syntax of expressions as follows:

$$e \quad ::= \quad \ldots \mid f\, v$$

(We only consider unary operations; multiple arguments must be passed in a tuple.) The semantics of every primitive operation f is a partial function $[\![f]\!]$ which maps closed Core ML configurations v / μ to closed Core ML outcomes. Let $|v / \mu|$ denote the configuration obtained from v / μ by removing all bindings in μ

$$\mathsf{unit} \blacktriangleleft \ell \qquad \frac{\ell' \leq \ell}{\mathsf{int}^{\ell'} \blacktriangleleft \ell} \qquad \frac{t_1 \blacktriangleleft \ell \qquad t_2 \blacktriangleleft \ell}{t_1 \times t_2 \blacktriangleleft \ell} \qquad \frac{\ell' \leq \ell \qquad t_1 \blacktriangleleft \ell \qquad t_2 \blacktriangleleft \ell}{(t_1 + t_2)^{\ell'} \blacktriangleleft \ell} \qquad \frac{t \blacktriangleleft \ell \qquad \ell' \leq \ell}{t\ \mathsf{ref}^{\ell'} \blacktriangleleft \ell}$$

Figure 6: Collecting security annotations

which are not accessible through v. The semantics of Core ML^2 is extended as follows:

$$\begin{array}{rcll} f\,v\,/_i\,\mu & \rightarrow & [\![f]\!](\lfloor v \,/\, \mathrm{read}_i\,\mu \rfloor)\,/_i\,\mu & \text{(prim)} \\ f\,v\,/\,\mu & \rightarrow & \langle f\,\lfloor v\rfloor_1 \mid f\,\lfloor v\rfloor_2\rangle\,/\,\mu & \text{(lift-prim)} \\ & & \text{if (prim) isn't applicable} & \end{array}$$

Rule (prim) gives the basic semantics of f. It uses the auxiliary function read_i to access the store; compare to (deref). Its use of $\lfloor\cdot\rfloor$ models the fact that the primitive operation can access the store only through v. The operation cannot affect the store; it may, however, raise an exception, since $[\![f]\!]$ ranges over outcomes, rather than values. Rule (lift-prim) must be applied whenever the configuration $\lfloor v \,/\, \mu\rfloor$ contains at least one $\langle \cdot \mid \cdot\rangle$ constructor; indeed, $[\![f]\!]$ is defined on Core ML configurations only. In that case, we lift all brackets to the toplevel. This is quite crude, but good enough given our intended typing.

Typing In the following, $\bar{r}$ denotes a row ranging over $\{\mathsf{Abs}, \mathsf{Pre}\}$. We write $\bar{r} \cdot pc$ for the row defined as follows: $\bar{r} \cdot pc(\varepsilon)$ equals $\mathsf{Pre}\ pc$ if $\bar{r}(\varepsilon)$ is Pre; it equals Abs otherwise.
The typing of primitive operations, like their semantics, is defined in two steps. First, we assume given, for every $f \in \mathcal{F}$, a set $\mathit{typeof}(f)$ such that, for every $(t', t, \bar{r}) \in \mathit{typeof}(f)$, $M \vdash v : t'$ and $M \vdash \mu$ imply $pc, M \vdash [\![f]\!](\lfloor v \,/\, \mu\rfloor) : t\ [\bar{r} \cdot pc]$. This amounts to assuming subject reduction for (prim); so far, no security concerns need be taken into account.
Then, to enforce security, we define a two-place predicate $\blacktriangleleft$, whose arguments are a type and a security level (figure 6). In short, $t \blacktriangleleft \ell$ requires *all* of the security annotations which appear in t and its sub-terms to be less than (or equal to) ℓ. It also requires t to be a "pure data" type, i.e. to have no function or exception types as its sub-terms. (Functions are ruled out because they are not valid arguments to the polymorphic comparison operators; exceptions must be ruled out because exn is, in practice, an extensible type, i.e. the mapping *typexn* is never fully known.) The predicate $\blacktriangleleft$ enjoys the following property:

Lemma 7.1 *Assume $\vdash_H v \,/\, \mu : t\ [*]$ and $t \blacktriangleleft \ell$. If $\lfloor v \,/\, \mu\rfloor$ isn't a Core ML configuration, then $\ell \in H$.*

We give the following typing rule for applications of primitive operations:

$$\textsc{e-Primitive}\quad \frac{(t', t, \bar{r}) \in \mathit{typeof}(f) \qquad \Gamma, M \vdash v : t' \qquad t' \blacktriangleleft \ell \qquad \ell \lhd t}{pc, \Gamma, M \vdash f\,v : t\ [\bar{r} \cdot (pc \sqcup \ell)]}$$

This is quite crude, since we require the security level of the result type t to dominate all those which appear in the argument type t'. However, as long as nothing is known about $[\![f]\!]$, no better approximation can be given; the outcome may actually depend on any part of f's argument.

Non-interference We now check that the new reduction rules satisfy subject reduction under the extended type system.
∘ *Case* (prim). By Conf and e-Primitive, we have $(t', t, \bar{r}) \in \mathit{typeof}(f)$ and $M \vdash v : t'$ and $M \vdash \mu$. According to our assumption concerning $\mathit{typeof}(\cdot)$, this implies $pc, M \vdash [\![f]\!](\lfloor v \,/\, \mu\rfloor) : t\ [\bar{r} \cdot pc]$. The result follows by e-Sub.
∘ *Case* (lift-prim). Conf and e-Primitive's premises allow applying lemma 7.1, yielding $\ell \in H$. Applying lemma 5.3 and building a new instance of e-Primitive, we get $pc \sqcup \ell, M \vdash f\,\lfloor v\rfloor_i : t\ [\bar{r} \cdot (pc \sqcup \ell)]$ for $i \in \{1, 2\}$. Recalling $\ell \lhd t$, we conclude with e-Bracket.

Applications Let us now illustrate the use of this general mechanism.

The treatment of binary integer arithmetic operations is quite simple, because they are monomorphic: they map pairs of integers to integers. This rule effectively makes the result's security level the union of the arguments' levels:

$$\frac{\Gamma, M \vdash v_1 : \mathsf{int}^\ell \qquad \Gamma, M \vdash v_2 : \mathsf{int}^\ell}{*, \Gamma, M \vdash v_1 \star v_2 : \mathsf{int}^\ell \;\; [*]} \quad \star \in \{+, -, \times, \ldots\}$$

The treatment of the generic (i.e. polymorphic) comparison operators is more interesting.

$$\frac{\Gamma, M \vdash v_1 : t \qquad \Gamma, M \vdash v_2 : t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash v_1 \star v_2 : \mathsf{bool}^\ell \;\; [*]} \quad \star \in \{=, \leq, \geq, \ldots\}$$

(The type bool^ℓ can be defined as $(\mathsf{unit} + \mathsf{unit})^\ell$ or added as a primitive type.) Because these operators traverse data structures recursively, the result of a comparison may reveal information about any sub-term. The premise $t \blacktriangleleft \ell$ reflects this by requiring ℓ to dominate all security annotations which appear in t.
Generic hashing and marshalling operations can be dealt with similarly:

$$\frac{\Gamma, M \vdash v : t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash \mathsf{hash}\, v : \mathsf{int}^\ell \;\; [*]} \qquad\qquad \frac{\Gamma, M \vdash v : t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash \mathsf{marshal}\, v : \mathsf{int}^\ell \;\; [*]}$$

By contrast, in Myers' Java-based framework [Mye99a, Mye99b], hashing is implemented by having every class override the standard hashCode method. However, that method is declared in the class Object with signature int{this} hashCode (). A re-implementation of hashCode by a sub-class of Object must also satisfy this signature. As a result, it may only rely on fields labelled this. The parametric class Vector[L], for instance, must compute hash codes in a way that does not depend upon the vector's length or contents, because their label is L. Of course, this severely limits hashCode's usefulness.

7.3 Common idioms

Because our type system is quite conservative, some common programming idioms deserve receiving special treatment, even though they are already expressible in the language.
For instance, consider the expression form e_1 finally e_2, akin to Lisp's unwind-protect and Java's try-finally constructs. Such an expression could be viewed as syntactic sugar for bind $x = (e_1$ handle $y \succ e_2;$ raise $y)$ in $e_2;\ x$. However, by duplicating e_2, this encoding prevents the type-checker from discovering that e_2 is executed always, i.e. regardless of e_1's behavior. As a result, e_2 is typechecked under an increased security assumption pc. Zdancewic and Myers [ZM01] show how ordered linear continuations provide a general solution to this problem. In our case, it is simpler to make e_1 finally e_2 a primitive construct, whose typing rule is given in figure 7.
Following Myers [Mye99a, Mye99b], we typecheck e_1 and e_2 at a common pc. However, we add the premise $\sqcup r_2 \leq \sqcap r_1$, which reflects that, by observing an exception thrown by e_1, one may deduce that e_2 terminated normally. Its absence in Myers' work is a flaw. Myers' typing rule in fact exhibits a second flaw: its overall effect should be $X_1 \oplus X_2$, rather than $X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2$, because normal termination of the whole statement betrays normal termination of e_1. This fact is taken into account in our typing rule, even though we do not explicitly associate a security level to normal termination; see section 10. Both flaws in Myers' framework were uncovered by our formal approach [Mye01].
Another common idiom which seems to require special treatment is the one which consists in anonymously handling an exception, then raising it again, to be handled further up the call chain. This is typically written e_1 handle $x \succ (e_2;$ raise $x)$. In our type system, the handler $e_2;$ raise x is typechecked at a security level increased by $\sqcup r_1$, where the row r_1 describes the exception x. Then, the second premise of E-Raise requires $\sqcup r_1 \leq \sqcap r_1$, i.e. the security levels associated with all exception names in r_1 must be conflated, leading to a loss of precision. If, on the other hand, we introduce a new expression form e_1 handle $x \succ e_2$ reraise with the same meaning, then we can safely give it a more precise type; see figure 7.

Non-interference The syntax of evaluation contexts and the semantics of Core ML^2 are extended as described in figure 8. (Making new evaluation contexts and new sequencing rules available effectively extends (throw-context), (lift-context) and (context) as well.) Sequential composition $e_1; e_2$ is defined as syntactic sugar for bind $x = e_1$ in e_2, where x doesn't appear free in e_2.
We begin by establishing the following simple lemma:

$$\frac{\begin{array}{c}\text{E-FINALLY}\\ pc,\Gamma,M\vdash e_1 : t\ [r_1] \qquad pc,\Gamma,M\vdash e_2 : *\ [r_2] \qquad \sqcup r_2 \leq \sqcap r_1\end{array}}{pc,\Gamma,M\vdash e_1\ \mathsf{finally}\ e_2 : t\ [r_1\sqcup r_2]}$$

$$\frac{\begin{array}{c}\text{E-RERAISE}\\ pc,\Gamma,M\vdash e_1 : t\ [r_1] \qquad pc\sqcup(\sqcup r_1),\Gamma[x\mapsto r_1\ \mathsf{exn}^*],M\vdash e_2 : *\ [r_2] \qquad \sqcup r_2 \leq \sqcap r_1\end{array}}{pc,\Gamma,M\vdash e_1\ \mathsf{handle}\ x\succ e_2\ \mathsf{reraise} : t\ [r_1\sqcup r_2]}$$

Figure 7: Typing finally and reraise

$$E \quad ::= \quad \ldots \mid [\,]\ \mathsf{finally}\ e \mid [\,]\ \mathsf{handle}\ x\succ e\ \mathsf{reraise}$$

$$\begin{array}{rcll} o\ \mathsf{finally}\ e\ /_i\ \mu & \to & e;\ o\ /_i\ \mu & \text{(finally)}\\ \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x\succ e\ \mathsf{reraise}\ /_i\ \mu & \to & e[x\Leftarrow\varepsilon\, v];\ \mathsf{raise}\ (\varepsilon\, v)\ /_i\ \mu & \text{(reraise)}\end{array}$$

Figure 8: Syntax and semantics of finally and reraise

Lemma 7.2 *$pc,M\vdash o : t\ [r]$ and $pc'\leq\sqcap r$ imply $pc\sqcup pc',M\vdash o : t\ [r]$.*

Proof. If o is a value, the result is a consequence of lemma 5.7 and E-VALUE. If o is of the form $\mathsf{raise}\ (\varepsilon\, v)$, then (discarding, w.l.o.g., any instances of E-SUB) the type derivation ends with an instance of E-RAISE, whose premises remain valid if pc is replaced with $pc\sqcup pc'$, thanks to the hypothesis $pc'\leq\sqcap r$. If o is $\langle o_1\mid o_2\rangle$, the result follows by E-BRACKET and the induction hypothesis. □

We now check that the new reduction rules satisfy subject reduction under the extended type system.

∘ *Case* (finally). e is o $\mathsf{finally}$ e_2 and e' is $(e_2;\ o)$. By E-FINALLY, we have $pc,M\vdash o : t\ [r_1]$ and $pc,M\vdash e_2 : *\ [r_2]$ where $\sqcup r_2\leq\sqcap r_1$. By lemma 7.2, the former yields $pc\sqcup(\sqcup r_2),M\vdash o : t\ [r_1]$. By E-BIND, we obtain $pc,M\vdash e' : t\ [r_1\sqcup r_2]$.

∘ *Case* (reraise). e is $\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x\succ e_2\ \mathsf{reraise}$ and e' is $(e_2[x\Leftarrow\varepsilon\, v];\ \mathsf{raise}\ (\varepsilon\, v))$. By E-RERAISE and lemma 5.2, we have $pc,M\vdash\mathsf{raise}\ (\varepsilon\, v) : t\ [r_1]$ and $pc,(x\mapsto r_1\ \mathsf{exn}^\ell),M\vdash e_2 : *\ [r_2]$ where $\sqcup r_2\leq\sqcap r_1$. By lemma 7.2, the former yields $pc\sqcup(\sqcup r_2),M\vdash\mathsf{raise}\ (\varepsilon\, v) : t\ [r_1]$. By E-SUB, E-RAISE, V-SUB and V-EXN, it also yields $M\vdash\varepsilon\, v : r_1\ \mathsf{exn}^\ell$. By lemma 5.6, the latter then yields $pc,M\vdash e_2[x\Leftarrow\varepsilon\, v] : *\ [r_2]$. Then, by E-BIND, $pc,M\vdash e' : t\ [r_1\sqcup r_2]$ holds.

∘ *Case* (throw-context), sub-case $E = [\ ]\ \mathsf{handle}\ x\succ e_2\ \mathsf{reraise}$. e is $E[o]$ and e' is o. By E-RERAISE, $pc,M\vdash o : t\ [r_1]$ holds. By E-SUB, so does $pc,M\vdash o : t\ [r_1\sqcup r_2]$.

∘ *Case* (lift-context), (context). The descriptions in the proof of lemma 5.8 still apply.

8 A constraint-based type system

We now give a more algorithmic presentation of our type system, called MLIF. It differs from MLIF_0 mainly by introducing type variables, constraints, and using them to form universally quantified, constrained type schemes, in the style of HM(X) [OSW99]. Like HM(X), it has principal types and decidable type inference. Because the construction is not the central topic of this paper, we will describe it only succinctly; the reader is referred to [OSW99, Pot01] for more details.

8.1 Types and constraints

In MLIF, the grammar of types, rows, alternatives and levels is extended with *type variables.* (By lack of symbols, we let α range over type variables of all four kinds; no ambiguity will arise.) Furthermore, Rémy's [Rém92] row syntax is introduced, turning rows into finite lists of bindings from exception names to

Values

$$\textsc{v-Unit}\quad C,\Gamma \vdash () : \mathsf{unit}$$

$$\textsc{v-Int}\quad C,\Gamma \vdash k : \mathsf{int}^*$$

$$\textsc{v-Var}\quad \frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad C \Vdash \exists\bar{\alpha}.D}{C \wedge D, \Gamma \vdash x : \tau}$$

$$\textsc{v-Abs}\quad \frac{C,\pi,\Gamma[x \mapsto \tau'][f \mapsto (\tau' \xrightarrow{\pi\ [\rho]} \tau)^\lambda] \vdash e : \tau\ \ [\rho]}{C,\Gamma \vdash \lambda\langle f\rangle x.e : (\tau' \xrightarrow{\pi\ [\rho]} \tau)^\lambda}$$

$$\textsc{v-Exn}\quad \frac{C,\Gamma \vdash v : \mathit{typexn}(\varepsilon)}{C,\Gamma \vdash \varepsilon\, v : (\varepsilon : \mathsf{Pre}\ *; *)\ \mathsf{exn}^*}$$

$$\textsc{v-Sub}\quad \frac{C,\Gamma \vdash v : \tau' \qquad C \Vdash \tau' \leq \tau}{C,\Gamma \vdash v : \tau}$$

Expressions

$$\textsc{e-Value}\quad \frac{C,\Gamma \vdash v : \tau}{C,*,\Gamma \vdash v : \tau\ \ [*]}$$

$$\textsc{e-App}\quad \frac{C,\Gamma \vdash v_1 : (\tau' \xrightarrow{\pi\sqcup\lambda\ [\rho]} \tau)^\lambda \qquad C,\Gamma \vdash v_2 : \tau' \qquad C \Vdash \lambda \lhd \tau}{C,\pi,\Gamma \vdash v_1\, v_2 : \tau\ \ [\rho]}$$

$$\textsc{e-Ref}\quad \frac{C,\Gamma \vdash v : \tau \qquad C \Vdash \pi \lhd \tau}{C,\pi,\Gamma \vdash \mathsf{ref}\ v : \tau\ \mathsf{ref}^*\ \ [*]}$$

$$\textsc{e-Assign}\quad \frac{C,\Gamma \vdash v_1 : \tau\ \mathsf{ref}^\lambda \qquad C,\Gamma \vdash v_2 : \tau \qquad C \Vdash \pi \sqcup \lambda \lhd \tau}{C,\pi,\Gamma \vdash v_1 := v_2 : \mathsf{unit}\ \ [*]}$$

$$\textsc{e-Deref}\quad \frac{C,\Gamma \vdash v : \tau'\ \mathsf{ref}^\lambda \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \lambda \lhd \tau}{C,\pi,\Gamma \vdash\, !v : \tau\ \ [*]}$$

$$\textsc{e-Raise}\quad \frac{C,\Gamma \vdash v : \rho\ \mathsf{exn}^\lambda \qquad C \Vdash \pi \sqcup \lambda \leq \sqcap\rho}{C,\pi,\Gamma \vdash \mathsf{raise}\ v : *\ \ [\rho]}$$

$$\textsc{e-Let}\quad \frac{C \wedge D,\Gamma \vdash v : \tau' \qquad C,\pi,\Gamma[x \mapsto \forall\bar{\alpha}[D].\tau'] \vdash e : \tau\ \ [\rho] \qquad \bar{\alpha} \cap \mathrm{fv}(C,\Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D,\pi,\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau\ \ [\rho]}$$

$$\textsc{e-Bind}\quad \frac{C,\pi,\Gamma \vdash e_1 : \tau'\ \ [\rho_1] \qquad C,\pi \sqcup (\sqcup\rho_1),\Gamma[x \mapsto \tau'] \vdash e_2 : \tau\ \ [\rho_2]}{C,\pi,\Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \tau\ \ [\rho_1 \sqcup \rho_2]}$$

$$\textsc{e-Handle}\quad \frac{C,\pi,\Gamma \vdash e_1 : \tau\ \ [\varepsilon : \mathsf{Pre}\ \pi';\rho] \qquad C,\pi \sqcup \pi',\Gamma[x \mapsto \mathit{typexn}(\varepsilon)] \vdash e_2 : \tau\ \ [\varepsilon : \eta;\rho] \qquad C \Vdash \pi' \lhd \tau}{C,\pi,\Gamma \vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : \tau\ \ [\varepsilon : \eta;\rho]}$$

$$\textsc{e-HandleAll}\quad \frac{C,\pi,\Gamma \vdash e_1 : \tau\ \ [\rho_1] \qquad C,\pi \sqcup (\sqcup\rho_1),\Gamma[x \mapsto \rho_1\ \mathsf{exn}^*] \vdash e_2 : \tau\ \ [\rho_2] \qquad C \Vdash (\sqcup\rho_1) \lhd \tau}{C,\pi,\Gamma \vdash e_1\ \mathsf{handle}\ x \succ e_2 : \tau\ \ [\rho_2]}$$

$$\textsc{e-Sub}\quad \frac{C,\pi,\Gamma \vdash e : \tau'\ \ [\rho'] \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \rho' \leq \rho}{C,\pi,\Gamma \vdash e : \tau\ \ [\rho]}$$

Figure 9: The type system MLIF

alternatives, terminated with a row variable or with a constant row $\partial\eta$.

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid \mathsf{unit} \mid \mathsf{int}^\lambda \mid (\tau \xrightarrow{\pi\ [\rho]} \tau)^\lambda \mid \tau\ \mathsf{ref}^\lambda \mid \rho\ \mathsf{exn}^\lambda \\
\rho & ::= & \alpha \mid (\varepsilon : \eta; \rho) \mid \partial\eta \\
\eta & ::= & \alpha \mid \mathsf{Abs} \mid \mathsf{Pre}\ \pi \\
\lambda, \pi & ::= & \alpha \mid \ell
\end{array}$$

The variable-free types (resp. rows, alternatives, levels) of MLIF are isomorphic to the types (resp. rows, alternatives, levels) of MLIF_0; we identify them and refer to them as *ground*. Then, *constraints* are defined as follows:

$$\begin{array}{rcl}
C & ::= & \mathbf{true} \mid C \wedge C \mid \exists\alpha.C \\
& \mid & \tau \leq \tau \mid \rho \leq \rho \mid \eta \leq \eta \mid \lambda \leq \lambda \\
& \mid & \lambda \lhd \tau \mid \sqcup\rho \leq \lambda \mid \lambda \leq \sqcap\rho \mid \tau \blacktriangleleft \lambda
\end{array}$$

The constraint forms on the first line are standard [OSW99]. Those on the second line are subtyping constraints; those on the third line are custom constraint forms, which correspond to the notions developed in sections 5 and 7.2. We omit the sorting rules necessary to ensure that terms and constraints involving rows are well-formed; see [Rém93].

Let a *ground assignment* ϕ map every type variable α to a ground type, row, alternative, or level, according to its kind. The meaning of terms and constraints under an assignment ϕ is defined in the obvious way. We write $C \Vdash C'$ (read: C entails C') if and only if every assignment ϕ which satisfies C satisfies C' as well.

Let a *type scheme* be a triple of a set of quantifiers $\bar{\alpha}$, a constraint C and a type τ; we write $\sigma = \forall\bar{\alpha}[C].\tau$. The type variables in $\bar{\alpha}$ are bound in σ; type schemes are considered equal modulo α-conversion. By abuse of notation, a type term τ may be viewed as a type scheme $\forall\varnothing[\mathbf{true}].\tau$. An *environement* Γ is a partial mapping from program variables to type schemes.

8.2 Typing rules

The typing rules for MLIF are given in figure 9. They look very similar to those of MLIF_0; let us briefly discuss the differences. We restrict our attention to *source* expressions, i.e. Core ML expressions which do not contain memory locations; this is enough for our purposes. Thus, typing judgements no longer contain a memory environment M. Every judgement begins with a constraint C which represents an assumption about its free type variables; for the judgement to be valid, C must be satisfiable. (We omit C when it is **true**.) Constrained type schemes are introduced by E-LET, which performs generalization, and eliminated by V-VAR, which performs instantiation. For the sake of conciseness, some rules use the binary operator $\sqcup$ on levels and on rows, as well as the unary operator $\sqcup$ on rows, as if they were part of our term syntax; we let the reader check that these notations can be de-sugared into extra type variables and constraints.

8.3 Non-interference

We prove the following statement by induction on type derivations, along the lines of [Pot01].

Lemma 8.1 (Soundness) *Assume $C, \pi, \Gamma \vdash e : \tau\ [\rho]$. Let ϕ be an arbitrary assignment which satisfies C. Then, $\phi(\pi), \phi(\Gamma), \varnothing \vdash e : \phi(\tau)\ [\phi(\rho)]$ holds in MLIF_0.*

(We do not define $\phi(\Gamma)$ here; see [Pot01].) In particular, every ground typing judgement in MLIF is also a valid judgement in MLIF_0. This allows us to lift our non-interference result to MLIF. That is, the statement of theorem 6.1 remains valid if $(x \mapsto t) \vdash e : \mathsf{int}^\ell$ and $\vdash v_i : t$ are read as MLIF typing judgements.

The typing rules given in figure 9 do not necessarily allow deriving *ground* typing judgements about every expression. However, it is easy to enrich the system with rules similar to HM(X)'s $\exists$-INTRO and WEAKEN [SMZ99, Pot01], which allow specializing a non-ground judgement to any of its ground instances.

8.4 Type inference

It is easy to check that there exists a type inference algorithm which computes principal types for MLIF. Sulzmann *et al.* [SMZ99] show how to derive a set of type inference rules from a set of typing rules similar to ours. The major point that remains to be settled is whether constraint solving is decidable. We claim that it is. Proofs and algorithms will be given in a later paper; let us only briefly indicate why we believe so.

As explained in section 5.2, our subtyping relation is atomic; constraint solving in the presence of atomic subtyping is decidable and well understood [Reh97]. The introduction of rows is essentially orthogonal to other constraint solving issues [Fäh99, Pot00a]. There remains to check that our custom constraint forms are tractable. It is easy to see that $\sqcup\rho \leq \lambda$ is in fact equivalent to $\rho \leq \partial\lambda$. Furthermore, $\lambda \leq \sqcap\rho$ may be encoded as the conditional constraint $\mathsf{Pre} \leq \rho \mathrel{?} \partial(\mathsf{Pre}\ \lambda) \leq \rho$. (See e.g. [Pot00b] for more information about conditional constraints.) Lastly, the assertions $\lambda \lhd \tau$ and $\tau \blacktriangleleft \lambda$ can be formulated as $\lambda \leq F^{\lhd}(\tau)$ and $F^{\blacktriangleleft}(\tau) \leq \lambda$, respectively, where $F^{\lhd}$ and $F^{\blacktriangleleft}$ are appropriate monotonic mappings from types to levels. We believe that such constraints can be solved in a "lazy" manner, reminiscent of the way conditional constraints are dealt with: a constraint of the form $\lambda \lhd \alpha$ or $\alpha \blacktriangleleft \lambda$ remains suspended as long as nothing is known about α, and is decomposed into a number of sub-constraints only when α is unified with a non-variable term τ.

9 Examples

We intend to integrate MLIF into a realistic programming language, such as Caml-Light [LD+97]. In this section, we give a taste of that by describing the principal type schemes inferred for some library functions by our prototype implementation. We use Caml-Light syntax, which can be easily de-sugared into Core ML. We omit type annotations on top of $\rightarrow$ when they are unconstrained, anonymous type variables. Because none of the type schemes below has free type variables, we omit the universally quantified variables after $\forall$. We have not explained how to include datatype declarations in the language. Since we already have product and sum types, this should be straightforward. Let us assume the type constructor list is declared as follows:

```
type ('a, 'b) list = <'b>
  | []
  | (::) of 'a * ('a, 'b) list
```

In $\alpha\,\mathsf{list}^\beta$, the parameter α is the type of the list's elements, as usual, while β is a security level. The annotation `<'b>` on the right-hand side is meant to indicate that β is the security annotation carried by the sum type. Our first example function computes the length of a list:

```
let rec length = function
  | []      -> 0
  | _ :: l -> 1 + length l
```

A valid type scheme for `length` is $\forall[\alpha \leq \beta]. * \ \mathsf{list}^\alpha \rightarrow \mathsf{int}^\beta$. As expected, the result's security annotation β does not depend on the type of the list's elements. The constraint $\alpha \leq \beta$ describes the information flow induced by the function: the length of a list contains some information about its structure. This type scheme is in fact equivalent to $\forall[]. * \ \mathsf{list}^\alpha \rightarrow \mathsf{int}^\alpha$, a simplification which our implementation performs automatically.

```
let rec iter f = function
  | []      -> ()
  | x :: l -> f x; iter f l
```

`iter` applies `f` successively to every element of a list. Its inferred type scheme is

$$\forall[\sqcup\gamma \leq \beta].(\alpha \xrightarrow{\beta\ [\gamma]} *)^\beta \rightarrow \alpha\,\mathsf{list}^\beta \xrightarrow{\beta\ [\gamma]} \mathsf{unit}$$

Here, γ represents `f`'s effect. Because `iter` does not throw any exceptions of its own, γ is also `iter`'s effect. β is `f`'s *pc* parameter. It must dominate `iter`'s own *pc* parameter (because `f` is invoked by `iter`), the list's security level (because gaining control tells `f` that the list is non-empty) and $\sqcup\gamma$ (because gaining control tells `f` that its previous invocation terminated normally).

```
let incr r =
  r := !r + 1
```

`incr` has $\forall[].\mathsf{int}^\alpha\ \mathsf{ref}^\alpha \xrightarrow{\alpha\ [*]} \mathsf{unit}$ as principal type scheme. Indeed, by E-ASSIGN, the security level of the reference's contents must dominate both `incr`'s *pc* parameter and the reference's own security level. We now re-implement `length` in imperative style:

```
let length' l =
  let count = ref 0 in
  iter (fun () -> incr count) l;
  !count
```

We obtain $\forall[].\ast\ \mathsf{list}^{\alpha} \xrightarrow{\alpha\ [\ast]} \mathsf{int}^{\alpha}$. This appears more restrictive than `length`'s type scheme: the result's security level must now be greater than or equal to the function's *pc* parameter. However, the difference is only superficial; it can be checked that both types in fact have the same expressive power. Formalizing this claim, and understanding its consequences, are left for future work. We continue with a few library functions which deal with association lists.

```
let rec mem_assoc x = function
  | []          -> false
  | (y, _) :: l ->
      if x = y then true else mem_assoc x l
```

Because `mem_assoc`'s result reveals information about both the structure and the keys stored in the list, we obtain:

$$\forall[\alpha \blacktriangleleft \beta].\alpha \rightarrow (\alpha \times \ast)\,\mathsf{list}^{\beta} \rightarrow \mathsf{bool}^{\beta}$$

The constraint $\alpha \blacktriangleleft \beta$, which arises due to the use of polymorphic equality, specifies that β must be an upper bound for all security annotations which occur in the type of the keys.

```
let rec assoc x = function
  | []          -> raise Not_found
  | (y, d) :: l -> if x = y then d else assoc x l
```

`assoc` returns the piece of data associated with a given key. If no such key exists, `Not_found` is raised, as reflected in `assoc`'s effect:

$$\forall[\alpha \blacktriangleleft \beta, \beta \lhd \gamma, \beta \leq \delta].\alpha \rightarrow (\alpha \times \gamma)\,\mathsf{list}^{\beta} \xrightarrow{\delta\ [\mathsf{Not\_found}:\ \delta;\ \ast]} \gamma$$

Here, as in `mem_assoc`, β represents the information associated with the list's structure and keys. Because this information is reflected both in `assoc`'s normal and exceptional results, the type system requires $\beta \lhd \gamma$ and $\beta \leq \delta$.
To conclude, we re-implement `mem_assoc` in terms of `assoc`, using an exception handler:

```
let mem_assoc' x l =
  try
    let _ = assoc x l in
    true
  with Not_found ->
    false
```

As in the case of `length` vs. `length'`, the new type scheme requires the result's security level to be greater than or equal to the function's *pc* parameter:

$$\forall[\alpha \blacktriangleleft \beta].\alpha \rightarrow (\alpha \times \ast)\,\mathsf{list}^{\beta} \xrightarrow{\beta\ [\ast]} \mathsf{bool}^{\beta}$$

This betrays the fact that the function's implementation uses effects, but does not otherwise restrict its applicability.

10 Discussion

The reader may notice that normal and exceptional results are not dealt with in a symmetric way by our type system. Indeed, in a typing judgement $pc, \Gamma, M \vdash e : t\ [r]$, the row r associates a security level with every exception name, so as to record how much information is gained by observing that particular exception. However, no information level is explicitly associated with normal termination. Instead, the typing rule for sequential composition, namely E-BIND, uses $\sqcup r$ as an approximation of it.

Myers' [Mye99a, Mye99b] sets of path labels X, on the other hand, record the security level associated with normal termination under a special label $\underline{\mathsf{n}}$, which is then used in the sequential composition rule. It is, however, typically an upper bound for the value reached by *pc* inside every sub-expression of the expression at hand, so this design alone would make the type system very restrictive. To prevent that, Myers adds a non-syntax-directed rule, the *single-path* rule, stating that $X[\underline{\mathsf{n}}]$ can be reset to $\underline{\emptyset}$ if the expression at hand can be shown to always terminate normally.

Our system doesn't need the single-path rule: indeed, when all entries in r_1 are Abs, then $\sqcup r_1$ is the least element of $\mathcal{L}$, and E-BIND typechecks e_1 and e_2 at a common *pc*, as desired. Myers' system is more precise than ours in a few cases, which involve expressions that *never* terminate normally; experience will tell how common they are. The single-path rule requires counting the number of non-Abs entries in a row; in the presence of row variables, this requires new (and quite heavy) constraint forms, which is why we avoid it. This difficulty does not arise in Myers' framework because it relies on Java's explicit, monomorphic throws clauses.

There exists a simple monadic encoding of exceptions into sums. Thus, it is possible, in principle, to derive a type system for exceptions out of a type system that can handle sums. This approach sounds interesting, because it is systematic and promises to yield a symmetric treatment of normal vs. exceptional results. However, some experiments show that, in order to obtain acceptable precision in the end, the treatment of sums that is chosen as a starting point must be very accurate (much more so than the one given in this paper). We leave it as a topic of future research.

Our main direction for future work is to create a full implementation of the system on top of Caml-Light and to assess its usability through a number of case studies.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999. URL: `http://www.star-lab.com/mabadi/Papers/flowpopl.ps`.

[ALL96] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 83–91, Philadelphia, Pennsylvania, May 1996. URL: `http://www.star-lab.com/mabadi/Papers/make-preprint.ps`.

[BL75] D. E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corp., Bedford, Massachusetts, July 1975. URL: `http://www.mitre.org/resources/centers/infosec/infosec.html`.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[Fäh99] Manuel Fähndrich. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999. URL: `http://research.microsoft.com/~maf/diss.ps`.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, June 1993. URL: `http://www.cs.rice.edu/CS/PLT/Publications/pldi93-fsdf.ps.gz`.

[GM82] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998. URL: `http://cm.bell-labs.com/cm/cs/who/nch/slam.ps`.

[LD+97] Xavier Leroy, Damien Doligez, et al. The Caml Light system, release 0.74. URL: `http://caml.inria.fr/`, 1997.

[Mye99a] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press. URL: http://www.cs.cornell.edu/andru/papers/popl99/myers-popl99.ps.gz.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control.* PhD thesis, Massachusetts Institute of Technology, January 1999. Technical Report MIT/LCS/TR-783. URL: http://www.cs.cornell.edu/andru/release/tr783.ps.gz.

[Mye01] Andrew C. Myers. Personal communication, June 2001.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz.

[PL00] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000. URL: http://pauillac.inria.fr/~xleroy/publi/exceptions-toplas.ps.gz.

[Pot00a] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping, February 2000. URL: http://pauillac.inria.fr/~fpottier/wallace/.

[Pot00b] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz.

[Pot01] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, March 2001. URL: ftp://ftp.inria.fr/INRIA/publication/RR/RR-4150.ps.gz.

[PS01] François Pottier and Vincent Simonet. Information flow inference for ML. Full version. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-popl02-long.ps.gz, July 2001.

[Reh97] Jakob Rehof. Minimal typings in atomic subtyping. In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages*, pages 278–291, Paris, France, January 1997. URL: http://research.microsoft.com/~rehof/popl97.ps.

[Rém92] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz.

[Rém93] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. URL: http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214:607–621, April 1997. URL: http://www.cs.nps.navy.mil/people/faculty/volpano/papers/tapsoft97.ps.Z.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996. URL: http://www.cs.nps.navy.mil/people/faculty/volpano/papers/jcs96.ps.Z.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. URL: `http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz`.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995. URL: `http://www.cs.rice.edu/CS/PLT/Publications/lasc95-w.ps.gz`.

[ZM01] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming (ESOP'01)*, Lecture Notes in Computer Science. Springer Verlag, April 2001. URL: `http://www.cs.cornell.edu/zdance/lincont.ps`.

Quatrième partie

Perspectives

A type based information flow analysis for ML with sums

Draft

Abstract

In this paper, we present a type-based information flow analysis for a call-by-value λ-calculus equipped with references, sums and let-polymorphism, which we refer to as ML_Σ. The main novelty resides in the fine treatment of sums. In the last section, we show how a type system for ML with exceptions may be obtained by encoding into ML_Σ.

1 ML with sums

1.1 ML_Σ

Let x, m, c range over disjoint denumerable sets of *program variables*, *memory locations* and *constructors names*, respectively. We denote by $\mathcal{C}$ the set of all constructors names and by Λ any subset of $\mathcal{C}$. Then, *values*, *expressions*, *handlers* and *evaluation contexts* are defined as follows:

$$
\begin{array}{rcl}
v & ::= & x \mid m \mid () \mid \lambda x.e \mid c\,v \\
e & ::= & v \mid v\,v \mid \mathsf{ref}\ v \mid v := v \mid !\,v \mid \bar{c}\,v \mid v\ \mathsf{case}\ [h \mid \ldots \mid h] \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid E[e] \\
h & ::= & \Lambda : x \mapsto e \\
E & ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e
\end{array}
$$

Our values include variables, memory locations, a unit constant, λ-abstractions and constructed values. (We do not provide integer values since they may be represented by constructors.) An expression is a value, a *basic expression*, a let construct or another expression enclosed within an evaluation context.
Basic expressions include function applications, primitives dealing with memory cells, projections and case expressions. For each constructor name c, ML_Σ provides a projection $\bar{c}$ such that $\bar{c}\,(c\,v)$ evaluates to v. A handler is a triple of a subset of $\mathcal{C}$, a program variable and an expression. The case construction allows to match a value's head constructor against a list of handlers: $c\,v\ \mathsf{case}\ [\Lambda_1 : x_1 \mapsto e_1 \mid \ldots \mid \Lambda_n : x_n \mapsto e_n]$ reduces to $e_j[x_j \Leftarrow c\,v]$ for j such that $c \in \Lambda_j$. (In the following, we will assume that case expressions are *exhaustive*, i.e. that $\Lambda_1, \ldots, \Lambda_n$ form a partition of the set of constructors.) The conjunction of projections and case clauses gives us all the expressiveness of usual pattern matching on constructors.
The only evaluation context allows to combine expressions and specify the evaluation order. The expression $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ evaluates e_1, binds its value to x and then evaluates e_2.

1.2 $\mathrm{ML}_\Sigma{}^2$

In this section, we present a technical extension of ML_Σ that allows reasoning simulaneously about the reduction of two expressions of ML_Σ that share some sub-terms. This extension is built using exactly the same method as in [1], so that we will describe it succinctly, focusing our attention on the addition of sums; the reader is referred to the previous paper for more details. This extension is as follows:

$$\begin{array}{lcl} v & ::= & \ldots \mid \langle v \mid v \rangle \mid \mathsf{void} \\ e & ::= & \ldots \mid \langle e \mid e \rangle \end{array}$$

We define two *projections* $\lfloor \cdot \rfloor_1$ and $\lfloor \cdot \rfloor_2$ that map $\mathrm{ML}_\Sigma{}^2$ expressions to ML_Σ ones. They are homomorphisms except at $\langle \cdot \mid \cdot \rangle$ nodes: $\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i$. We extend them to handlers by $\lfloor \Lambda : x \mapsto e \rfloor_i = \Lambda : x \mapsto \lfloor e \rfloor_i$.
A *store* μ is a partial map from memory locations to values. We write $\mu[m \mapsto v]$ and $\mu \oplus [m \mapsto v]$ for the store which maps m to v and otherwise agrees with μ; the latter is defined only if $m \notin \mathrm{dom}(\mu)$. A *configuration* $e \,/_i\, \mu$ is a triple of an expression e, a store μ and an index $i \in \{\bullet, 1, 2\}$. We restrict our attention to well-formed, closed configurations (see [2], section 4.1). The capture-free substitution of v for x in e, written $e[x \Leftarrow v]$, is defined in the usual way, except at $\langle \cdot \mid \cdot \rangle$ nodes, where we must use an appropriate projection of v in each branch: $\langle e_1 \mid e_2 \rangle[x \Leftarrow v]$ is $\langle e_1[x \Leftarrow \lfloor v \rfloor_1 \mid x \Leftarrow \lfloor v \rfloor_2 \rangle$.
The small step operational semantics of $\mathrm{ML}_\Sigma{}^2$ (and ML_Σ) is given in figure 1. To clarify the presentation, we divide the set of reduction rules into four groups. *Basic reductions* and *sequencing* rules are those of ML_Σ (with some technical twists concerning access to the store, see [1]). *Lifting* rules are specific to $\mathrm{ML}_\Sigma{}^2$ because they deal with information sharing.
(proj) and (lift-proj) reduce projections. The former is the standard rule and the latter is to prevent $\langle \cdot \mid \cdot \rangle$ constructs from blocking reduction. Note that it would be correct to replace it by

$$\bar{c}\,\langle c\,v'_1 \mid c\,v'_2 \rangle \,/\, \mu \to \langle v'_1 \mid v'_2 \rangle \,/\, \mu \qquad \text{(lift-proj')}$$

(case) defines the semantics of case clauses. We write $v \Subset \Lambda$ if and only Λ *matches* v, that means if $v = c\,v'$ for some $c \in \Lambda$ or $v = \langle c_1\,v_1 \mid c_2\,v_2 \rangle$ for some $c_1 \in \Lambda$ and $c_2 \in \Lambda$. Then the expression $c\,v'$ case $[\Lambda_1 : x_1 \mapsto e_1 \mid \ldots \mid \Lambda_n : x_n \mapsto e_n]$ reduces to $e_j[x_j \Leftarrow c\,v']$ for j such that $c \in \Lambda_j$. Our semantics of case is quite sharp towards sharing since values like $\langle v_1 \mid v_2 \rangle$ may be matched without any lifting if the two head constructors belong to the same Λ_j. If they don't, (lift-case) must be applied before reducing the two resulting case sub-expressions separately.

1.3 Relating $\mathrm{ML}_\Sigma{}^2$ to ML_Σ

We now show that $\mathrm{ML}_\Sigma{}^2$ expressions may correctly simulate the simultaneous reductions of two ML_Σ expressions. As in [2], we state two auxiliaries lemmas (1.1 and 1.3); then lemmas 1.2 and 1.4 establish the *soundness* and the *completeness* of the simulation, respectively.

▶ **Lemma 1.1** *Let $i \in \{1, 2\}$. If $e \,/_i\, \mu \to e' \,/_i\, \mu'$, then $e \,/\, \lfloor \mu \rfloor_i \to e' \,/\, \lfloor \mu' \rfloor_i$.*

Proof By inspection of (ref), (assign) and (deref). □

▶ **Lemma 1.2 (Soundness)** *Let $i \in \{1, 2\}$. If $e \,/\, \mu \to e' \,/\, \mu'$, then $\lfloor e \,/\, \mu \rfloor_i \to^= \lfloor e' \,/\, \mu' \rfloor_i$.*

Proof By inspection of the reduction rules and appeal to lemma 1.1. □

A configuration $e \,/_i\, \mu$ is *stuck* if it is irreducible and e isn't a value. It is *successful* if e is a value. The following lemma will be used in the proof of the completeness property.

▶ **Lemma 1.3 (Stuck Configurations)** *If $e \,/\, \mu$ is stuck, then $\lfloor e \,/\, \mu \rfloor_i$ is stuck for some $i \in \{1, 2\}$.*

Proof By induction on the structure of e.

∘ *Cases* $e = v_1\,v_2$, $e = (v_1 := v_2)$, $e = {!}\,v$, $e = \mathsf{ref}\ v$, $e = (\mathsf{let}\ x = v\ \mathsf{in}\ e')$ and $e = \langle e_1 \mid e_2 \rangle$. See the corresponding cases in the proof of lemma 4.4 in [2].

∘ *Case* $e = \bar{c}\,v$. Because neither (proj) nor (lift-proj) is applicable, v cannot be of the form $\langle v_1 \mid v_2 \rangle$ or $c\,v'$. As a result, for any $i \in \{1, 2\}$, $\lfloor v \rfloor_i$ cannot be of the form $c\,v'$. It follows that $\lfloor e \,/\, \mu \rfloor_i$ is stuck.

$$
\begin{array}{llcll}
\textbf{Basic reductions} & (\lambda x.e)\,v \mathbin{/_i} \mu & \to & e[x \Leftarrow v] \mathbin{/_i} \mu & (\beta) \\
 & \mathsf{let}\ x = v\ \mathsf{in}\ e \mathbin{/_i} \mu & \to & e[x \Leftarrow v] \mathbin{/_i} \mu & (\mathrm{let}) \\
 & \mathsf{ref}\ v \mathbin{/_i} \mu & \to & m \mathbin{/_i} \mu \oplus [m \mapsto \mathrm{new}_i\, v] & (\mathrm{ref}) \\
 & m := v \mathbin{/_i} \mu & \to & () \mathbin{/_i} \mu[m \mapsto \mathrm{update}_i\, \mu(m)\, v] & (\mathrm{assign}) \\
 & !\,m \mathbin{/_i} \mu & \to & \mathrm{read}_i\, \mu(m) \mathbin{/_i} \mu & (\mathrm{deref}) \\
 & \bar{c}\,(c\,v) \mathbin{/_i} \mu & \to & v \mathbin{/_i} \mu & (\mathrm{proj}) \\
 & v\ \mathsf{case}\ [\Lambda_1 : x_1 \mapsto e_n \mid \ldots \mid \Lambda_n : x_n \mapsto e_n] \mathbin{/_i} \mu & \to & e_j[x_j \Leftarrow v] \mathbin{/_i} \mu \qquad (\textit{if } v \in \Lambda_j) & (\mathrm{case}) \\
 & & & & \\
\textbf{Sequencing} & \mathsf{bind}\ x = v\ \mathsf{in}\ e \mathbin{/_i} \mu & \to & e[x \Leftarrow v] \mathbin{/_i} \mu & (\mathrm{bind}) \\
 & & & & \\
\textbf{Lifting} & \langle v_1 \mid v_2 \rangle\, v \mathbin{/} \mu & \to & \langle v_1\, \lfloor v \rfloor_1 \mid v_2\, \lfloor v \rfloor_2 \rangle \mathbin{/} \mu & (\text{lift-app}) \\
 & \langle v_1 \mid v_2 \rangle := v \mathbin{/} \mu & \to & \langle v_1 := \lfloor v \rfloor_1 \mid v_2 := \lfloor v \rfloor_2 \rangle \mathbin{/} \mu & (\text{lift-assign}) \\
 & !\,\langle v_1 \mid v_2 \rangle \mathbin{/} \mu & \to & \langle\, !\,v_1 \mid !\,v_2 \rangle \mathbin{/} \mu & (\text{lift-deref}) \\
 & \bar{c}\,\langle v_1 \mid v_2 \rangle \mathbin{/} \mu & \to & \langle \bar{c}\,v_1 \mid \bar{c}\,v_2 \rangle \mathbin{/} \mu & (\text{lift-proj}) \\
 & \langle v_1 \mid v_2 \rangle\ \mathsf{case}\ [\vec{h}] \mathbin{/} \mu & \to & \langle v_1\ \mathsf{case}\ [\lfloor \vec{h} \rfloor_1] \mid v_2\ \mathsf{case}\ [\lfloor \vec{h} \rfloor_2] \rangle \mathbin{/} \mu & (\text{lift-case}) \\
 & & & (\textit{if } (\mathrm{case}) \textit{ does not apply}) &
\end{array}
$$

$$
\textbf{Reduction under a context} \qquad \frac{e \mathbin{/_i} \mu \to e' \mathbin{/_i} \mu'}{E[e] \mathbin{/_i} \mu \to E[e'] \mathbin{/_i} \mu'} \qquad (\mathrm{context})
$$

$$
\frac{e_i \mathbin{/_i} \mu \to e_i' \mathbin{/_i} \mu' \qquad e_j = e_j' \qquad \{i, j\} = \{1, 2\}}{\langle e_1 \mid e_2 \rangle \mathbin{/} \mu \to \langle e_1' \mid e_2' \rangle \mathbin{/} \mu'} \qquad (\mathrm{bracket})
$$

$$
\begin{array}{lrcl rcl rcl}
 & \mathrm{new}_\bullet\, v & = & v & \mathrm{update}_\bullet\, v\, v' & = & v' & \mathrm{read}_\bullet\, v & = & v \\
\textbf{Auxiliary functions} & \mathrm{new}_1\, v & = & \langle v \mid \mathsf{void} \rangle & \mathrm{update}_1\, v\, v' & = & \langle v' \mid \lfloor v \rfloor_2 \rangle & \mathrm{read}_1\, v & = & \lfloor v \rfloor_1 \\
 & \mathrm{new}_2\, v & = & \langle \mathsf{void} \mid v \rangle & \mathrm{update}_2\, v\, v' & = & \langle \lfloor v \rfloor_1 \mid v' \rangle & \mathrm{read}_2\, v & = & \lfloor v \rfloor_2
\end{array}
$$

Figure 1: Semantics of $\mathrm{ML}_{\Sigma}{}^{2}$

$$\odot\ \mathsf{ref}^{\oplus} \qquad (\ominus \xrightarrow{\ominus} \oplus)^{\oplus} \qquad \Sigma^{\oplus}\ \oplus \qquad \{c \mapsto \oplus\} \qquad \{(c,c) \mapsto \oplus\} \qquad \mathsf{Pre}\ \oplus \qquad \mathsf{Abs} \leq \mathsf{Pre} *$$

Figure 2: Subtyping

- *Case* v case $\{\Lambda_j : x_j \mapsto e_j\}_j$. Because neither (case) nor (lift-case) is applicable, v cannot be of the form $\langle v_1 \mid v_2 \rangle$ or $c\, v'$. As a result, for any $i \in \{1,2\}$, $\lfloor v \rfloor_i$ cannot be of the form $c\, v'$. It follows that $\lfloor e \,/\, \mu \rfloor_i$ is stuck.
- *Case* $e = E[e_1]$. By induction hypothesis, because e_1 cannot be a value. □

▶ **Lemma 1.4 (Completeness)** *Assume $\lfloor e/\mu \rfloor_i \to^\star v_i/\mu'_i$ for all $i \in \{1,2\}$. Then, there exists a configuration $v \,/\, \mu'$ such that $e \,/\, \mu \to^\star v \,/\, \mu'$ and, for all $i \in \{1,2\}$, $\lfloor v \,/\, \mu' \rfloor_i = v_i \,/\, \mu'_i$.*

Proof See [2], lemma 4.5. □

2 Typing

In the following, given two distinct constructors c_1 and c_2, we denote by $c_1 \cdot c_2$ the (unordered) pair $\{c_1, c_2\}$. Λ^2 denotes the set of pairs of elements of Λ for any $\Lambda \subseteq \mathcal{C}$.

2.1 Types

Let $(\mathcal{L}, \leq)$ be a lattice whose elements, denoted by ℓ and pc, represent *security levels*. *Types*, *alternatives*, *content-rows* and *level-rows* are defined as follows:

$$\begin{array}{rcl} t & ::= & \mathsf{unit} \mid t\ \mathsf{ref}^{\ell} \mid (t \xrightarrow{pc} t)^{\ell} \mid \Sigma^q\, r \\ a & ::= & \mathsf{Abs} \mid \mathsf{Pre}\, t \\ r & ::= & \{c \mapsto a\} \\ q & ::= & \{c \cdot c \mapsto \ell\} \end{array}$$

A *content-row* (resp. a *level-row*) is an infinite, quasi-constant family of alternatives (resp. levels) indexed by constructors (resp. pairs of constructors). We write $(c : a; r)$ for the content-row whose element at index c is a and whose other elements are given by the sub-row r, which is indexed by $\mathcal{C} \backslash \{c\}$. We denote by ∂a the row that maps all its entries to a. We adopt similar notations for level-rows. Furthermore, we write $r \Cap \Lambda$ (read: r *cuts* Λ) if and only if there exists $c \in \Lambda$ such that $r(c) = \mathsf{Pre} *$.
Types for the unit constant (unit), references ($t\ \mathsf{ref}^{\ell}$) and functions ($(t \xrightarrow{pc} t)^{\ell}$) are the same as in [1] (excepted the absence of the effect annotation $[r]$ on arrows). The novelty resides in sum types, $\Sigma^q\, r$, that consist in two distinct rows. First, the content-row r indicates for each constructor c if the described expression may ($\mathsf{Pre}\, t$) or may not (Abs) produce a value whose head constructor is c (the constructor Pre carries in addition the expected type of the constructor's argument). Second, for every pair of constructors $c_1 \cdot c_2$, $q(c_1 \cdot c_2)$ gives an approximation of the level of information you are likely to leak by revealing that the described expression has produced a value whose head constructor is c_1 rather than c_2, or symmetrically c_2 rather than c_1. (Note that if $r(c) = \mathsf{Abs}$, all the levels $q(c \cdot *)$ do not carry in practice any relevant information and may be constrained to $\bot$. In fact, it is possible to restrict q to $\{c \mid r(c) = \mathsf{Pre} *\}^2$.)
A *polytype* s is a non-empty, upward-closed set of types. A *program environment* Γ is a partial mapping from program variables to polytypes while a *memory environment* M is a partial mapping form memory locations to types.

Subtyping We equip types, rows and alternatives with a subtyping relation $\leq$ extending $(\mathcal{L}, \leq)$. It is defined by axioms in figure 2. $\oplus$, $\ominus$ and $\odot$ stand respectively for *covariant*, *contravariant* and *invariant* arguments. For instance, the axiom $\Sigma^{\oplus} \oplus$ is an abbreviation of $\Sigma^{q_1}\, r_1 \leq \Sigma^{q_2}\, r_2 \Leftrightarrow r_1 \leq r_2 \wedge q_1 \leq q_2$. In the following, we write $r_1 \leq_\Lambda r_2$ if and only if $\forall c \in \Lambda,\ r_1(c) \leq r_2(c)$. At the same, $q_1 \leq_\Lambda q_2$ means $\forall c \cdot c' \in \Lambda^2,\ q_1(c \cdot c') \leq q_2(c \cdot c')$.

$$\ell \lhd (\mathsf{unit}, \mathsf{unit}) \qquad \frac{\ell \leq \ell_1 \sqcap \ell_2}{\ell \lhd (*\ \mathsf{ref}^{\ell_1}, *\ \mathsf{ref}^{\ell_2})} \qquad \frac{\ell \leq \ell_1 \sqcap \ell_2}{\ell \lhd ((* \xrightarrow{*} *)^{\ell_1}, (* \xrightarrow{*} *)^{\ell_2})}$$

$$\frac{\begin{array}{c}\forall c \in \mathcal{C},\ \ r_1(c) = \mathsf{Pre}\, t_1 \ \wedge\ r_2(c) = \mathsf{Pre}\, t_2 \ \Rightarrow\ \ell \lhd (t_1, t_2) \\ \forall c_1 \cdot c_2 \in \mathcal{C}^2,\ \ r_1(c_1) = \mathsf{Pre}\ * \ \wedge r_2(c_2) = \mathsf{Pre}\, * \ \Rightarrow\ \ell \leq q_1(c_1 \cdot c_2) \sqcap q_2(c_1 \cdot c_2)\end{array}}{\ell \lhd (\Sigma^{q_1}\, r_1, \Sigma^{q_2}\, r_2)}$$

Figure 3: Guards

Guards In order to obtain a fine treatment of sums, the $\lhd$ relation must involve a pair of types rather than a simple one. Then we define $\ell \lhd (t_1, t_2)$ by rules of figure 3, and $\ell \lhd t$ as a shortcut for $\ell \lhd (t, t)$. This definition is equivalent to the one of [1] for unit, references and functions types.
In practice, we will be interested by the union of the two guarded types: let us consider an expression e that performs a test (involving datas whose level is approximated by ℓ) and may produce two different results whose types are respectively t_1 and t_2. With simple guards, we would have to compute the union t of t_1 and t_2 (that is t such that $t_1 \leq t$ and $t_2 \leq t$) and then guarding t by ℓ. Conversely, here we will be able to guard the pair (t_1, t_2) and then compute the union. This treatment is finer in presence of two Σ-types because we keep trace of the origin of each field of the resulting row. For instance if $t_1 = \Sigma^* \,(c_1 : \mathsf{Pre}\,*; c_2 : \mathsf{Pre}\,*; \partial\mathsf{Abs})$, $t_2 = \Sigma^* \,(c_3 : \mathsf{Pre}\,*; \partial\mathsf{Abs})$ (with c_1, c_2 and c_3 all different) and $t = \Sigma^q\, r$ (with $t_1 \leq t$ and $t_2 \leq t$), $\ell \lhd (t_1, t_2)$ does not constrain $q(c_1 \cdot c_2)$ while $\ell \lhd t$ does.
Because of the special treatment for sums, lemma 5.1 of [1] must be restricted as follows.

▶ **Lemma 2.1** *If $\ell' \leq \ell$ and $\ell \lhd (t_1, t_2)$ then $\ell' \lhd (t_1, t_2)$. If $\ell \lhd (t_1, t_2)$, $t_1 \leq t'$, $t_2 \leq t'$ and t' is not a Σ-type, then $\ell \lhd t'$.*

2.2 Typing rules

In our system, we distinguish two forms of typing judgements: one deals with values only ($\Gamma, M \vdash v : t$), the other with arbitrary expressions ($pc, \Gamma, M \vdash e : t$). The typing rules are given in figure 4. Most of them are identical to [1] ones (excepted again the absence of the effet $[r]$ at the end of judgements on expressions). So, here we just comment rules dealing with sums: V-INJ, E-PROJ and E-MATCH.
V-INJ is quite standard. It associates to the constructed value $c v$ a content-row which maps c to $\mathsf{Pre}\, t$ (where t is v's type) and leaves other entries unconstrained, allowing them to be Abs. The level-row is alike unconstrained and may be $\partial\bot$ since knowing the head constructor of the value cannot reveal any information at this point. E-PROJ requires $\bar{c}$'s argument to have the type $\Sigma^q\,(c : \mathsf{Pre}\,*; \partial\mathsf{Abs})$, what would be the case for a value matched by $\{c\}$ in a case expression.
E-MATCH is quite tricky. The first premise specifies the type of the matched value, which is as expected a sum type. The premises of the second line must be read as quantified over all indices j. The handler e_j is typechecked at a security level increased by ℓ_j which is the union of all the security levels you are likely to leak by revealing that the head constructor of the matched value belongs to Λ_j or not. The type assigned in the environment to the program variable x is nothing but the "restriction" to Λ_j of the matched value's one. Last, the third line must be read as quantified over all pairs of indices $j \cdot j'$. It computes the result type t from the handlers types t_j. First, each t_j is taken into account only if its corresponding handler may be executed, that is $r \Cap \Lambda_j$. Secondly, these types are combined by pair and guarded by the corresponding ℓ_j.

2.3 Subject reduction

▶ **Lemma 2.2 (Subsumption)** *$pc' \leq pc$ and $pc, \Gamma, M \vdash e : t$ imply $pc', \Gamma, M \vdash e : t$.*

Proof See [2], lemma 5.2. □

▶ **Lemma 2.3 (Projection)** *Let $i \in \{1, 2\}$. If $\Gamma, M \vdash v : t$ then $\Gamma, M \vdash \lfloor v \rfloor_i : t$. If $pc, \Gamma, M \vdash e : t$ then $pc, \Gamma, M \vdash \lfloor e \rfloor_i : t$.*

Values

v-Unit
$$\Gamma, M \vdash () : \mathsf{unit}$$

v-Void
$$\Gamma, M \vdash \mathsf{void} : *$$

v-Loc
$$\Gamma, M \vdash m : M(m)\ \mathsf{ref}^*$$

v-Var
$$\frac{t \in \Gamma(x)}{\Gamma, M \vdash x : t}$$

v-Abs
$$\frac{pc, \Gamma[x \mapsto t'], M \vdash e : t}{\Gamma, M \vdash \lambda x.e : (t' \xrightarrow{pc} t)^\ell}$$

v-Inj
$$\frac{\Gamma, M \vdash v : t}{\Gamma, M \vdash c\, v : \Sigma^* \left(c : \mathsf{Pre}\, t; *\right)}$$

v-Bracket
$$\frac{\Gamma, M \vdash v_1 : t_1 \qquad \Gamma, M \vdash v_2 : t_2 \qquad pc' \in H \qquad pc' \lhd (t_1, t_2) \qquad t_1 \sqcup t_2 \leq t}{\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t}$$

v-Sub
$$\frac{\Gamma, M \vdash v : t' \qquad t' \leq t}{\Gamma, M \vdash v : t}$$

Expressions

e-Value
$$\frac{\Gamma, M \vdash v : t}{*, \Gamma, M \vdash v : t}$$

e-App
$$\frac{\Gamma, M \vdash v_1 : (t' \xrightarrow{pc \sqcup \ell} t)^\ell \qquad \Gamma, M \vdash v_2 : t' \qquad \ell \lhd t}{pc, \Gamma, M \vdash v_1\, v_2 : t}$$

e-Ref
$$\frac{\Gamma, M \vdash v : t \qquad pc \lhd t}{pc, \Gamma, M \vdash \mathsf{ref}\ v : t\ \mathsf{ref}^*}$$

e-Assign
$$\frac{\Gamma, M \vdash v_1 : t\ \mathsf{ref}^\ell \qquad \Gamma, M \vdash v_2 : t \qquad pc \sqcup \ell \lhd t}{pc, \Gamma, M \vdash v_1 := v_2 : \mathsf{unit}}$$

e-Deref
$$\frac{\Gamma, M \vdash v : t'\ \mathsf{ref}^\ell \qquad t' \leq t \qquad \ell \lhd t}{pc, \Gamma, M \vdash\ !v : t}$$

e-Proj
$$\frac{\Gamma, M \vdash v : \Sigma^q \left(c : \mathsf{Pre}\, t; \partial\mathsf{Abs}\right)}{pc, \Gamma, M \vdash \bar{c}\, v : t}$$

e-Match
$$\frac{\begin{array}{c}\Gamma, M \vdash v : \Sigma^q\, r \\ pc \sqcup \ell_j, \Gamma[x_j \mapsto \Sigma^{q_j}\, r_j], M \vdash e_j : t_j \qquad r \leq_{\Lambda_j} r_j \qquad q \leq_{\Lambda_j} q_j \qquad \sqcup\{q(c \cdot c') \mid c \in \Lambda_j, c' \notin \Lambda_j\} \leq \ell_j \\ \ell_j \sqcup \ell_{j'} \lhd (t_j, t_{j'}) \qquad (r \Cap \Lambda_j)\ ?\ (t_j \leq t)\end{array}}{pc, \Gamma, M \vdash v\ \mathsf{case}\ \{\Lambda_j : x_j \mapsto e_j\}_j : t}$$

e-Let
$$\frac{\Gamma, M \vdash v : s \qquad pc, \Gamma[x \mapsto s], M \vdash e : t}{pc, \Gamma, M \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : t}$$

e-Bind
$$\frac{pc, \Gamma, M \vdash e_1 : t' \qquad pc, \Gamma[x \mapsto t'], M \vdash e_2 : t}{pc, \Gamma, M \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : t}$$

e-Bracket
$$\frac{\begin{array}{c}pc \sqcup pc', \Gamma, M \vdash e_1 : t_1 \qquad pc \sqcup pc', \Gamma, M \vdash e_2 : t_2 \\ pc' \in H \qquad pc' \lhd (t_1, t_2) \qquad t_1 \sqcup t_2 \leq t\end{array}}{pc, \Gamma, M \vdash \langle e_1 \mid e_2 \rangle : t}$$

e-Sub
$$\frac{pc, \Gamma, M \vdash e : t' \qquad t' \leq t}{pc, \Gamma, M \vdash e : t}$$

Configurations

Store
$$\frac{\begin{array}{c}\mathrm{dom}(M) = \mathrm{dom}(\mu) \\ \forall m \in \mathrm{dom}(\mu) \quad M \vdash \mu(m) : M(m)\end{array}}{M \vdash \mu}$$

Conf
$$\frac{pc, \Gamma, M \vdash e : t \qquad M \vdash \mu}{\Gamma \vdash e \,/\, \mu : t}$$

Figure 4: The type system MLIF_Σ

Proof See [2], lemma 5.3. □

▶ **Lemma 2.4 (Guard)** *Assume t is not a Σ-type. If $\Gamma, M \vdash \langle v_1 \mid v_2\rangle : t$ then there exists $pc' \in H$ that $pc' \lhd t$.*

Proof The derivation of $\Gamma, M \vdash \langle v_1 \mid v_2\rangle : t$ terminates by an instance of v-Bracket followed by a series of v-Sub. It follows that there exists $t' \leq t$ such that $\Gamma, M \vdash \langle v_1 \mid v_2\rangle : t'$ holds, with a derivation terminating by an instance of v-Bracket, among whose premises we find $pc' \lhd (t'_1, t'_2)$, $t'_1 \sqcup t'_2 \leq t'$ and $pc' \in H$. Last, because t is not a Σ-type, lemma 2.1 gives $pc' \lhd t$. □

▶ **Lemma 2.5 (Store access)** *Let i be in $\{\bullet, 1, 2\}$. Assume $\Gamma, M \vdash v : t$ and $\Gamma, M \vdash v' : t$. Then $\Gamma, M \vdash \mathrm{read}_i\, v : t$ holds. Moreover, if $i \in \{1, 2\}$, assume there exists some $pc' \in H$ such that $pc' \lhd t$. Then $\Gamma, M \vdash \mathrm{new}_i\, v : t$ and $\Gamma, M \vdash \mathrm{update}_i\, v\, v' : t$ hold.*

Proof By definition of the functions new, update and read (figure 1), by lemma 2.3, by v-Void and v-Bracket. □

▶ **Lemma 2.6** *Assume $\Gamma, M \vdash v : \Sigma^q\, r$. If $v \Subset \Lambda$, $r \leq_\Lambda r'$ and $q \leq_\Lambda q'$ then $\Gamma, M \vdash v : \Sigma^{q'}\, r'$.*

Proof We assume, w.l.o.g., that the derivation of $\Gamma, M \vdash v : \Sigma^q\, r$ does not end with an instance of v-Sub. Because $v \Subset \Lambda$, one of the two following cases must arise.

∘ *Case* $v = c\, v'$ with $c \in \Lambda$. By v-Inj, we have $\Gamma, M \vdash v' : t$ with $r(c) = \mathsf{Pre}\, t$. Re-building an instance of v-Inj, we obtain $\Gamma, M \vdash v : \Sigma^{\partial\perp}\, (c : \mathsf{Pre}\, t; \partial\mathsf{Abs})$. Because $c \in \Lambda$, this yields, by v-Sub, $\Gamma, M \vdash v : \Sigma^{q'}\, r'$.

∘ *Case* $v = \langle c_1\, v_1 \mid c_2\, v_2\rangle$ with $c_1 \in \Lambda$ and $c_2 \in \Lambda$. By v-Bracket, v-Inj and v-Sub, we have $\Gamma, M \vdash v_1 : t_1$ and $\Gamma, M \vdash v_2 : t_2$ with $r(c_1) = \mathsf{Pre}\, t_1$, $r(c_2) = \mathsf{Pre}\, t_2$ and $pc' \leq q(c_i \cdot c_j)$ (for all $i, j \in \{1, 2\}$ and some $pc' \in H$). Let $r'' = (c_1 : \mathsf{Pre}\, t_1; c_2 : \mathsf{Pre}\, t_2; \partial\mathsf{Abs})$ and $q'' = (c_1, c_2 \cdot : pc'; \partial\perp)$. By v-Inj, we have $\Gamma, M \vdash c\, v_1 : \Sigma^{q''}\, r''$ and $\Gamma, M \vdash c\, v_2 : \Sigma^{q''}\, r''$. Conclude with an instance of v-Sub. □

▶ **Lemma 2.7** *Assume $v \Subset \Lambda$. If $\Gamma, M \vdash v : \Sigma^q\, r$ then $r \Cap \Lambda$.*

▶ **Lemma 2.8 (Substitution)** *Assume $M \vdash v : s$. Then $\Gamma[x \mapsto s], M \vdash v' : t$ implies $\Gamma, M \vdash v'[x \Leftarrow v] : t$. Also $pc, \Gamma[x \mapsto s], M \vdash e : t$ implies $pc, \Gamma, M \vdash e[x \Leftarrow v] : t$.*

Proof See [2], lemma 5.6. □

▶ **Lemma 2.9 (Value)** *$pc, \Gamma, M \vdash v : t$ implies $\Gamma, M \vdash v : t$*

Proof See [2], lemma 5.7. □

▶ **Lemma 2.10 (Subject reduction)** *Let $e\ /_i\ \mu \to e'\ /_i\ \mu'$. Assume $pc, M \vdash e : t$ and $M \vdash \mu$. If $i \in \{1, 2\}$, assume $pc \in H$. Then, there exists a memory environment M', which extends M, such that $pc, M' \vdash e' : t$ and $M' \vdash \mu'$.*

Proof By induction on the derivation of $e\ /_i\ \mu \to e'\ /_i\ \mu'$. We assume, w.l.o.g., that the derivation of $pc, M \vdash e : t$ does not end with an instance of e-Sub. As a result, it must end with an instance of the single syntax-directed rule that matches e's structure.

∘ *Cases* (β), (let), (ref), (assign) and (deref). See [2].

∘ *Case* (proj). e is $\bar{c}\,(c\, v)$ and e' is v. By e-Proj, we have $M \vdash c\, v : \Sigma^*\, (c : \mathsf{Pre}\, t; *)$. By v-Sub and v-Inj, this yields $M \vdash c\, v : t$. Conclude with e-Value.

∘ *Case* (case). e is v case $[\Lambda_1 : x_1 \mapsto e_1 \mid \ldots \mid \Lambda_n : x_n \mapsto e_n]$ and e' is $e_j[x_j \Leftarrow v]$, with $v \Subset \Lambda_j$. By E-Match and lemma 2.7, we have $M \vdash v : \Sigma^q\, r$, $pc \sqcup \ell_j, (x_j \mapsto \Sigma^{q_j}\, r_j), M \vdash e_j : t_j$ with $t_j \leq t$, $r \leq_{\Lambda_j} r_j$ and $q \leq_{\Lambda_j} q_j$. By lemma 2.6, this yields $M \vdash v : \Sigma^{q_j}\, r_j$. Applying lemmas 2.2, 2.8 and E-Sub, we obtain $pc, M \vdash e' : t$.

∘ *Case* (bind). By E-Bind and lemma 2.9.

∘ *Case* (lift-app), (lift-assign) and (lift-deref). See [2].

∘ *Case* (lift-proj). e is $\bar{c}\,\langle v_1 \mid v_2\rangle$ and e' is $\langle \bar{c}\, v_1 \mid \bar{c}\, v_2\rangle$. By E-Proj, we have $M \vdash \langle v_1 \mid v_2\rangle : \Sigma^q\,(c : \mathsf{Pre}\, t; \partial\mathsf{Abs})$. This derivation terminates by an instance of V-Bracket possibly followed by a series of V-Sub. It follows that there exists $t_1 \leq t$, $t_2 \leq t$ and $pc' \in H$ such that $M \vdash v_1 : \Sigma^q\,(\mathsf{Pre}\, t_1; \partial\mathsf{Abs})$, $M \vdash v_2 : \Sigma^q\,(\mathsf{Pre}\, t_2; \partial\mathsf{Abs})$ and $pc' \lhd (t_1, t_2)$. Conclude by E-Proj and E-Bracket.

∘ *Case* (lift-case). e is $\langle v_1 \mid v_2\rangle$ case $[\Lambda_j : x_j \mapsto e_j]_j$ and e' is $\langle v_1$ case $[\Lambda_j : x_j \mapsto \lfloor e_j\rfloor_1]_j \mid v_2$ case $[\Lambda_j : x_j \mapsto \lfloor e_j\rfloor_2]_j\rangle$. By E-Match, we have $\Gamma, M \vdash \langle v_1 \mid v_2\rangle : \Sigma^q\, r$. Then v_1 and v_2 are respectively of the form $c_1\, v_1'$ and $c_2\, v_2'$. Because (case) does not apply, we may assume w.l.o.g. that $c_1 \in \Lambda_1$ and $c_2 \in \Lambda_2$. The derivation of $\Gamma, M \vdash \langle v_1 \mid v_2\rangle : \Sigma^q\, r$ must end by an instance of V-Bracket possibly followed by a series of V-Sub. It follows that there exists $r_1 \leq r$, $q_1 \leq q$, $r_2 \leq r$, $q_2 \leq q$ and $pc' \in H$ such that $M \vdash v_1 : \Sigma^{q_1}\, r_1$, $M \vdash v_2 : \Sigma^{q_2}\, r_2$ and $pc' \lhd (\Sigma^{q_1}\, r_1, \Sigma^{q_2}\, r_2)$. Lemma 2.7 implies $r_1(c_1) = \mathsf{Pre}\, *$ and $r_2(c_2) = \mathsf{Pre}\, *$. Then $pc' \leq q(c_1 \cdot c_2)$.
By E-Match and lemmas 2.1, 2.2, $pc \sqcup q(c_1 \cdot c_2), (x \mapsto \Sigma^{q_j}\, r_j), M \vdash e_j : t_j$, $r \leq_{\Lambda_j} r_j$, $q \leq_{\Lambda_j} q_j$, $t_j \leq t$ (for all $j \in \{1, 2\}$) and $q(1 \cdot 2) \lhd (t_1, t_2)$ hold. By lemmas 2.2, 2.3, 2.6 and E-Match, we have $pc \sqcup pc', M \vdash v_1$ case $\{\Lambda_j : x_j \mapsto \lfloor e_j\rfloor_1\}_j : t_1$ and $pc \sqcup pc', M \vdash v_2$ case $\{\Lambda_j : x_j \mapsto \lfloor e_j\rfloor_2\}_j : t_2$. Conclude by E-Bracket.

∘ *Case* (bracket). See [2].

∘ *Case* (context). e is $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ and e' is $\mathsf{bind}\ x = e_1'\ \mathsf{in}\ e_2$ where $e_1 \,/_i\, \mu \rightarrow e_1' \,/_i\, \mu'$. By E-Bind, we have $pc, M \vdash e_1 : t'$ and $pc, (x \mapsto t'), M \vdash e_2 : t$. Applying the induction hypothesis, the former yields $pc, M' \vdash e_1' : t'$ and $M' \vdash \mu'$ for some M' extending M. Because M' extends M, $pc, (x \mapsto t'), M' \vdash e_2 : t$ remains valid. Conclude by building a new instance of E-Bind.

□

▶ **Theorem 2.11 (Subject reduction)** *If* $\vdash e \,/\, \mu : t$ *and* $e \,/\, \mu \rightarrow e' \,/\, \mu'$ *then* $\vdash e' \,/\, \mu' : t$.

Proof By Conf and lemma 2.10. □

2.4 Non-interference

▶ **Lemma 2.12** *Let* H *be an upward-closed subset of* $\mathcal{L}$. *Let* $\ell \notin H$. *If* $\vdash_H e : \Sigma^{\partial\ell}\, r$ *and* $e \rightarrow^\star v$ *then* $\lfloor v\rfloor_1$ *and* $\lfloor v\rfloor_2$ *share the same head constructor.*

Proof By theorem 2.10 and Conf, there exists a memory environment M such that $M \vdash_H v : \Sigma^{\partial\ell}\, r$ holds. A value of type $\Sigma^{\partial\ell}\, r$ must be of the form $c\, v'$ or $\langle c_1\, v_1 \mid c_2\, v_2\rangle$. If the latter, then, by V-Bracket or E-Bracket, there exists $t_1 \leq \Sigma^{\partial\ell}\, r$, $t_2 \leq \Sigma^{\partial\ell}\, r$ and $pc' \in H$ such that $pc' \lhd (t_1, t_2)$. We conclude that c_1 and c_2 must be equal. □

▶ **Theorem 2.13 (Non-interference)** *Choose* $\ell, h \in \mathcal{L}$ *such that* $h \not\leq \ell$. *Let* $h \lhd t$. *Assume* $(x \mapsto t) \vdash e : \Sigma^{\partial\ell}\, r$, *where* e *is a* ML_Σ *expression. If* $\vdash v_i : t$ *and* $e[x \Leftarrow v_i] \rightarrow^\star v_i'$, *for* $i \in \{1, 2\}$, *then* v_1' *and* v_2' *share the same head constructor.*

Proof See [2]. □

Basic reductions	$(\lambda x.e)\, v \,/\, \mu \;\to\; e[x \leftarrow v] \,/\, \mu$		(β)
	$\mathsf{ref}\ v \,/\, \mu \;\to\; m \,/\, \mu \oplus [m \mapsto v]$		(ref)
	$m := v \,/\, \mu \;\to\; () \,/\, \mu[m \mapsto v]$		(assign)
	$!\, m \,/\, \mu \;\to\; \mu(m) \,/\, \mu$		(deref)
	$\mathsf{let}\ x = v\ \mathsf{in}\ e \,/\, \mu \;\to\; e[x \leftarrow v] \,/\, \mu$		(let)
Sequencing	$\mathsf{bind}\ x = v\ \mathsf{in}\ e \,/\, \mu \;\to\; e[x \leftarrow v] \,/\, \mu$		(bind)
	$\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ \varepsilon\, x \succ e \,/\, \mu \;\to\; e[x \leftarrow v] \,/\, \mu$		(handle)
	$\mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e \,/\, \mu \;\to\; e[x \leftarrow \varepsilon\, v] \,/\, \mu$		(handle-all)
	$E[o] \,/\, \mu \;\to\; o \,/\, \mu$	if E does not handle o	(throw-context)
Reduction under a context	$\dfrac{e \,/\, \mu \to e' \,/\, \mu'}{E[e] \,/\, \mu \to E[e'] \,/\, \mu'}$		(context)

Figure 5: Operational semantics of $\mathrm{ML}_{\mathcal{E}}$

3 ML with exceptions

3.1 $\mathrm{ML}_{\mathcal{E}}$

In this section, we define a variant of ML with exceptions, $\mathrm{ML}_{\mathcal{E}}$. Let $\mathcal{E}$ be a denumerable set whose elements are called *exceptions names* and denoted by ε. Then values, outcomes, expressions and evaluation contexts of $\mathrm{ML}_{\mathcal{E}}$ are defined as follows:

$$
\begin{array}{rcl}
v & ::= & x \mid m \mid () \mid \lambda x.e \mid \varepsilon\, v \\
o & ::= & v \mid \mathsf{raise}\ (\varepsilon\, v) \\
e & ::= & v \mid v\, v \mid \mathsf{ref}\ v \mid v := v \mid !\, v \mid \mathsf{raise}\ v \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid E[e] \\
E & ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \mid [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e \mid [\,]\ \mathsf{handle}\ x \succ e
\end{array}
$$

In fact, $\mathrm{ML}_{\mathcal{E}}$ is identical to the language studied in [1] where it is referred as "*Core ML*". Its small-step semantics is reminded in figure 5.

3.2 Encoding $\mathrm{ML}_{\mathcal{E}}$ into ML_{Σ}

We assume that the constructors of ML_{Σ} are exactly the exception names of $\mathrm{ML}_{\mathcal{E}}$, with an additional one (in order to represent values), denoted by η (i.e. $\mathcal{C} = \mathcal{E} \cup \{\eta\}$). Figure 6 defines a translation $[\![\cdot]\!]$ mapping from $\mathrm{ML}_{\mathcal{E}}$ outcomes into ML_{Σ} values and $\mathrm{ML}_{\mathcal{E}}$ expressions into ML_{Σ} ones. (At the same, we define an auxiliary application $(\!|\cdot|\!)$ mapping from $\mathrm{ML}_{\mathcal{E}}$ values into ML_{Σ} values.) The translation is stable towards substitution in the sense that $[\![e[x \leftarrow v]]\!] = [\![e]\!][x \Leftarrow (\!|v|\!)]$. The following lemma establishes the correction of the encoding towards the semantics.

▶ **Lemma 3.1 (Simulation)** *If $e \,/\, \mu \to e' \,/\, \mu'$ then $[\![e]\!] \,/\, [\![\mu]\!] \to^\star [\![e']\!] \,/\, [\![\mu']\!]$*

Proof By induction on the derivation of $e \,/\, \mu \to e' \,/\, \mu'$.

∘ *Case* (β). e is $(\lambda x.e_0)\, v$ and e' is $e_0[x \leftarrow v]$. Then $[\![e]\!] = (\lambda x.[\![e_0]\!])\, (\!|v|\!)$. We have $[\![e]\!]/[\![\mu]\!] \to [\![e_0]\!][x \Leftarrow (\!|v|\!)]/[\![\mu]\!]$, i.e. $[\![e]\!] \,/\, [\![\mu]\!] \to [\![e']\!] \,/\, [\![\mu]\!]$.

∘ *Case* (ref). e is $\mathsf{ref}\ v$, e' is m and $\mu' = \mu \oplus [m \mapsto v]$. Then $[\![e]\!] = \mathsf{bind}\ y = \mathsf{ref}\ (\!|v|\!)\ \mathsf{in}\ \eta\, y$. We have $[\![e]\!] \,/\, [\![\mu]\!] \to^\star \eta\, m \,/\, [\![\mu]\!] \oplus [m \mapsto (\!|v|\!)]$, i.e. $[\![e]\!] \,/\, [\![\mu]\!] \to^\star [\![e']\!] \,/\, [\![\mu']\!]$.

∘ *Cases* (assign) and (deref) are similar to (ref), (let) to (β).

∘ *Case* (bind). e is $\mathsf{bind}\ x = v\ \mathsf{in}\ e_0$ and e' is $e_0[x \leftarrow v]$. We have

$$
\begin{array}{rcl}
[\![e]\!] & = & \mathsf{bind}\ y = \eta\, (\!|v|\!)\ \mathsf{in} \\
& & y\ \mathsf{case}\ \{\eta : y_1 \mapsto \mathsf{bind}\ x = \bar{\eta}\, y_1\ \mathsf{in}\ e_0;\ \ \mathcal{E} : y_1 \mapsto y_1\}
\end{array}
$$

Then, by applying successively (bind), (case) and (bind), $[\![e]\!] \,/\, [\![\mu]\!] \to^\star [\![e_0]\!][x \leftarrow (\!|v|\!)] \,/\, [\![\mu]\!]$, i.e. $[\![e]\!] \,/\, [\![\mu]\!] \to^\star [\![e']\!] \,/\, [\![\mu']\!]$.

Values

$$\begin{array}{rcl}
(\!|x|\!) &=& x \\
(\!|m|\!) &=& m \\
(\!|()|\!) &=& () \\
(\!|\lambda x.e|\!) &=& \lambda x.[\![e]\!] \\
(\!|\varepsilon\, v|\!) &=& \varepsilon\, (\!|v|\!)
\end{array}$$

Expressions

$$\begin{array}{rcl}
[\![v]\!] &=& \eta\, (\!|v|\!) \\
[\![v_1\, v_2]\!] &=& (\!|v_1|\!)\, (\!|v_2|\!) \\
[\![\mathsf{ref}\ v]\!] &=& \mathsf{bind}\ y = \mathsf{ref}\ (\!|v|\!)\ \mathsf{in}\ \eta\, y \\
[\![v_1 := v_2]\!] &=& \mathsf{bind}\ y = (\!|v_1|\!) := (\!|v_2|\!)\ \mathsf{in}\ \eta\, y \\
[\![!\, v]\!] &=& \mathsf{bind}\ y = !\, (\!|v|\!)\ \mathsf{in}\ \eta\, y \\
[\![\mathsf{raise}\ v]\!] &=& (\!|v|\!) \\
[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] &=& \mathsf{let}\ x = (\!|v|\!)\ \mathsf{in}\ [\![e]\!] \\
[\![E[e]]\!] &=& [\![E]\!][[\![e]\!]]
\end{array}$$

Evaluation contexts

$$\begin{array}{rcl}
[\![\mathsf{bind}\ x = [\,]\ \mathsf{in}\ e]\!] &=& \mathsf{bind}\ y = [\,]\ \mathsf{in} \\
&& y\ \mathsf{case}\ [\eta : y' \mapsto \mathsf{bind}\ x = \bar{\eta}\, y'\ \mathsf{in}\ [\![e]\!] \mid \mathcal{E} : y' \mapsto y'] \\
[\![[\,]\ \mathsf{handle}\ \varepsilon\, x \succ e]\!] &=& \mathsf{bind}\ y = [\,]\ \mathsf{in} \\
&& y\ \mathsf{case}\ [\varepsilon : y' \mapsto \mathsf{bind}\ x = \bar{\varepsilon}\, y'\ \mathsf{in}\ [\![e]\!] \mid \mathcal{C}\backslash\{\varepsilon\} : y' \mapsto y'] \\
[\![[\,]\ \mathsf{handle}\ x \succ e]\!] &=& \mathsf{bind}\ y = [\,]\ \mathsf{in} \\
&& y\ \mathsf{case}\ [\eta : y' \mapsto y' \mid \mathcal{E} : x \mapsto [\![e]\!]]
\end{array}$$

Stores

$$[\![\mu]\!] = \{m \mapsto (\!|\mu(m)|\!)\}_{m \in \mathrm{dom}(\mu)}$$

Figure 6: Encoding $\mathrm{ML}_{\mathcal{E}}$ into ML_{Σ}

- *Cases* (handle), (handle-all) and (throw-context) are similar to (bind).
- *Case* (context). e is $E[e_0]$ and e' is $E[e'_0]$ with $e_0 / \mu \to e'_0 / \mu'$. By induction hypothesis, we have $[\![e_0]\!] / [\![\mu]\!] \to^\star [\![e'_0]\!] / [\![\mu']\!]$. Then, by (context), $[\![E]\!][[\![e_0]\!]] / [\![\mu]\!] \to^\star [\![E]\!][[\![e'_0]\!]] / [\![\mu']\!]$, i.e. $[\![e]\!] / [\![\mu]\!] \to^\star [\![e']\!] / [\![\mu']\!]$. □

3.3 Typing $\mathsf{ML}_{\mathcal{E}}$

In this section, we will define the type system $\mathrm{MLIF}_{\mathcal{E}}$ and proof its non-interference property by translating $\mathrm{MLIF}_{\mathcal{E}}$ judgements into MLIF_{Σ} ones. *Types*, *alternatives*, *content-rows* and *level-rows* are defined as follows:

$$\begin{array}{rcl}
\mathrm{t} &::=& \mathsf{unit} \mid \mathrm{t}\ \mathsf{ref}^{\ell} \mid (\mathrm{t} \xrightarrow{pc} \mathrm{r}^{[q]})^{\ell} \mid \dot{\mathrm{r}}\ \mathsf{exn}^{\dot{q}} \\
\mathrm{a} &::=& \mathsf{Abs} \mid \mathsf{Pre}\,\mathrm{t} \\
\mathrm{r} &::=& \{c \mapsto \mathrm{a}\} \qquad \dot{\mathrm{r}} ::= \{\varepsilon \mapsto \mathrm{a}\} \\
q &::=& \{c \cdot c \mapsto \ell\} \qquad \dot{q} ::= \{\varepsilon \cdot \varepsilon \mapsto \ell\}
\end{array}$$

(The dotted versions of rows are nothing but restriction to the set of exception names of the standard ones.) An effect $\mathrm{r}^{[q]}$ a pair of a content-row r and of a level-row q. It describes the possible normal or exceptional results of a computation with the associated security levels. The encoding of types and effects is defined by figure 7. Polytypes and environments are defined as usual, and the translation application $[\![\cdot]\!]$ is extended to them by:

$$[\![s]\!] = \uparrow\{[\![t]\!] \mid t \in s\} \qquad\qquad [\![\Gamma]\!](x) = [\![\Gamma(x)]\!]$$

Guard over types and effects is defined by rules of figure 8. Because this definition is nothing but a translation of the previous one, the following lemma holds.

▶ **Lemma 3.2** $\ell \lhd \mathrm{t}$ *(resp.* $\ell \lhd \mathrm{r}^{[q]}$*) implies* $\ell \lhd [\![\mathrm{t}]\!]$ *(resp.* $\ell \lhd [\![\mathrm{r}^{[q]}]\!]$*).*

Proof By inspection of the rules defining $\lhd$ in $\mathrm{ML}_{\mathcal{E}}$. □

$$
\begin{array}{rcl}
[\![\mathsf{unit}]\!] & = & \mathsf{unit} \\
[\![\mathrm{t}\ \mathsf{ref}^{\ell}]\!] & = & [\![\mathrm{t}]\!]\ \mathsf{ref}^{\ell} \\
[\![(\mathrm{t} \xrightarrow{pc} \mathrm{r}^{[q]})^{\ell}]\!] & = & ([\![\mathrm{t}]\!] \xrightarrow{pc} [\![\mathrm{r}^{[q]}]\!])^{\ell} \\
[\![\dot{\mathrm{r}}\ \mathsf{exn}^{\dot{q}}]\!] & = & \Sigma^{(\eta\cdot *:\partial\bot;\dot{q})}\,(\eta : \mathsf{Abs}; [\![\dot{\mathrm{r}}]\!]) \\
[\![\mathrm{r}]\!] & = & \{c \mapsto [\![r(c)]\!]\} \\
[\![\dot{\mathrm{r}}]\!] & = & \{\varepsilon \mapsto [\![r(c)]\!]\} \\
[\![\mathsf{Abs}]\!] & = & \mathsf{Abs} \\
[\![\mathsf{Pre}\,\mathrm{t}]\!] & = & \mathsf{Pre}\,[\![\mathrm{t}]\!] \\
[\![\mathrm{r}^{[q]}]\!] & = & \Sigma^{q}\,[\![\mathrm{r}]\!]
\end{array}
$$

Figure 7: Encoding $\mathrm{ML}_{\mathcal{E}}$ types and effects

$$
\ell \lhd \mathsf{unit} \qquad \frac{\ell \le \ell'}{\ell \lhd * \ \mathsf{ref}^{\ell'}} \qquad \frac{\ell \le \ell'}{\ell \lhd (* \xrightarrow{*} *)^{\ell'}} \qquad \ell \lhd \mathsf{Abs} \qquad \frac{\ell \lhd t}{\ell \lhd \mathsf{Pre}\,t}
$$

$$
\frac{\forall \varepsilon \in \mathcal{E},\ \ell \lhd \dot{\mathrm{r}}(\varepsilon) \qquad \forall \varepsilon_1 \cdot \varepsilon_2,\ r(\varepsilon_1) = \mathsf{Pre}\ * \ \wedge r(\varepsilon_2) = \mathsf{Pre}\, * \ \Rightarrow\ \ell \le \dot{q}(\varepsilon_1 \cdot \varepsilon_2)}{\ell \lhd \dot{\mathrm{r}}\ \mathsf{exn}^{\dot{q}}}
$$

$$
\frac{\begin{array}{c}\forall c \in \mathcal{C},\ r_1(c) = \mathsf{Pre}\,t_1 \ \wedge\ r_2(c) = \mathsf{Pre}\,t_2 \ \Rightarrow\ \ell \lhd (t_1, t_2) \\ \forall c_1 \cdot c_2 \in \mathcal{C}^2,\ r_1(c_1) = \mathsf{Pre}\ * \ \wedge r_2(c_2) = \mathsf{Pre}\, * \ \Rightarrow\ \ell \le q_1(c_1 \cdot c_2) \sqcap q_2(c_1 \cdot c_2)\end{array}}{\ell \lhd ({r_1}^{[q_1]}, {r_2}^{[q_2]})}
$$

Figure 8: Guards

The typing rules for $\mathrm{MLIF}_{\mathcal{E}}$ are given in figure 9. Because we do not intend to establish a subject reduction theorem, we restrict our attention to *source* expressions, i.e. $\mathrm{ML}_{\mathcal{E}}$ expressions which do not contain memory locations. Thus typing judgements do not involve a memory environment. As usual, we distinguish two forms of judgements: $\Gamma \vdash v : t$ and $pc, \Gamma \vdash e : \mathrm{r}^{[q]}$. The former deals with values and involves a simple types (because values cannot raise any exception). The latter associates to an expression an effect $r^{[q]}$, describing what kind of result (value or exceptions) it may produce: $r(\eta)$ is Abs if the described expression never reduces to a value and is $\mathsf{Pre}\,t$ if it may produce value of type t. By the same, for every expression name ε, $r(\varepsilon)$ is Abs if the expression never raises any exception named ε and is $\mathsf{Pre}\,t$ if it may raise such an exception with an argument of type t. $q(c_1 \cdot c_2)$ is as expected an approximation of the level of the informations you leak by revealing that the expression has produced a result of name c_1 (value or exception) rather than c_2.

▶ **Lemma 3.3 (Soundness)** $\Gamma \vdash v : \mathrm{t}$ *implies* $[\![\Gamma]\!] \vdash v : [\![\mathrm{t}]\!]$. $pc, \Gamma \vdash e : \mathrm{r}^{[q]}$ *implies* $pc, [\![\Gamma]\!] \vdash e : [\![\mathrm{r}^{[q]}]\!]$.

Proof By induction on the input derivation. Cases E-BIND$_{\mathcal{E}}$ and E-HANDLEALL$_{\mathcal{E}}$ are detailed on figure 10. □

▶ **Theorem 3.4 (Non-interference)** *Choose $\ell, h \in \mathcal{L}$ such that $h \not\le \ell$. Let $h \lhd t$. Assume $(x \mapsto t) \vdash e : *^{[\partial\ell]}$, where e is a $ML_{\mathcal{E}}$ expression. If $\vdash v_i : t$ and $e[x \Leftarrow v_i] \rightarrow^{\star} o_i$, for $i \in \{1, 2\}$, then o_1 and o_2 are outcomes of the same kind (i.e. o_1 are o_2 are two values or two raised exceptions of the same name).*

Proof By lemma 3.3 and theorem 2.13. □

3.4 Back to MLIF_0

We now show how the type system MLIF_0 described in [1] as a type system for $\mathrm{ML}_{\mathcal{E}}$ may be considered as a restriction of $\mathrm{MLIF}_{\mathcal{E}}$. To achieve this comparaison, we will exhibit a *translation* of MLIF_0 judgements into $\mathrm{MLIF}_{\mathcal{E}}$ ones. This translation which is rather straightforward is however complicated by two superficial choices we made throughout the presentation of MLIF_0:

Values

v-Unit$_{\mathcal{E}}$

$$\Gamma \vdash () : \mathsf{unit}$$

v-Var$_{\mathcal{E}}$

$$\frac{\mathrm{t} \in \Gamma(x)}{\Gamma \vdash x : \mathrm{t}}$$

v-Abs$_{\mathcal{E}}$

$$\frac{pc, \Gamma[x \mapsto \mathrm{t}'] \vdash e : \mathrm{r}^{[q]}}{\Gamma \vdash \lambda x.e : (\mathrm{t}' \xrightarrow{pc} \mathrm{r}^{[q]})^{\ell}}$$

v-Exn$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : \mathrm{t}}{\Gamma \vdash \varepsilon\, v : (\varepsilon : \mathsf{Pre}\,\mathrm{t}; *)\ \mathsf{exn}^{*}}$$

v-Sub$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : \mathrm{t}' \qquad \mathrm{t}' \le \mathrm{t}}{\Gamma \vdash v : \mathrm{t}}$$

Expressions

e-Value$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : \mathrm{t}}{*, \Gamma \vdash v : (\eta : \mathsf{Pre}\,\mathrm{t}; *)^{[*]}}$$

e-Raise$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : \dot{\mathrm{r}}\ \mathsf{exn}^{\dot{q}} \qquad \dot{q} \le_{\mathcal{E}} q}{*, \Gamma \vdash \mathsf{raise}\ v : (\eta : *; \dot{\mathrm{r}})^{[q]}}$$

e-App$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v_1 : (\mathrm{t}' \xrightarrow{pc \sqcup \ell} \mathrm{r}^{[q]})^{\ell} \qquad \Gamma \vdash v_2 : \mathrm{t}' \qquad \ell \lhd \mathrm{r}^{[q]}}{pc, \Gamma \vdash v_1\, v_2 : \mathrm{r}^{[q]}}$$

e-Ref$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : \mathrm{t} \qquad pc \lhd \mathrm{t}}{pc, \Gamma \vdash \mathsf{ref}\ v : (\eta : \mathsf{Pre}\,(\mathrm{t}\ \mathsf{ref}^{*}); *)^{[*]}}$$

e-Assign$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v_1 : \mathrm{t}\ \mathsf{ref}^{\ell} \qquad \Gamma \vdash v_2 : \mathrm{t} \qquad pc \sqcup \ell \lhd \mathrm{t}}{pc, \Gamma \vdash v_1 := v_2 : (\eta : \mathsf{Pre}\,\mathsf{unit}; *)^{[*]}}$$

e-Deref$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : \mathrm{t}'\ \mathsf{ref}^{\ell} \qquad \mathrm{t}' \le \mathrm{t} \qquad \ell \lhd \mathrm{t}}{pc, \Gamma \vdash\ !v : (\eta : \mathsf{Pre}\,\mathrm{t}; *)^{[*]}}$$

e-Let$_{\mathcal{E}}$

$$\frac{\Gamma \vdash v : s \qquad pc, \Gamma[x \mapsto s] \vdash e : \mathrm{r}^{[q]}}{pc, \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \mathrm{r}^{[q]}}$$

e-Bind$_{\mathcal{E}}$

$$\frac{\begin{array}{c} pc, \Gamma \vdash e_1 : \mathrm{r}_1{}^{[q_1]} \qquad pc \sqcup \ell, \Gamma[x \mapsto \mathrm{t}] \vdash e_2 : \mathrm{r}_2{}^{[q_2]} \qquad \mathrm{r}_1(\eta) \le \mathsf{Pre}\,\mathrm{t} \qquad \mathrm{r}_1{}^{[q_1]} \le_{\mathcal{E}} \mathrm{r}_1'{}^{[q_1']} \\ \bigsqcup\{q_1(\eta \cdot \varepsilon) \mid \varepsilon \in \mathcal{E}\} \le \ell \qquad \mathrm{r}_1 \Cap \{\eta\}\ ?\ (\mathrm{r}_2{}^{[q_2]} \le \mathrm{r}^{[q]}) \qquad \mathrm{r}_1'{}^{[q_1']} \le_{\mathcal{E}} \mathrm{r}^{[q]} \qquad \ell \lhd (\mathrm{r}_1'{}^{[q_1']}, \mathrm{r}_2{}^{[q_2]}) \end{array}}{pc, \Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \mathrm{r}^{[q]}}$$

e-Handle$_{\mathcal{E}}$

$$\frac{\begin{array}{c} pc, \Gamma \vdash e_1 : \mathrm{r}_1{}^{[q_1]} \qquad pc \sqcup \ell, \Gamma[x \mapsto \mathrm{t}] \vdash e_2 : \mathrm{r}_2{}^{[q_2]} \qquad \mathrm{r}_1(\varepsilon) \le \mathsf{Pre}\,\mathrm{t} \qquad \mathrm{r}_1{}^{[q_1]} \le_{\mathcal{C}\setminus\{\varepsilon\}} \mathrm{r}_1'{}^{[q_1']} \\ \bigsqcup\{q_1(\varepsilon \cdot c) \mid c \in \mathcal{C}\setminus\{\varepsilon\}\} \le \ell \qquad \mathrm{r}_1 \Cap \{\varepsilon\}\ ?\ (\mathrm{r}_2{}^{[q_2]} \le \mathrm{r}^{[q]}) \qquad \mathrm{r}_1'{}^{[q_1']} \le_{\mathcal{C}\setminus\{\varepsilon\}} \mathrm{r}^{[q]} \qquad \ell \lhd (\mathrm{r}_1'{}^{[q_1']}, \mathrm{r}_2{}^{[q_2]}) \end{array}}{pc, \Gamma \vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : \mathrm{r}^{[q]}}$$

e-HandleAll$_{\mathcal{E}}$

$$\frac{\begin{array}{c} pc, \Gamma \vdash e_1 : \mathrm{r}_1{}^{[q_1]} \qquad pc \sqcup \ell, \Gamma[x \mapsto \dot{\mathrm{r}}\ \mathsf{exn}^{\dot{q}}] \vdash e_2 : \mathrm{r}_2{}^{[q_2]} \qquad \dot{\mathrm{r}}^{[\dot{q}]} \le_{\mathcal{E}} \mathrm{r}_1{}^{[q_1]} \\ \bigsqcup\{q_1(\eta \cdot \varepsilon) \mid \varepsilon \in \mathcal{E}\} \le \ell \qquad \mathrm{r}_1 \Cap \mathcal{E}\ ?\ (\mathrm{r}_2{}^{[q_2]} \le \mathrm{r}^{[q]}) \qquad \mathrm{r}_1(\eta) \le \mathrm{r}(\eta) \qquad \ell \lhd ((\eta : \mathrm{r}_1(\eta); \partial\mathsf{Abs})^{[\partial\bot]}, \mathrm{r}_2{}^{[q_2]}) \end{array}}{pc, \Gamma \vdash e_1\ \mathsf{handle}\ x \succ e_2 : \mathrm{r}^{[q]}}$$

e-Sub$_{\mathcal{E}}$

$$\frac{pc, \Gamma \vdash e : \mathrm{r}'^{[q']} \qquad \mathrm{r}'^{[q']} \le \mathrm{r}^{[q]}}{pc, \Gamma \vdash e : \mathrm{r}^{[q]}}$$

Figure 9: Typing rules

Rule E-BIND$_{\mathcal{E}}$

$$
\dfrac{
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_y^1 \vdash y' : \Sigma^{q_1'}\, r_1'}\ \text{v-Var}}{}}{}}{}}{\;}
}{\;}
$$

$$
\dfrac{
\begin{array}{c}
\dfrac{}{\Gamma_y \vdash y : \Sigma^{q_1}\, r_1}\ \text{v-Var}
\qquad
\text{e-Bind}\ \dfrac{\begin{array}{c}\text{e-Proj}\ \dfrac{\text{v-Var}\ \dfrac{}{\Gamma_y^2 \vdash y' : \Sigma^{*}\,(\eta : \mathsf{Pre}\, t; \partial\mathsf{Abs})}}{pc \sqcup \ell, \Gamma_y^2 \vdash \bar{\eta}\, y' : t} \\ \vdots \qquad pc \sqcup \ell, \Gamma_y^2[x \mapsto t] \vdash [\![e_2]\!] : \Sigma^{q_2}\, r_2\end{array}}{pc \sqcup \ell, \Gamma_y^2 \vdash \mathsf{bind}\ x = \bar{\eta}\, y'\ \mathsf{in}\ [\![e_2]\!] : \Sigma^{q_2}\, r_2}
\qquad
\dfrac{\dfrac{}{\Gamma_y^1 \vdash y' : \Sigma^{q_1'}\, r_1'}\ \text{v-Var}}{pc \sqcup \ell, \Gamma_y^1 \vdash y' : \Sigma^{q_1'}\, r_1'}\ \text{e-Value}
\\
r_1 \leq_{\mathcal{E}} r_1' \qquad q_1 \leq_{\mathcal{E}} q_1' \qquad \sqcup\{q_1(\eta \cdot \varepsilon) \mid \varepsilon \in \mathcal{E}\} \leq \ell
\\
\ell \lhd (\Sigma^{q_1'}\, r_1', \Sigma^{q_2}\, r_2) \qquad r_1 \Cap \{\eta\}\ ?\ (\Sigma^{q_2}\, r_2 \leq \Sigma^{q}\, r) \qquad \Sigma^{q_1'}\, r_1' \leq_{\mathcal{E}} \Sigma^{q}\, r
\end{array}
}{pc, \Gamma_y \vdash y\ \mathsf{case}\ [\eta : y' \mapsto \mathsf{bind}\ x = \bar{\eta}\, y'\ \mathsf{in}\ [\![e_2]\!] \mid \mathcal{E} : y' \mapsto y'] : \Sigma^{q}\, r}\ \text{e-Match}
$$

$$
\dfrac{pc, \Gamma \vdash [\![e_1]\!] : \Sigma^{q_1}\, r_1 \qquad \vdots}{pc, \Gamma \vdash \underbrace{\mathsf{bind}\ y = [\![e_1]\!]\ \mathsf{in}\ (y\ \mathsf{case}\ [\eta : y' \mapsto \mathsf{bind}\ x = \bar{\eta}\, y'\ \mathsf{in}\ [\![e_2]\!] \mid \mathcal{E} : y' \mapsto y'])}_{[\![\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2]\!]} : \Sigma^{q}\, r}\ \text{e-Bind}
$$

$\Gamma_y = \Gamma[y \mapsto \Sigma^{q_1}\, r_1]$
$\Gamma_y^1 = \Gamma_y[y' \mapsto \Sigma^{q_1'}\, r_1']$
$\Gamma_y^2 = \Gamma_y[y' \mapsto \Sigma^{\partial\bot}\,(\mathsf{Pre}\, t; \partial\mathsf{Abs})]$

Rule E-HANDLEALL$_{\mathcal{E}}$

$$
\dfrac{
\begin{array}{c}
\text{v-Var}\ \dfrac{}{\Gamma_y \vdash y : \Sigma^{q_1}\, r_1}
\qquad
pc \sqcup \ell, \Gamma_y^2 \vdash [\![e_2]\!] : \Sigma^{q_2}\, r_2
\qquad
\dfrac{r_1(\eta) \leq r(\eta)}{\Sigma^{\partial\bot}\, r_1' \leq \Sigma^{q}\, r}
\qquad
\dfrac{\dfrac{}{\Gamma_y^1 \vdash y' : \Sigma^{\partial\bot}\, r_1'}\ \text{v-Var}}{pc \sqcup \ell, \Gamma_y^1 \vdash y' : \Sigma^{\partial\bot}\, r_1'}\ \text{e-Value}
\\
r_1 \leq_{\mathcal{E}} \dot{r} \qquad q_1 \leq_{\mathcal{E}} \dot{q} \qquad \sqcup\{q_1(\eta \cdot \varepsilon) \mid \varepsilon \in \mathcal{E}\} \leq \ell \qquad \ell \lhd (\Sigma^{\partial\bot}\, r_1', \Sigma^{q_2}\, r_2) \qquad r_1 \Cap \mathcal{E}\ ?\ (\Sigma^{q_2}\, r_2 \leq \Sigma^{q}\, r)
\end{array}
}{pc, \Gamma_y \vdash y\ \mathsf{case}\ [\eta : y' \mapsto y' \mid \mathcal{E} : x \mapsto [\![e_2]\!]] : \Sigma^{q}\, r}\ \text{e-Match}
$$

$$
\dfrac{pc, \Gamma \vdash [\![e_1]\!] : \Sigma^{q_1}\, r_1 \qquad \vdots}{pc, \Gamma \vdash \underbrace{\mathsf{bind}\ y = [\![e_1]\!]\ \mathsf{in}\ (y\ \mathsf{case}\ [\eta : y' \mapsto y' \mid \mathcal{E} : x \mapsto [\![e_2]\!]])}_{[\![e_1\ \mathsf{handle}\ x \succ e_2]\!]} : \Sigma^{q}\, r}\ \text{e-Bind}
$$

$\Gamma_y = \Gamma[y \mapsto \Sigma^{q_1}\, r_1]$
$\Gamma_y^1 = \Gamma_y[y' \mapsto \Sigma^{\partial\bot}\, r_1']$
$\Gamma_y^2 = \Gamma_y[y' \mapsto \Sigma^{\dot{q}}\,(\eta : \mathsf{Abs}; \dot{r})]$
$r_1' = (\eta : r_1(\eta); \partial\mathsf{Abs})$

Figure 10: Deriving ML$_{\mathcal{E}}$ judgements from ML$_{\Sigma}$ ones

Effects

$$\llbracket t^0, r^0 \rrbracket_{pc} = r^{[q]} \quad \text{where} \begin{cases} r(\eta) & = & \mathsf{Pre}\, t^0 & \\ r(\varepsilon) & = & \mathsf{Pre}\, \llbracket \mathit{typexn}(\varepsilon) \rrbracket & \text{if } \mathsf{Pre}\, pc \leq r^0(\varepsilon) \\ & & \mathsf{Abs} & \text{otherwise} \\ q(\varepsilon_1 \cdot \varepsilon_2) & = & \text{level}\, r^0(\varepsilon_1) \sqcup \text{level}\, r^0(\varepsilon_2) & \\ q(\eta \cdot \varepsilon) & = & \text{level}\, r^0(\varepsilon) & \end{cases}$$

Types

$$\begin{array}{rcl} \llbracket \mathsf{unit} \rrbracket & = & \mathsf{unit} \\ \llbracket \mathsf{int}^\ell \rrbracket & = & \mathsf{int}^\ell \\ \llbracket (t^{0\prime} \xrightarrow{pc\ [r^0]} t^0)^\ell \rrbracket & = & (\llbracket t^{0\prime} \rrbracket \xrightarrow{pc} \llbracket t^0, r^0 \rrbracket_{pc})^\ell \\ \llbracket t^0\ \mathsf{ref}^\ell \rrbracket & = & \llbracket t^0 \rrbracket\ \mathsf{ref}^\ell \\ \llbracket r^0\ \mathsf{exn}^\ell \rrbracket & = & \Sigma^{\dot{q}}\, \dot{\mathrm{r}} \quad \text{where} \begin{cases} \dot{\mathrm{r}}(\varepsilon) & = & \mathsf{Pre}\, \llbracket \mathit{typexn}(\varepsilon) \rrbracket & \text{if } r^0(\varepsilon) = \mathsf{Pre}\, * \\ & & \mathsf{Abs} & \text{otherwise} \\ \dot{q}(\varepsilon_1 \cdot \varepsilon_2) & = & \text{level}\, r^0(\varepsilon_1) \sqcup \text{level}\, r^0(\varepsilon_2) & \end{cases} \end{array}$$

Levels

$$\begin{array}{rcl} \text{level}\, \mathsf{Abs} & = & \bot \\ \text{level}(\mathsf{Pre}\, pc) & = & pc \end{array}$$

Figure 11: Encoding MLIF$_0$ effects and types into ML$_\mathcal{E}$ ones

- The type of exceptions arguments are global in MLIF$_0$: there exists a total mapping *typexn* from exception names into types. In order to keep the translation as simple as possible and not have to deal with recursive types, we will assume that these types do not involе any arrow type.
- In MLIF$_0$, effects (in the meaning of this paper) are divided into two distinct parts: a type t (describing the normal result of an evaluation) and a row (describing the exceptional results of an evaluation). While types are normally guarded (as in V-BRACKET and V-BRACKET rules), rows are never because they are directly generated with the appropriate levels by V-RAISE.

Figure 11 defines the translation $\llbracket t, r \rrbracket_{pc}$ of an effect from MLIF$_0$ (t, r). It depends on the current pc since we have to "erase" parasitic fields $\mathsf{Pre}\,\ell$ with $pc \not\leq \ell$ which cannot correspond to a possible outcome of the described expression. The translation $\llbracket t \rrbracket$ of a type t is straightforward. $\llbracket \cdot \rrbracket$ is extended to polytypes and environments.

▶ **Theorem 3.5** *If $pc, \Gamma \vdash e : t^0\ [r^0]$ (resp. $\Gamma \vdash v : t$) is a valid judgement in* MLIF$_0$ *then $pc, \llbracket \Gamma \rrbracket \vdash e : \llbracket t^0, r^0 \rrbracket_{pc}$ (resp. $\llbracket \Gamma \rrbracket \vdash v : \llbracket t \rrbracket$) is derivable in* MLIF$_\mathcal{E}$.

Proof By induction on the input derivation.

∘ *Case* E-BIND. e is $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$. By E-BIND, $pc, \Gamma \vdash e_1 : t_1^0\ [r_1^0]$ and $pc \sqcup (\sqcup r_2^0), \Gamma \vdash e_2 : t^0\ [r_2^0]$ hold and r^0 is $r_1^0 \sqcup r_2^0$. Let $\ell = \sqcup r_2^0$, ${r_1}^{[q_1]} = \llbracket t_1^0, r_1^0 \rrbracket_{pc}$, ${r_2}^{[q_2]} = \llbracket t^0, r_2^0 \rrbracket_{pc \sqcup \ell}$ and $r^{[q]} = \llbracket t^0, r^0 \rrbracket_{pc}$. By induction hypothesis, $pc, \llbracket \Gamma \rrbracket \vdash e_1 : {r_1}^{[q_1]}$ and $pc \sqcup \ell, \llbracket \Gamma \rrbracket [x \mapsto \llbracket t_1^0 \rrbracket] \vdash e_2 : {r_2}^{[q_2]}$ hold.

Let r_1' (resp. q_1') be the "restriction" of r_1 (resp. q_1) to $\mathcal{E}$, completed by Abs (resp. $\bot$) fields. By definition of $\llbracket \cdot, \cdot \rrbracket_.$, we have $\sqcup \{ q_1(\eta \cdot \varepsilon) \mid \varepsilon \in \mathcal{E} \} \leq \ell$, $r_1(\eta) \leq \mathsf{Pre}\, t_1^0$, ${r_2}^{[q_2]} \leq r^{[q]}$ and $\ell \lhd ({r_1'}^{[q_1']}, {r_2}^{[q_2]})$. We conclude by applying E-BIND$_\mathcal{E}$. □

References

[1] François Pottier and Vincent Simonet. Information flow inference for ML. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-popl02.ps.gz, July 2001.

[2] François Pottier and Vincent Simonet. Information flow inference for ML. Full version. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-popl02-long.ps.gz, July 2001.

Annexe

Comparaison d'outils de vérification

(Stage de première année)

Résumé

Ce rapport est le compte-rendu du stage de maîtrise que j'ai effectué au LIAFA de juin à septembre 1999. Le but de ce stage était de réaliser des outils formant des passerelles entre des vérificateurs afin en particulier de pouvoir comparer leurs résultats et leurs performances.
En rédigeant ce mémoire, je me suis fixé deux objectifs : tout d'abord montrer les connaissances que j'ai acquises – c'est principalement le rôle de la première partie qui effectue rapidement un vaste tour d'horizon du domaine étudié – et présenter mes réalisations – ce que font la deuxième et la troisième parties.
Ces deux derniers chapitres se veulent être à la fois une présentation des outils réalisés et un premier mode d'emploi à l'usage de ceux qui seront amenés à les utiliser.

Table des matières

Introduction

La complexité des systèmes informatiques et électroniques s'accroît de telle sorte qu'il est de plus en plus difficile de cerner leur fonctionnement dans leur ensemble. Parrallèlement leur place dans la "vie quotidienne" ne cesse de grandir de telle sorte que des dysfonctionnements peuvent avoir des conséquences économiques, sociales ou humaines inacceptables.

Ces deux phénomènes rendent nécessaire le développement de techniques permettant de prouver la validité d'algorithmes, d'implantations, de conceptions... Le simple contrôle par test heuristique ne peut plus suffire, il est souvent à la fois trop coûteux et insuffisamment fiable. Plusieurs approches sont possibles, de la preuve mathématique formelle à la vérification exhaustive.

Le model-checking *en est une. Il permet de vérifier de manière automatique des systèmes de natures très diverses. Après avoir présenté ses principales caractéristiques, nous tenterons de tisser des liens entre trois* model checkers*, MEC, SMV* et *SPIN sur deux plans : la modélisation dans un premier temps et la spécification dans un second.*

Chapitre 1

Vérification de modèles

1.1 Présentation

L'expression *model-checking*, vérification de modèle, désigne un ensemble de techniques et d'outils qui permettent de prouver – ou de contredire – des propriétés de correction relatives à des systèmes. La nature de ces systèmes peut être variée : circuits intégrés, systèmes interactifs, protocoles de communication... Ceux qui se prêtent le mieux à cette approche de preuve sont ceux qui se modélisent facilement par un automate fini, ou une construction plus élaborée qui s'y ramène.

Plusieurs applications industrielles de taille notoire de ces techniques ont déjà été mises en œuvre. Elles ont permis de déceler des erreurs sur des systèmes dont la complexité combinatoire rendait une analyse classique difficile. Dans une telle vérification, on distingue généralement trois étapes :

- *La modélisation.* Il s'agit de décrire l'objet étudié de manière formelle sous une forme analysable. La méthode de base consiste à définir un automate dont les états correspondent à ceux du système. On préfère généralement des formalisations de plus hauts niveaux dont il est plus facile de se "convaincre" de la conformité au système réel et qui se ramènent ensuite, par "compilation", à des automates.
- *La spécification.* Après avoir posé le problème, il faut exprimer les propriétés à vérifier. Celles-ci portent généralement sur les états du système. On peut pour cela utiliser des *logiques temporelles* qui se prêtent particulièrement bien d'une part à la description de ces systèmes discrets et d'autre part à la vérification.
- *La vérification.* On peut enfin passer à la preuve. Le principe général est d'effectuer une analyse exhaustive, rendue possible par le caractère fini du système. Les algorithmes utilisés doivent cependant être relativement fins car les complexités en temps et en espace sont souvent problématiques.

  En plus de donner une réponse booléenne à la question posée, les programmes disponibles fournissent généralement un exemple ou contre-exemple, souvent utile pour corriger l'erreur.

Un des grands intérêts de la "vérification de modèles" réside dans le fait qu'elle fournit des réponses complètes et de manière automatique. La portée de ces résultats est cependant limitée, comme par exemple en physique, par la question de la validité des modèles formels utilisés.

1.2 Modélisation des systèmes étudiés

1.2.1 Système de transitions

Avant de donner une définition précise, considérons un exemple simple : celui d'un interrupteur. Il peut être dans deux *états* : *ouvert* ou *fermé*. On peut passer d'un état à l'autre en effectuant une *transition* : *ouvrir* qui fait passer de l'état *fermé* à l'état *ouvert* et *fermer* qui fait passer de l'état *ouvert* à l'état *fermé*. On rajoute généralement deux transitions qui bouclent chacune sur un état et qui correspondent au cas où le système n'évolue pas.

On peut représenter ce modèle d'un interrupteur par le schéma de la figure 1.1.

Automate Un *automate* ou *système de transitions* est un triplet $A = (Q, T, I)$:

- Q est l'ensemble des états qui peut être fini ou infini,

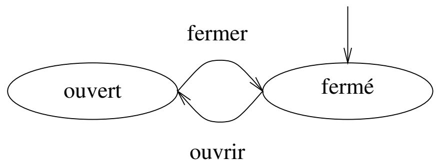


FIG. 1.1: Automate d'un interrupteur

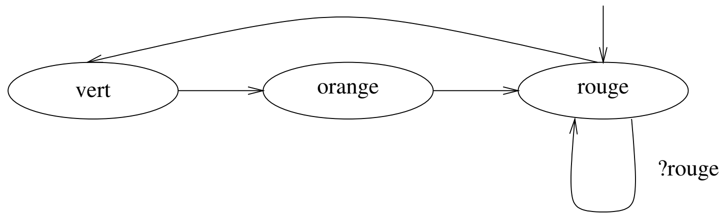


FIG. 1.2: Automate d'un feu tricolore

- T est l'ensemble des transitions. Une transition est un triplet $t = (q, e, q')$: $q \in Q$ est l'origine de la transition (l'état dont elle part) et $q' \in Q$ l'extrémité (l'état où elle mène), e est l'étiquette de la transition ; elle permet fondemmentalement de distinguer formellement deux transitions qui ont mêmes origine et extrémité.
- I est l'ensemble des états de départ de l'automate.

Les chemins On appelle chemin de longueur n ($n > 0$) dans un système de transitions A, une suite de transitions $t_1, t_2, ..., t_n$ telle que pour tout i, l'extrémité de t_i et l'origine de t_{i+1} soient le même état. On peut également considérer des chemins infinis $t_1, t_2, ..., t_n, ...$. Une exécution est un chemin dont le premier état est dans l'ensemble I.
Un chemin représente une évolution du système d'un état à un autre.

Un exemple concret Considérons un carrefour où se croisent deux routes, régulé par des feux tricolores commandés par des "boucles" détectant l'arrivée de voitures. Les feux opposés marchant par paire, le système ne comporte en fait que quatre entités distinctes qui interagissent entre elles :
- Deux (paires de) feux tricolores. Chacun d'entre-eux peut prendre trois états : *vert*, *orange* ou *rouge* et peut être modélisé par l'automate de la figure 1.2.
- Deux (paires de) boucles qui peuvent être soit *actives*, soit *inactives* (figure 1.3).

Pour simplifier, on supposera que les feux sont exclusivement commandés par les boucles (il n'y a en particulier pas de *time-out*).
Le système global peut à priori prendre $3 \times 3 \times 2 \times 2 = 36$ états distincts (ces états étant des quadruplets des états des quatres automates). A chaque pas du système global, les quatre sous-sytèmes doivent effectuer une transition. On sous-entend généralement au niveau de chacun des états de chaque système une ϵ-transition qui exprime qu'à chaque pas un des composants peut ne pas changer d'état. C'est le modèle asynchrone.
Cependant les comportements respectifs de nos quatre sous-sytèmes ne sont pas indépendants. Ce qui nous amène à introduire la notion de synchronisation.

1.2.2 Produit d'automates synchronisés

Théoriquement, tout système fini (dans le sens "qui a un nombre d'états discernables finis") peut être représenté par un automate fini. Cependant, utilisée de manière brute, cette approche n'est pas la plus appropriée lorsque les systèmes ont une taille importante, plusieurs milliers d'états par exemple. Une méthode

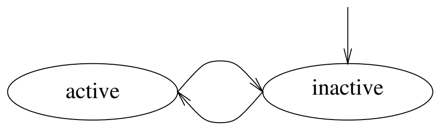


FIG. 1.3: Automate d'une boucle

Feu 1	**Feu 2**
$vert \to orange$	–
$orange \to rouge$	–
$rouge \to vert$	$?rouge$
–	$vert \to orange$
–	$orange \to rouge$
$?rouge$	$rouge \to vert$

FIG. 1.4: Synchronisation des transitions entre les deux feux

Feu 1	**Feu 2**	**Boucle 1**	**Boucle 2**
$vert \to orange$	–	–	–
$orange \to rouge$	–	–	–
$rouge \to vert$	$?rouge$	–	$active \to inactive$
–	$vert \to orange$	–	–
–	$orange \to rouge$	–	–
$?rouge$	$rouge \to vert$	–	$active \to inactive$
–	–	$inactive \to active$	–
–	–	–	$inactive \to active$

FIG. 1.5: Synchronisation des feux avec les boucles

consite à découper le système étudié en un certain nombre de sous-systèmes que l'on modélise chacun par un automate puis à décrire les relations voulues entre ces entités (c'est le rôle de la *contrainte de modélisation*).

Automate produit Considérons n ($n \geq 1$) automates $A_1, \ldots, A_n$. On peut construire leur produit $A = (Q, T, I)$ en posant :
- $Q = Q_1 \times \ldots \times Q_n$. Les états de l'automate produit sont des n-uplets dont la *i-ème* composante est un état de l'automate A_i.
- $T = \{((q_1, \ldots, q_n), (e_1, \ldots, e_n), (q'_1, \ldots, q'_n)) \ / \ \forall\ i, (q_i, e_i, q'_i) \in T_i\}$. Chaque transition du produit correspond au passage d'une transition pour chacun des automates qui le constitue.
- Et naturellement $I = I_1 \times \ldots \times I_n$.

Il est immédiat que le nombre d'états du produit est égal au produit des nombres d'états des différents systèmes de transitions mis en jeux. On aperçoit d'ailleurs ici l'origine d'une des principales difficultés du *model-checking*, l'explosion combinatoire : même s'ils ne comportent que des sous-système relativement simples, les systèmes globaux étudiés possèdent généralement un nombre très important d'états.

Synchronisation Dans le produit général décrit ci-dessus, un certain nombre des transitions ne correspond pas au comportement *global* des différents sous-systèmes. Pour exprimer les relations entre ces entités, on limite l'ensemble des transitions du produit en ne gardant que la sous-partie de T formée des transitions qui respectent les relations entre les différents sous-systèmes.
La manière la plus simple d'exprimer cette contrainte de synchronisation entre les différents automates du produit est tout simplement de faire la liste des "transitions-produits" qui sont possibles.

Exemple Reprenons notre exemple précédent. Le comportement des deux (paires de) feux est bien entendu lié : il ne s'agit pas que les automobilistes venant des deux routes puissent simultanément traverser le carrefour. Il faut donc exprimer cette contrainte lorsqu'on effectue le produit. Si l'on considère les deux feux simultanément, les seuls couples de transitions possibles sont ceux décrits par la figure 1.4.
On peut ensuite ajouter au produit les deux boucles qui se synchronisent avec les deux feux. Il faut bien sûr tenir compte des deux transitions qui sont effectuées lorsque une automobile arrive sur l'une des routes. C'est la "commande" extérieure du système qui est matérialisée par les deux dernières lignes du tableau de la figure 1.5.
Ces contraintes réduisent très fortement le nombre de transitions que le système complet peut passer et par la même le nombre d'états accessibles. Elles permettent d'exprimer les communications qui interviennent entre les différents processus mis en jeux.

Synchronisation par variables partagées Une manière plus élaborée de spécifier les contraintes de synchronisation est d'ajouter au modèle un certains nombre de variables. Ces variables, qui peuvent être par exemple des entiers, des booléens, ..., doivent évoluer dans des ensembles finis (et correspondent ainsi facilement à des automates finis).
On ajoute à chaque transition deux clauses : la première condition est une expression en les variables du système qui doit être vérifiée pour que la transition puisse être effectuée ; la seconde est une liste d'affectations, c'est-à-dire de modifications du contenu des variables, qui sont réalisées lorsque la transition est passée. Nous formaliserons ces idées plus précisément en décrivant un langage de modélisation dans la seconde partie.

1.3 Spécification des propriétés

1.3.1 Motivation

Une fois le système décrit, on souhaite généralement exprimer de la manière la plus simple possible les propriétés que l'on souhaite qu'il ait. Reprenons par exemple notre système de feux. On pourrait par exemple se demander si :
– A aucun moment, les deux feux ne sont simultanément au vert,
– Toute activation d'une boucle doit aboutir à un passage au vert du feu correspondant.
Une façon de formaliser ces expressions serait d'utiliser des formules temporelles sembables à celles que nous utilisons en physique :

$$\forall\ t,\ \neg(couleur(feu_1) = vert \wedge couleur(feu_2) = vert)$$

$$\forall\ t,\ etat(boucle_i) = active \Rightarrow \exists\ t' \geq t,\ couleur(feu_i) = vert$$

Cette écriture a levé un certain nombre d'ambiguïtés des phrases écrites en français (par exemple, "*doit aboutir*" a été retranscrit par "$\exists\ t' \geq t$"). Cependant ces notations sont relativement lourdes et peu adaptées à la situation étudiée.

1.3.2 Un exemple de logique temporelle : CTL

La logique CTL, pour *computation tree logic* a été introduite par Clarke, Emerson et Sistla précisément pour exprimer des propriétés servant à vérifier des systèmes de processus représentés par des systèmes de transitions.

Syntaxe Commençons par donner la syntaxe d'une expression de la logique CTL. En logique CTL, il existe en fait deux sortes de formules : des formules de chemins et des formules d'états :

$$\begin{aligned} formule_{etat} :== \ & \text{proposition atomique} \\ & \neg formule_{etat} \\ & formule_{etat} \ \wedge \ formule_{etat} \\ & formule_{etat} \ \vee \ formule_{etat} \\ & \mathsf{E}\ formule_{chemin} \\ & \mathsf{A}\ formule_{chemin} \end{aligned}$$

$$\begin{aligned} formule_{chemin} :== \ & \mathsf{X}\ formule_{etat} \\ & \mathsf{F}\ formule_{etat} \\ & \mathsf{G}\ formule_{etat} \\ & formule_{etat}\mathsf{U}formule_{etat} \\ & formule_{etat}\mathsf{W}formule_{etat} \end{aligned}$$

Les seules formules que l'on considère en tant que telles sont les formules d'états, les formules de chemins ne servant que d'intermédiaires. Une autre façon – moins naturelle me semble-t-il – d'introduire cette syntaxe est donc de ne considérer que des formules d'états et de définir 10 opérateurs : EX, EF, EG, E[.U.], E[.W.] et leurs analogues en remplaçant E par A.

Sémantique Il nous faut maintenant définir la relation de satisfaction, ou plutôt simultanément deux relations de satisfaction : la satisfaction d'une formule d'état par un état ($q \models \phi_e$) et la satisfaction d'une formule de chemin par un chemin ($\sigma \models \phi_c$).
Avant tout, précisons quelques notations. Si σ est un chemin de l'automate, $|\sigma|$ dénote sa longueur (en nombre d'états) éventuellement infinie et $\sigma(i)$, pour $0 \leq i < |\sigma|$, désigne son i-ème état. La satisfaction se définit naturellement par induction (figure 1.6).

$\sigma \models \mathsf{X}\phi_e$	"L'état suivant satisfait ϕ_e" : $|\sigma| > 1 \ \wedge\ \sigma(1) \models \phi_e$
$\sigma \models \mathsf{F}\phi_e$	"Un jour ϕ_e" : $\exists\ i < |\sigma|,\ \sigma(i) \models \phi_e$
$\sigma \models \mathsf{G}\phi_e$	"Toujours ϕ_e" : $\forall\ i < |\sigma|,\ \sigma(i) \models \phi_e$
$\sigma \models \phi_e \mathsf{U} \phi'_e$	"ϕ_e jusqu'à ϕ'_e" : $\exists\ i_0 < |\sigma|,\ (\forall\ i < i_0,\ \sigma(i) \models \phi_e)\ \wedge\ (\sigma(i_0) \models \phi'_e)$
$\sigma \models \phi_e \mathsf{W} \phi'_e$	"ϕ_e jusqu'à ϕ'_e", au sens faible : $(\sigma \models \mathsf{G}\phi_e\ \wedge\ \sigma = \infty)\ \vee\ (\sigma \models \phi_e \mathsf{U} \phi'_e)$
$q \models \neg\phi_e$	Négation logique : $q \not\models \phi_e$
$q \models \phi_e\ \wedge\ \phi'_e$	Et logique : $(q \models \phi_e)\ \wedge\ (q \models \phi'_e)$
$q \models \phi_e\ \vee\ \phi'_e$	Ou logique : $(q \models \phi_e)\ \vee\ (q \models \phi'_e)$
$q \models \mathsf{E}\ \phi_c$	Quantification existentielle : $\exists\ \sigma,\ (\sigma(0) = q)\ \wedge\ (\sigma \models \phi_c)$
$q \models \mathsf{A}\ \phi_c$	Quantification universelle : $\forall\ \sigma,\ (\sigma(0) = q)\ \Rightarrow\ (\sigma \models \phi_c)$

FIG. 1.6: Définition de la satisfaction d'une formule

Notons toutefois que tous les opérateurs temporels peuvent se définir à partir des seuls X et U :

$$\begin{aligned}
\mathsf{F}\phi_e &= tt\mathsf{U}\phi_e \\
\mathsf{G}\phi_e &= \neg\mathsf{F}\neg\phi_e \\
&= \neg(tt\mathsf{U}\neg\phi_e) \\
\phi_e\mathsf{W}\phi'_e &= (\phi_e\mathsf{U}\phi'_e)\ \vee\ (\mathsf{G}(\phi_e\ \wedge\ \mathsf{EX}tt)) \\
&= (\phi_e\mathsf{U}\phi'_e)\ \vee\ \neg(tt\mathsf{U}\neg\phi_e)
\end{aligned}$$

Domaine de quantification et condition d'équité. La logique temporelle CTL fait intervenir deux quantificateurs E et A qui permettent d'exprimer des propriétés du type "*pour tout chemin*" ou "*il existe un chemin*". Il faut préciser sur quel ensemble portent ces quantifications, c'est à dire en fin de compte quels chemins on considère.
Remarquons tout d'abord que si on tient compte de tous les chemins de l'automate, certains opérateurs perdent de leur intérêt. Par exemple, AF devient l'identité. Nous nous intéresserons par la suite à trois domaines (ce qui donnera trois sémantiques différentes) :
- *Les chemins maximaux.* On ne considère que les chemins infinis ou finis et dont le dernier état est bloquant (aucune transition n'a cet état pour origine).
- *Les chemins infinis.* On ne tient compte que des chemins infinis.
- *Les chemins équitables.* On fixe une condition d'équité F (qui est un ensemble d'états). Un chemin est dit équitable si il passe un infinité de fois par un état de F. Dans la sémantique des chemins équitables, on ne considère que des chemins équitables.

Notons que la sémantique des chemins infinis est un cas particulier de la sémantique des chemins équitables, en prenant $F = Q$.

Exemple On peut écrire en logique CTL les deux formules précédentes :
- À aucun moment, les deux feux ne sont simultanément au vert : $\mathsf{AG}(feu_1 \neq vert \ \vee \ feu_2 \neq vert)$.
- Toute activation d'une boucle doit aboutir à un passage au vert du feu correspondant : $\mathsf{AG}(boucle_i = active \ \Rightarrow \ \mathsf{AF}(feu_i = vert))$.

1.4 Outils et vérification des propriétés

Chacun des outils que nous allons étudier comporte en fait deux parties relativement distinctes : un langage permettant de modéliser le système étudié et de spécifier les propriétés d'une part, un "moteur" de vérification d'autre part. De notre point de vue, c'est principalement au premier point que nous allons nous attacher.

1.4.1 MEC

MEC est un langage de relativement bas niveau. On décrit un système sous forme d'un produit d'automates synchronisés : dans un premier temps on définit chaque automate en faisant la liste de ses transitions (et par la même de ses états), puis on forme un produit en faisant la liste des n-uplets de transitions possibles.
L'avantage de cette approche est qu'elle permet à l'utilisateur de parfaitement cerner pas à pas la teneur de son modèle et la façon dont elle est traitée par le système. Il est ainsi assez facile de se convaincre de la validité d'une modélisation. Le revers de la médaille est que la description d'un système est souvent assez fastidieuse et peu agréable.
Une fois cette construction effectuée, *MEC* peut effectuer des calculs sur des ensembles d'états à partir d'équations de point fixes. Ceci permet d'effectuer la spécification et la vérification des propriétés. Ce mode d'expression est en particulier assez puissant pour écrire toute la logique temporelle CTL.

Exemple de modélisation Décrivons en *MEC* notre système de feux précédent. Il nous faut d'abord définir l'automate modélisant un feu :

```
transition_system feu < width = 0 >;
vert   |- vert_to_orange  -> orange;
orange |- orange_to_rouge -> rouge;
rouge  |- rouge_to_vert   -> vert;
rouge |- ?rouge -> rouge;
< initial = { rouge } >.
```

Ce mode de description est relativement clair. On donne successivement toutes les transitions du sytèmes en précisant pour chacune d'entre elles son origine, son étiquette et son extrémité.

On procède de même pour une boucle :

```
transition_system boucle < width = 0 >;
inactif |- active -> actif;
actif |- inactive -> inactif;
< initial = { inactif } >.
```

La dernière étape consiste à définir le produit synchronisé qui comporte 4 composantes :

```
synchronization_system carrefour < width = 4 ;
  list = ( feu, feu, boucle, boucle ); comp >;

(vert_to_orange. e. e. e);
(orange_to_rouge. e. e. e);
(rouge_to_vert. ?rouge. inactive. e);
(e. vert_to_orange. e. e);
(e. orange_to_rouge. e. e);
(?rouge. rouge_to_vert. e. inactive);
(e. e. active. e);
(e. e. e. active);
```

Spécification En *MEC*, la spécification se fait sous forme d'équations de points fixes. On peut par exemple définir l'ensemble des états qui satisfont $\mathsf{EF}(feu_1 = vert)$:

```
phi = !state[1] = "vert";
EF = phi \/ src(rtgt(EF));
```

La fonction $x \mapsto src(rtgt(x))$ étant tout simplement équivalente à EX.

Commentaire *MEC* est un langage très simple et facile à apprendre. Sa "puissance" intrinsèque est très grande puisqu'il permet de décrire tout système de transition. De plus, l'utilisateur reste bien maître de la façon dont le système est interprété.
Malheureusement, le travail de modélisation est souvent malaisé tant le langage est de bas niveau. C'est tout particulièrement la partie où il faut énumérer les n-uplets de transitions possibles qui est fastidieuse à écrire. De plus la modification d'un système déjà défini (comme par exemple l'ajout d'un composant) est parfois difficile.

1.4.2 SMV

SMV est un langage de plus haut niveau. Il permet de décrire des systèmes à variables partagées en définissant la relation de transition, c'est à dire en précisant une expression permettant de calculer l'état suivant du système en fonction de l'état actuel.

Exemple de modélisation Reprenons l'exemple de notre carrefour.
L'un des grands intérêts de *SMV* est qu'il permet des définitions modulaires. On peut donc écrire chaque composante de notre système de façon indépendante et tisser ensuite des liens. Commençons par définir une boucle :

```
MODULE boucle
VAR
  etat : {active,inactive};
ASSIGN
  init(etat) := inactive;
  next(etat) := case
                  etat = active : inactive;
                  etat = inactive : active;
                esac;
```

Un *module* de *SMV* est un ensemble de variable dont les valeurs évoluent. Les variables sont définies dans la clause `VAR`. Leur valeurs sont attribuées dans la clause `ASSIGN` : `init(etat) := inactive` signifie que la valeur initiale de la variable *etat* est *inactive* tandis `next(etat)` définit la valeur suivante de cette même variable en fonction de l'état actuel du système : c'est une définition par cas, si la valeur actuelle est *active*, la valeur suivante est également *active* ; sinon la valeur suivante est *inactive*.

Définissons ensuite un feu :

```
MODULE feu (etat_boucle, vert_possible)
VAR
  couleur : {rouge,orange,vert};
ASSIGN
  next(couleur) := case
                     couleur = orange : rouge;
                     couleur = rouge & vert_possible : vert;
                     couleur = vert & etat_boucle = active : orange;
                     1 : couleur
                   esac;
```

Notons dans cette définition deux nouveaux points par rapport à la notion suivante :
– Le module *feu* prend deux variables en argument. Ces deux variables seront précisées lorsque l'on fera appel au module. Elles permettront de faire le lien entre les différents modules.
– L'état initial de la variable *couleur* n'est pas défini. Elle est donc pour l'instant choisi de manière non-déterministe parmi les trois valeurs possibles.

Pour terminer notre modélisation, il nous faut écrire le module `main` qui est le module principal. Il va permettre d'établir les liens entre les différents composants.

```
MODULE main
VAR
  vp1 : boolean;
  vp2 : boolean;
  boucle1 : boucle;
  boucle2 : boucle;
  feu1 : feu(boucle2.etat; vp1);
  feu2 : feu(boucle1.etat; vp2);
ASSIGN
  init(feu1.couleur) = vert;
  init(feu2.couleur) = rouge;
  vp1 := feu2.couleur = rouge;
  vp2 := feu1.couleur = rouge;
```

Notre système est à présent défini. À chaque "pas" de l'ensemble, chacun des quatre modules implantés va devoir effectuer un pas. A chaque fois, toutes les affectations sont effectuées en parallèle. *SMV* vérifie par lui-même syntaxiquement qu'il n'y a pas de conflit d'écriture.

Exemple de spécification *SMV* permet de spécifier les propriétés du système étudié directement en logique temporelle CTL au sein d'une clause `SPEC`. *SMV* peut alors dire si la spécification est exacte ou non.
On peut ainsi par exemple lui demander si à chaque fois qu'une boucle va être activée, le feu correspondant va passer au vert :

```
SPEC
  AG(boucle1.etat = active -> (AF feu1.couleur = vert))
```

Commentaire *SMV* ne possède pas le même inconvénient que *MEC* : le langage de modélisation est assez évolué et souple, en particulier grâce à la description sous forme de modules faisant appel les uns aux autres. La description et surtout la synchronisation ne sont cependant pas toujours très aisées dans la mesure où il n'y a pas en *SMV* la notion d'état courant. Il est par conséquent souvent nécessaire de faire appel à une variable factice pour se substituer à cette notion, ce qui n'est pas toujours très naturel.

1.4.3 SPIN

SPIN est un vérificateur basé sur des processus asynchrones communicant par des canaux bornés. Le système étudié doit être décrit en *Promela*, le langage de modélisation de *SPIN* qui est un langage impératif qui ressemble un peu au *C*. Il permet de décrire le comportement de chacun des processus.
En *Promela*, il n'y a pas de différence entre une condition et une commande de telle sorte qu'une condition booléeene isolée peut être utilisée comme une commande. L'exécution de tout commande est conditionnée par son *exécutabilité* : une commande est exécutable ou bloquée. Par exemple, au lieu d'écrire une boucle d'attente telle que

```
while (a != b)
  skip
```

Il suffit d'écrire en *Promela* :

```
(a == b)
```

Une condition peut être exécutée seulement quand elle est vraie. Tant qu'elle est fausse, le processus correspondant est bloqué.
Deux commandes successives doivent être séparées par un point-virgule `;` ou indifféremment par une flèche `->` (qui est souvent utilisée pour mettre en évidence la causalité).

Exemple Ecrivons en *Promela* une modélisation de notre carrefour. Commençons par définir une boucle. Une boucle est en fait un processus qui se contente de faire passer – à n'importe quel moment – une variable à vrai :

```
proctype boucle(bool etat)
{
  do
  :: d_step{etat == 1 -> etat = 0}
  :: d_step{etat == 0 -> etat = 1}
  od
}
```

La construction `do ... od` permet de répéter la même instruction indéfiniment.
On peut maintenant passer à la définition d'un feu :

```
#define vert = 0;
#define orange = 1;
#define rouge = 2;
proctype feu(byte etat, etat_autre)
{
  bool etat_boucle = 0;
  run boucle(etat_boucle);

  do
  :: d_step{etat_boucle & etat_autre == rouge -> etat = vert;
                                                 etat_boucle = 0}
  :: d_step{etat == vert -> etat = orange}
  :: d_step{etat == orange -> etat = rouge}
  od
}
```

Les deux premières lignes de la définition permettent de créer une boucle à chaque fois que l'on instancie un feu. On donne ensuite le comportement infini du feu à l'aide d'une structure `do ... od`. Chacune des alternatives est atomique (elle est exécutée en un seul pas), c'est le rôle du `d_step`. Dans la pratique, la première commande de chaque alternative (à gauche de `->`) est la condition et la suite (à droite de `->`) donne les actions à effectuer.
Pour terminer, il suffit de d'écrire une procédure `init` pour lancer le système :

```
init
{
  byte etat1, etat2 = rouge;
  run feu(etat1, etat2);
  run feu(etat2, etat1);
}
```

Commentaire *Promela* est un langage très complet. Nous n'avons entrevu ici qu'une toute petite partie de ses possibilités. Bien que rebutant lors d'une première approche (car assez singulier), il s'avère par la suite assez agréable à manipuler et adapté aux cas concrets. Le danger est cependant que l'utilisateur perd parfois le contrôle de la façon dont la définition est interprétée.
Le vérificateur *SPIN* lui-même est performant. Il permet soit de simuler une exécution du système décrit (ce qui est utile pour s'approprier son fonctionnement) soit de vérifier des propriétés de manière exhaustive. *SPIN* peut générer du code *C* à partir du code *Promela*.

Chapitre 2

Modélisation : Automates synchronisés par variables partagées

2.1 Motivation

Un peu d'utilisation des trois langages de spécification décrits précédemment, *MEC*, *SMV* et *SPIN* permet de formuler les remarques suivantes :

– Ces trois langanges ne sont pas toujours très adaptés à la définition de systèmes de taille moyenne : *MEC* est de trop bas niveau, *SPIN* ne permet pas toujours à l'utilisateur de contrôler exactement le modèle qu'il construit et on regrette l'éloignement de *SMV* par rapport à la structure classique d'automate.
– Suivant les questions que l'on se pose, chaque moteur de vérification possède ses intérêts et ses défauts. Il est difficile de faire un choix général. La description d'un système dans les trois langages, outre le fait qu'elle est fastidieuse, n'est pas souvent satisfaisante car il est difficile de produire trois modèles strictement équivalents tant les spécifités de chaque produit amènent naturellement à effectuer des modifications.
– Il serait intéressant d'établir des passerelles entre langages afin de pouvoir comparer leur comportements sur des spécifications équivalentes.

L'intérêt d'une plateforme commune apparaît maintenant évident. Il est illustré par les quelques exemples que nous avons donnés à la fin de la partie 1. Deux approches différentes se présentent alors :

1. Ecrire six convertisseurs permettant de traduire une modélisation écrite dans l'un des trois langages dans n'importe lequel des deux autres.

2. Définir un langage à partir duquel on peut produire une modélisation pour chacun des trois langages.

Chaque approche possède ses avantages et ses inconvénients. La première permettrait de réutiliser les modélisations déjà effectuées et ne nécessiterait pas l'apprentissage d'un nouveau langage. Elle se heurterait cependant à plusieurs difficultés. Les différences de niveau entre les langages ne permettraient pas de générer du code très "lisible" par l'utilisateur. Ce dernier ne percevrait donc pas toujours comment son modèle a été traduit et ne pourrait interpréter facilement les réponses données par le vérificateur. Un deuxième problème, plus technique, se pose : les trois langages sont susceptibles d'évoluer et leur spécification propre n'est pas toujours bien précisées par leurs auteurs. Il serait donc probablement nécessaire de se limiter à un fragment de chacun (sauf peut-être pour *MEC*).

La seconde approche présente l'avantage que le langage d'entrée n'étant pas défini, nous avons toute latitude pour le choisir adapté ; d'une part à l'utilisateur et d'autre part à la production de code pour *MEC*, *SMV* et *SPIN* (sans toutefois hypothéquer de nouvelles passerelles). Le revers de la médaille est qu'il sera nécessaire à l'utilisateur d'en faire l'apprentissage et d'éventuellement réécrire ses programmes.

C'est cette deuxième approche qui a été retenue.

2.2 Introduction du langage

2.2.1 Un premier exemple

Avant de donner une définition précise de la syntaxe du langage, étudions quelques exemples simples pour illustrer les concepts de base. Considérons le petit programme suivant :

```
SYSTEME main
VAR
  request : bool = ff
INCL
TRANS
  ready |- request = tt -- execute -- request = ff -> buzy;
  ready |- -- -- request = tt -> ready;
  buzy |- -- -- request = tt -> buzy;
  buzy |- -- -- -> ready
INITIAL
  ready

SPEC
  AG(request = tt -> AF curr_state = buzy)
```

Le fichier d'entrée décrit à la fois le modèle et la spécification. Le modèle est un automate à variables. Dans notre exemple, l'automate décrit peut prendre deux états : *ready* (l'état initial) et *buzy*. Il comporte une variable booléenne, *request* dont la valeur initiale est `ff` (faux).
Pour décrire un automate, on donne la liste des variables (c'est la partie `VAR` de la définition), la liste des transitions (c'est la clause `TRANS`) et la liste des états initiaux (`INITIAL`).

Variables La définition d'une variable a la forme suivante :

```
identificateur : type = valeur_initial
```

On peut utiliser trois sortes de types de base :
- Booléeen : `bool`,
- Entiers : `x .. y` pour l'ensemble $\{x, \dots y\}$,
- Littéraux, c'est à dire un ensemble fini de constantes. Par convention, un nom de constante commence toujours par une apostrophe `'`. Pour désigner le type, on utilise la notation ensembliste usuelle, par exemple : {`'rouge ; 'vert ; 'bleu`}.

Transitions L'écriture d'une transition comporte cinq clauses :

```
etat_origine |- condition -- etiquette -- actions -> etat_but
```

Les trois clauses `etat_origine`, `etat_but` et `etiquette` sont des identificateurs (qui peuvent être toute suite de lettres et chiffres commençant par une lettre). La clause `condition` est une expression booléeene en les variables. Elle exprime quand la transition peut être effectuée. La clause `actions` décrit ce que l'on doit faire lorsqu'on la transition est passée. Il s'agit principalement d'une liste d'affectations de la forme `x1 = nouvelle_valeur ; x2 = ...` qui permettent de modifier le contenu des variables : on calcule tous les membres de droite (`nouvelle_valeur`) puis on affecte les résultats aux variables respectives (`x1`, ...).

L'exécution d'un tel système se déroule de la manière suivante : on part de l'état initial défini par les clauses `VAR` et `INITIAL`. A chaque pas, on choisit une transition dont la condition est vraie (si plusieurs transitions sont possibles, le choix est non déterministe, si aucune n'est possible, le système est bloqué). On passe cette transition en effectuant toutes les actions *en parallèle*.

2.2.2 Modules

L'exemple suivant illustre la définitions de systèmes réutilisables. Notez que le système nommée `main` a un rôle spécial, de la même façon que la fonction `main` d'un programme C. L'ordre de définition des différents systèmes doit être tel que tout système ne fasse référence qu'à un système préalablement défini.

```
SYSTEME feu (controle : bool; boucle : bool)
VAR
INCL
TRANS
  vert |- -- -- -> orange;
  orange |- -- -- controle = ff -> rouge;
  rouge |- controle = ff & boucle = tt --
           -- controle = tt; boucle = ff -> vert
INITIAL
  rouge

SYSTEME main ()
VAR
  controle : bool = ff;
  boucle1 : bool = ff;
  boucle2 : bool = ff
INCL
  process feu1 = feu(controle; boucle1);
  process feu2 = feu(controle; boucle2)
TRANS
  i |- -- -- boucle1 = tt -> i;
  i |- -- -- boucle2 = tt -> i
INITIAL
  i
```

Chaque système instancié peut lancer l'exécution d'autres systèmes par la clause `INCL` en leur passant éventuellement des variables comme paramètres. Les systèmes sont ainsi liés les uns aux autres par une structure d'arbre.
Le mot clef `process` dans l'instanciation d'un système signifie que celui-ci s'exécute de manière indépendante et asynchrone. En l'absence de ce mot-clef, le processus est synchrone avec son père.

2.2.3 Files

Le langage permet d'utiliser des files d'attentes finies pour communiquer entre les processus. Une file d'attente doit contenir des objets de même type, par exemple des entiers ou des littéraux. Une file est déclarée comme toute variable dans la clause `VAR`, par exemple

```
ma_file : [5] {'a; 'b; 'c} = []
```

crée une file de taille 5. Le symbole `[]` signifie que la file est initialement vide – ce qui est toujours le cas.
Dans les conditions, on peut faire des tests sur le sommet d'une file. Par exemple `ma_file? 'a` permet de tester si le premier élément à sortir d'une file est `'a` (par convention, faux si la file est vide). On peut aussi faire intervenir le nombre d'éléments de la file, `len(ma_file)`.
Dans la liste des actions d'une transition, on peut ajouter pour chaque file, parralèlement à des affectations de variables "normales", un enfilement ou un défilement : `ma_file << expr` enfile un objet tandis que `ma_file >> variable` défile un objet et le place dans la variable précisée. Une transition comportant une action `push` ou `pop` ne peut être passée si la file correspondante est respectivement pleine ou vide.
L'exemple suivant modélise une imprimante reliée à plusieurs utilisateurs :

```
SYSTEME utilisateur(id : 1 .. 2; file : [5] 1 .. 2)
TRANS
  repos |- -- imprime -- file << id -> repos
INITIAL
  repos

SYSTEME main()
VAR
  id1 : 1 .. 2 = 1;
```

```
    id2 : 1 .. 2 = 2;
    null : 1 .. 2;
    file : [5] 1 .. 2
  INCL
    u1 = utilisateur(id1; file);
    u2 = utilisateur(id2; file)
  TRANS
    repos |- file ? 1 -- -- file >> null -> imprime1;
    repos |- file ? 2 -- -- file >> null -> imprime2;
    imprime1 |- -- -- -> repos;
    imprime2 |- -- -- -> repos
  INITIAL
    repos
```

2.3 Syntaxe et référence du langage

2.3.1 Convention lexicales

Un `atome` de la syntaxe décrite ci-après peut être n'importe quelle suite de caractères de l'ensemble {`A-Z,a-z,0-9`}, commençant par une lettre. Tous les caractères d'un nom sont significatifs et la casse l'est aussi. Les espaces, tabulations et retours à la lignes sont considérés comme des caractères d'espacement. Un `nombre` est n'importe suite de chiffres. Les autres éléments lexicaux reconnus par l'analyseur sont écrits entre guillemets dans les expressions qui suivent.

2.3.2 Expressions

Les expressions sont construites à partir des variables, constantes et d'une collection d'opérateurs. La syntaxe et la suivante :

`constante : :`	
`"'" atome`	*Une constante*
`| nombre`	*Une constante numérique*
`| "tt" | "ff"`	*Une constante booléene*

`expr : :`	
`constante`	*Une constante symbolique*
`| atome`	*Un identificateur de variable*
`| ( expr )`	*Expression parenthésée*
`| expr1 "+" expr2`	*Addition entière*
`| expr1 "-" expr2`	*Différence entière*
`| expr1 "*" expr2`	*Multiplication entière*
`| "!" expr1`	*Négation logique*
`| expr1 "&" expr2`	*Et logique*
`| expr1 "|" expr2`	*Ou logique*
`| expr1 "=" expr2`	*Egalité*
`| expr1 "<" expr2`	*Inférieur à*
`| expr1 ">" expr2`	*Supérieur à*
`| "len" "(" atome ")"`	*Longueur d'une file*
`| atome "?" constante`	*Test sur le sommet d'une file*

La priorité des opérateurs est la suivante (les opérateurs de même priorité sont associatifs à gauche et les expressions peuvent être parenthésées) :

```
*
+,-
=,<,>
!
&
|
```

2.3.3 Définitions

Une définition est la structure qui permet de spécifier la constitution d'un système. Un programme est ainsi une suite de définitions. L'écriture d'une définition en elle-même ne dit pas comment elle sera instanciée. Au lancement du programme, seule la définition `main` (dont la présence, en dernier, est impérative) est instanciée. Les autres définitions le sont ensuite de manière arborescente grâce aux clauses `INCL`. Chaque définition peut ainsi être instanciée plusieurs fois, ce qui permet de réutiliser des fragments de code.
La syntaxe d'une définition est la suivante :

`définition : :`	
`"SYSTEME" atome "(" par1 ; ... ; parn ")"`	*Paramètres.*
`"VAR" clause_var`	*Les variables locales*
`"INCL" clause_incl`	*Les systèmes inclus*
`"TRANS" clause_trans`	*Les transitions*
`"INITIAL" clause_initial`	*Les états initiaux*

Variables

Types Trois types "de base" sont manipulés : entiers, booléens et littéraux. Une définition de type de base à la forme suivante :

`type_base : :`	
`"bool"`	*Booléen*
`| nombre ".." nombre`	*Intervalle d'entiers*
`| {'atome1 ; 'atome2 ; ... atomen}`	*Littéraux*

Un type général peut être soit un type de base, soit une file sur un des types de base. Une file est d'une longueur bornée spécifiée dans la définition du type :

`type : :`	
`type_base`	*Un type de base*
`| "[" nombre "]" type_base`	*Une file*

Clause VAR On peut maintenant donner la syntaxe de la clause `VAR` d'une définition :

```
clause_var : :
    atome1 " :" type1 "=" constante1 " ;"
    atome2 " :" type2 "=" constante2 " ;"
    ...
    atomen " :" typen "=" constanten " ;"
```

Inclusions

La clause `INCL` d'une définition permet de spécifier la liste des systèmes qui seront créés simultanément à chaque instanciation de la définition. Les systèmes ainsi lancés doivent avoir été définis préalablement.
Pour chaque inclusion, il faut indiquer successivement le nom à donner au système créé, l'identificateur de la définition utilisée et la liste des variables partagées passées en argument.

```
clause_incl : :
    ident1 " :" systeme1 "(" var1 ; var2 ... ")" " ;"
    ident2 " :" systeme2 "(" var1 ; var2 ... ")" " ;"
    ...
    identn " :" systemen "(" var1 ; var2 ... ")" " ;"
```

En en-tête de chaque inclusion, il est possible d'ajouter le mot-clef `process` pour indiquer que le système doit être lancé comme un processus indépendant asynchrone. Dans le cas contraire, le sous-système est synchrone, ce qui signifie qu'il exécute une transition à chaque fois que le système appelant en passe une.

Transitions

Actions Les actions permettent de modifier la valeur des variables du système. Il en existe trois sortes :

```
action : :
    variable "=" expr      Affectation d'une variable simple
    file "<<" expr         Ajout d'un élément dans une file
    file ">>" variable     Lecture d'un élément d'une file
```

Transitions La syntaxe d'une transition est la suivante :

```
transition : :
    atome                                   état initial
      |- expr                               condition
      -- atome                              étiquette
      -- action1 ; action2 ; ... actionn    actions
      -> atome                              état final
```

La condition ou la liste d'actions peuvent éventuellement être vides. Les actions sont effectuées en parallèle : les expressions affectées (membres de droite) sont en effet toutes calculées en fonction des anciennes valeurs des variables.

Clause TRANS La clause `TRANS` permet de donner la liste des transitions du système défini. Elle suit la syntaxe suivante :

```
clause_trans : :
    transition1 " ;"
    transition2 " ;"
    ...
    transitionn " ;"
```

Clause INITIAL

La clause `INITIAL` permet de donner la liste des états initiaux de l'automate. Elle suit la syntaxe suivante :

```
clause_initial : :
    atome1 ";"
    atome2 ";"
    ...
    atomen ";"
```

2.3.4 Alias de types

Pour éviter d'avoir à recopier plusieurs fois une définition d'un type (simple) relativement longue (en particulier pour les littéraux), il est possible de définir des alias de types. L'alias peut ensuite être utilisé systématiquement à la place de la forme normale du type.

```
alias : :
    TYPE atome "=" type_simple
```

2.3.5 Spécification

Logique CTL

Avant de donner la syntaxe des expressions de logique CTL, définissons la notion de chaîne. Une chaîne permet de désigner n'importe quel automate ou variable du système étudié. Par exemple, la chaîne `.alpha.beta` désigne soit la variable *beta* du systeme *alpha* soit l'état du sosu-système *beta* du système *alpha* (pour le savoir, il suffit de regarder la définition du système *alpha*). Par convention, la chaîne `.` désigne le module *main*. C'est pour cela que toute chaîne commence par un point.
La syntaxe d'une chaîne est la suivante :

```
chaîne : :
    "." atome1 "." atome2 "." ... "." atomen
```

Pour spécifier les propriétés à vérifier, on utilise la logique temporelle CTL. Les atomes de cette logique sont les chaînes des automates et des variables. Elle permet donc d'exprimer des propriétés relatives aux états courants des automates du système et au contenu des variables.

```
ctlexpr : :
    chaîne
    | EX ctlexpr
    | AX ctlexpr
    | EF ctlexpr
    | AF ctlexpr
    | EG ctlexpr
    | AG ctlexpr
    | E ctlexpr U ctlexpr
    | A ctlexpr U ctlexpr
    | E ctlexpr W ctlexpr
    | A ctlexpr W ctlexpr
    | ctlexpr | ctlexpr
    | ctlexpr & ctlexpr
    | ctlexpr = ctlexpr
```

Clauses SPEC

Pour chaque formule CTL, il convient d'utiliser une clause SPEC. Chacune de ces clauses permet ainsi d'indiquer une formule qui sera vérifiée.

`clause_spec : :`	
`SPEC atome "="`	*L'identificateur de la formule*
`ctlexpr`	*La formule CTL*

Chaque formule porte un nom qui permet de la distinguer.

2.4 Utilisation

2.4.1 Compilation

Après avoir écrit un fichier d'entrée, enregistrez-le en lui donnant un nom dont l'extension est `.avr`, par exemple `foo.avr`. Pour générer le code pour les trois langages, il vous suffit d'exécuter la commande suivante :

```
autovar foo.avr
```

Le compilateur lit votre source et écrit trois fichiers, un par langage, `foo.mec`, `foo.smv` et `foo.spin`.

2.4.2 Messages d'erreurs

Si votre code source comporte une erreur de syntaxe, le compilateur va vous le signaler en indiquant le numéro de la ligne ainsi qu'un court message d'explication.
Il se peut également que votre système comporte un conflit d'écriture, c'est à dire que lors d'une même transition, deux contenus différents soit affectés à la même variable. Cela vous sera indiqué lors de la traduction en *MEC*.
Enfin, les erreurs "système" (accès aux fichiers...) sont directement gérées par le rattrapage d'exceptions fourni par *Objective Caml*.

2.5 Description technique

Je vais essayer de décrire succintement dans cette partie la manière dont le compilateur produit du code pour les trois langages, *MEC*, *SMV*, et *SPIN*.

2.5.1 Vérifications préalables

Après la lecture du fichier source et la production d'un arbre syntaxique, quelques vérifications sont opérées. Il s'agit de vérifications syntaxiques de base :
- Le compilateur vérifie que toutes les variables utilisées sont bien définies.
- Le compilateur vérifie que tous les systèmes utilisés au sein d'autre systèmes sont bien préalablement définis. Simultanément, il contrôle que les appels comportent les bons nombres d'arguments.
- Le compilateur effectue une vérification de la correction du type des expressions manipulées et des arguments passés.

Le principal rôle de ces vérifications est d'éviter la production de code qui n'aurait pas de sens dans l'un ou l'autre des langages.

2.5.2 Production de code MEC

La production de code *MEC* est une véritable "compilation" tant le langage de destination est de bas niveau. Elle se déroule en deux parties bien distinctes :
- l'écriture des définitions des systèmes de transitions,
- l'écriture du produit de synchronisation.

Écriture des définitions des systèmes de transitions C'est la partie la plus aisée. De chaque "système" on déduit aisément un *transition_system* en ne prenant en compte que les origines, étiquettes et extrémités de chaque transition et en "oubliant" la condition et les actions. Il faut toutefois prendre garde pour la suite à ce que deux transitions ne portent la même étiquette que si elles ont même extrémité. Pour cela, on code dans chaque étiquette du résultat à la fois l'étiquette de la source mais aussi l'origine et l'extrémité (par concaténation).
Cette retranscription permet d'éviter les doublons : à chaque "système" d'origine corrrespond un et un seul *transition_system* dans le code *MEC*, quel que soit le nombre d'utilisation qui en soit fait ultérieurement. Le *transition_system* porte le nom du "système" originel.

MEC ne permettant pas de manipuler de variables, il faut coder chaque variable par un automate. Il suffit pour cela d'écrire un automate dont les états sont les valeurs possibles de la variable et d'envisager toutes les transitions possibles sous la forme `x |- x_to_y -> y`. Par exemple, pour une variable à valeur dans $\{'a,'b,'c\}$, on écrirait l'automate :

```
a |- a_to_c -> c,
  |- a_to_b -> b,
  |- a_to_a -> a;
b |- b_to_c -> c,
  |- b_to_b -> b,
  |- b_to_a -> a;
c |- c_to_c -> c,
  |- c_to_b -> b,
  |- c_to_a -> a;
```

Les transitions de la forme `x |- x_to_x -> x` permettront par la suite de vérifier le contenu d'une variable.

Écriture du produit de synchronisation Pour écrire le produit de synchronisation, la technique utilisée consiste à envisager de manière exhaustive les états que peut prendre le système. On considère successivement tous les environnements de valeurs possibles pour les variables. Pour chacun d'entre eux, on teste pour chaque processus si un passage de transition est possible. Si c'est le cas, on écrit le n-uplet de transition correspondant, composé tout d'abord des transitions effectuées par les variables sous la forme `ancienneValeur_to_nouvelleValeur` puis les transitions des systèmes. Les automates des processus non mis en jeu effectuent une ϵ-transition.

2.5.3 Production de code SMV

La production de code *SMV* peut se faire de manière relativement fidèle étant donné que ce langage utilise une structure analogue pour les relations entre les différents systèmes, sous la forme d'un arbre.

Notion d'états En *SMV*, il n'y a pas de notion d'état courant pour un sytème (ou module)... Pour simuler cette notion, il suffit d'ajouter à chaque définition une variable `curr_state` dont la valeur est une constante décrivant l'état courant du système.

Files Le langage *SMV* ne fournit pas de structure permettant de définir directement des files. Pour pallier à ce manque, lors de la compilation chaque file est simulée par un module distinct. Par exemple, pour une file de type `[2] {'a ; 'b ; 'c}`, le code suivant est généré :

```
MODULE systeme_file
VAR
  data : array 0 .. 1 of {a,b,c};
  len : {0,1,2};
  top : {a,b,c};
  data_in : {a,b,c};
  action : {defiler,enfiler,rien};

ASSIGN
  init(data[0]) := a;
  init(data[1]) := a;
```

```
init(data_in) := a;
init(len) := 0;
init(action) := rien;
next(len) := case
               action = enfiler & len < 2 : len + 1;
               action = defiler & len > 0 : len - 1;
               1 : len;
             esac;

top := data[0];

next(data[0]) := case
                   action = defiler : data[1];
                   action = enfiler & len = 0 : data_in;
                   1 : data[0];
                 esac;
next(data[1]) := case
                   action = enfiler & len = 1 : data_in;
                   1 : data[1];
                 esac;
```

Les variables `data_in` et `action` permettent de "commander" la file tandis que `len` et `top` permettent de la lire.

Transitions La fonction de transition de l'automate décrit par un module est spécifiée non pas à l'aide d'une clause `ASSIGN` mais d'une clause `TRANS`. Cette clause est une expression booléene qui permet de spécifier l'ensemble des transitions possibles.
L'expression que nous écrivons est une disjonction dont chaque terme est une conjonction représentant une transition du système initial.

2.5.4 Production de code SPIN

Regroupement par processus A chaque processus indépendant du système défini doit correspondre en *SPIN* un et un seul processus *SPIN*. Or, dans, notre langage, un processus synchrone peut comprendre plusieurs automates synchronisés. Il est donc nécessaire d'effectuer préalablement leur produit formel avant de produire le code *SPIN*.
Le produit de plusieurs automates est réalisé en prenant la réunion de toutes les variables puis en considérant comme états et comme transitions les n-uplets dont les composantes proviennent des automates sources.

Description de l'automate Comme en *SMV*, il faut définir une variable `curr_state` dans chaque processus. On peut alors facilement définir la relation de transition par une clause `do ... od` dont chaque ligne est une instruction atomique représentant une transition de la forme `d_step{condition -> action}`

Variables Les variables entières et booléeenes sont retranscrites en utilisant respectivement les types *short* et *bool*. Pour les types littéraux, on utilise également le type *short* (chaque littéral étant codé par un entier). Afin de conserver la lisibilité du code, des clause `#define` sont insérées en préambule.
Les files sont directement codées à l'aide des canaux de communication *SPIN*.

Chapitre **3**

Spécification : Logique temporelle CTL

3.1 Motivation

La spécification en *MEC* souffre de deux principaux inconvénients :
- Le formalisme utilisé, celui des équations de points fixe, est certe très puissant, mais peu lisible. Il est donc difficile d'en comprendre le "sens" en relisant les égalités. De plus, il est souvent nécessaire de prouver la validité des points fixes calculés pour que les résultats donnés par *MEC* puissent être utilisés. L'utilisation d'un langage comme CTL permet de pallier à cet inconvénient.
- La syntaxe de *MEC* est peu agréable. Elle ne permet en particulier pas de faire des appels imbriqués de fonctions ce qui nécessite de systématiquement décomposer tous les calculs en étapes élémentaires.

Nous allons procéder en deux étapes. Dans un premier temps, nous définirons un certain nombre de fonction *MEC* pour permettre d'utiliser la logique temporelle CTL directement dans ce langage.
Dans un deuxième temps, nous écrirons un petit programme permettant de traduire facilement des spécifications données avec la syntaxe CTL usuelle en équations *MEC*.

3.2 Points fixes de fonctions monotones

3.2.1 Monotonie, additivité, continuité

Monotonie Soit $(\Omega, \sqsubseteq)$ un treillis dénombrable. Considérons une fonction f de Ω dans Ω. On dit que f est monotone si et seulement si :

$$\forall\, x, y \in \Omega,\ x \sqsubseteq y \ \Rightarrow\ f(x) \sqsubseteq f(y)$$

Additivité On dit de plus que f est additive dans le cas où :

$$\forall\, x, y \in \Omega,\ f(x \sqcup y) = f(x) \sqcup f(y)$$

L'additivité est une notion strictement plus forte que la monotonie (L'implication est évidente. Pour montrer qu'une fonction monotone n'est pas nécessairement additive, il suffit de considérer par exemple pour $\Omega = \mathcal{P}(\mathbb{N})$, $x \mapsto \{1, \ldots card(x)\}$).

Continuité Une fonction f est dite $\sqcup$-continue (respectivement $\sqcap$-continue) si et seulement si pour toute suite (finie ou infinie) $(x_i)_i$ croissante (respectivement décroissante) :

$$f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i) \qquad \text{(respectivement, } f(\bigsqcap_i x_i) = \bigsqcap_i f(x_i))$$

On remarquera que toute fonction $\sqcup$-continue ou $\sqcap$-continue est monotone et que si f est monotone on a toujours :

$$\bigsqcup_i f(x_i) \sqsubseteq f(\bigsqcup_i x_i) \text{ et } f(\bigsqcap_i x_i) \sqsubseteq \bigsqcap_i f(x_i))$$

3.2.2 Théorême de Tarski

Le théorême de Tarski permet de résoudre dans un certain nombre de cas des équations du type $x = f(x)$ sur un treillis.

Théorême 1 (Tarski) *Soit f une application monotone de Ω dans Ω. L'équation $x = f(x)$ admet une plus petite et une plus grande solution qui sont respectivement égales à :*

$$\bigsqcap\{x \in \Omega \ / \ f(x) \sqsubseteq x\}$$

et

$$\bigsqcup\{x \in \Omega \ / \ x \sqsubseteq f(x)\}$$

Démonstration Posons $X = \{x \ / \ x \sqsubseteq f(x)\}$, $\hat{x} = \bigsqcup_{x \in X} x$ (X n'est pas vide puisque $\perp \sqsubseteq f(\perp)$) et montrons que $\hat{x}$ est la plus grande solution de l'équation $x = f(x)$.
Toute solution de cette équation est un élément de X et est donc plus grande que $\hat{x}$. Il nous suffit donc de montrer que $f(\hat{x}) = \hat{x}$.
Soit $x \in X$. On a $x \sqsubseteq \hat{x}$ et donc par monotonie de f, $f(x) \sqsubseteq f(\hat{x})$. Or $x \sqsubseteq f(x)$ donc $x \sqsubseteq f(\hat{x})$. Ceci étant vrai quel que soit s, on en déduit que $\bigsqcup_{x \in X} x \sqsubseteq f(\hat{x})$ c'est-à-dire $\hat{x} \sqsubseteq f(\hat{x})$.
Comme f est monotone, on en déduit que $f(\hat{x}) \sqsubseteq f(f(\hat{x}))$ donc $f(\hat{x}) \in X$. Il en résulte que $f(\hat{x}) \sqsubseteq \hat{x}$.
On procède de la même façon pour la plus petite solution. $\square$

Théorême 2 *Si f est $\sqcup$-continue, le plus petit point fixe de f est $\bigsqcup_{i \geq 0} f^i(\perp)$.*
Si f est $\sqcap$-continue, le plus grand point fixe de f est $\bigsqcap_{i \geq 0} f^i(\top)$.

Démonstration Supposons f $\sqcup$-continue. f étant monotone, la suite $(f^i(\perp))_i$ est croissante. Puisque f est $\sqcup$-continue, $f(\bigsqcup_{i \geq 0} f^i(\perp))$ est égal à $\bigsqcup_{i \geq 0} f^{i+1}(\perp)$, qui est aussi égal à $\bigsqcup_{i \geq 0} f^i(\perp)$, et est donc un point fixe de f.
C'est aussi le plus petit point fixe car si x est un point fixe de f alors $\perp = f^0(\perp) \sqsubseteq x$, et si $f^i(\perp) \sqsubseteq x$ alors $f^{i+1}(\perp) \sqsubset x$. $\square$

Remarque Si Ω est un ensemble fini, il existe dans les deux cas un entier n tel que

$$\bigsqcup_{i \geq 0} f^i(\perp) = \bigsqcup_{i=0}^{n} f^i(\perp) = f^n(\perp)$$

ou

$$\bigsqcap_{i \geq 0} f^i(\top) = \bigsqcap_{i=0}^{n} f^i(\top) = f^n(\top).$$

Le point fixe est alors facilement calculable.

3.2.3 Principe de dualité

Théorême 3 (Dualité) *Considérons le treillis des parties d'un ensemble. Le complémentaire du plus petit point fixe de la fonction $f : x \mapsto f(x)$ est le plus grand point fixe de la fonction duale $x \mapsto \neg f(\neg x)$.*

3.3 Sémantique des chemins maximaux

Dans cette partie on considère un automate $\mathcal{A} = (Q, I, T)$ (Q est l'ensemble des états et T est l'ensemble des transitions).
Pour tout état $q \in Q$, on note $C(q)$ l'ensemble des chemins (finis ou infinis) d'origine q, $C_m(q)$ l'ensemble des chemins maximaux d'origine q. Pour tout chemin σ, on note $|\sigma|$ son nombre d'états et pour $0 \leq i < |\sigma|$, $\sigma(i)$ son i-ème état.
Enfin, afin de simplifier les notations, on s'autorise à confondre une expression logique CTL avec l'ensemble des états qui la satisfont.

3.3.1 État suivant

EX

Définition ensembliste

$$\begin{aligned}
[\mathsf{EX}\phi] &= \{q \in Q \ / \ q \models \mathsf{EX}\phi\} \\
&= \{q \in Q \ / \ \exists\, \sigma \in C_m(q),\ |\sigma| > 1 \ \wedge\ \sigma(1) \models \phi\} \\
&= \{q \in Q \ / \ \exists\, \sigma \in C(q),\ |\sigma| > 1 \ \wedge\ \sigma(1) \models \phi\} \\
&= \{q \in Q \ / \ \exists\, q' \models \phi,\ q \rightarrow q' \in T\}
\end{aligned}$$

Implantation

```
function EX(phi:state) return r:state;
begin
  r = src(rtgt(phi))
end.
```

AX

Définition ensembliste

$$[\mathsf{AX}\phi] = \{q \in Q \ / \ \forall\, \sigma \in C_m(q),\ |\sigma| > 1 \ \wedge\ \sigma(1) \models \phi\}$$

Relation avec EX

$$\begin{aligned}
q \models \mathsf{AX}\phi \ &\Leftrightarrow\ \forall\, \sigma \in C_m(q),\ |\sigma| > 1 \ \wedge\ \sigma(1) \models \phi \\
&\Leftrightarrow\ q \text{ a un successeur } \wedge\ \forall \sigma \in C(q),\ |\sigma| > 1 \ \Rightarrow\ \sigma(1) \models \phi \\
&\Leftrightarrow\ q \models \mathsf{EX}tt \ \wedge\ \neg(\exists\, \sigma \in C(q),\ |\sigma| > 1 \ \wedge\ \sigma(1) \models \neg\phi) \\
&\Leftrightarrow\ q \models \mathsf{EX}tt \ \wedge\ q \models \neg\mathsf{EX}(\neg\phi)
\end{aligned}$$

On en déduit que $[\mathsf{AX}\phi] \ = \ [\mathsf{EX}tt] \cap [\neg\mathsf{EX}(\neg\phi)]$.

Implantation

```
function AX(phi:state) return r:state;
begin
  r = src(rtgt(*)) - src(rtgt(* - phi))
end.
```

3.3.2 Until

EU

Définition ensembliste

$$[\mathsf{E}\phi_1\mathsf{U}\phi_2] = \\ \{q \in Q \ / \ \exists\, \sigma \in C_m(q),\ \exists\, i < |\sigma|,\ \sigma(i) \models \phi_2 \ \wedge\ (\forall\, j < i, \sigma(j) \models \phi_1)\}$$

Définition par point fixe Soit f la fonction $\psi \mapsto \phi_2 \cup ([\mathsf{EX}\psi] \cap \phi_1)$. EX étant monotone, on en déduit que f est monotone. Définissons également pour $k \in \mathbb{N}$ l'ensemble $[\mathsf{E}\phi_1 \mathsf{U}_k \phi_2]$ suivant :

$$[\mathsf{E}\phi_1 \mathsf{U}_k \phi_2] = \{q \in Q \ / \ \exists\, \sigma \in C_m(q), \exists\, i < k,\ \sigma(i) \vDash \phi_2 \ \wedge\ (\forall\, j < i, \sigma(j) \vDash \phi_1)\}$$

Le "ou" étant distributif par rapport à la quantification existentielle, on a l'égalité $[\mathsf{E}\phi_1 \mathsf{U} \phi_2] = \bigcup_{k \geq 0} [\mathsf{E}\phi_1 \mathsf{U}_k \phi_2]$.
On vérifie par récurrence que $f^k(\emptyset) = [\mathsf{E}\phi_1 \mathsf{U}_k \phi_2]$:

$$\begin{array}{lll} \text{Base :} & f^0(\emptyset) & = \emptyset \\ \text{Induction :} & f^{k+1}(\emptyset) & = \phi_2 \cup ([\mathsf{EX} f^k(\emptyset)] \cap \phi_1) \\ & & = \phi_2 \cup ([\mathsf{EXE}\phi_1 \mathsf{U}_k \phi_2] \cap \phi_1) \\ & & = \ [\mathsf{E}\phi_1 \mathsf{U}_{k+1} \phi_2] \end{array}$$

La fonction f étant continue, on en déduit que l'ensemble $[\mathsf{E}\phi_1 \mathsf{U} \phi_2]$ est le plus petit point fixe de la fonction f.

Implantation

```
function EU(phi1:state; phi2:state) return r:state;
begin
  r = phi2 \/ (src(rtgt(r)) /\ phi1)
end.
```

AU

Définition ensembliste

$$[\mathsf{A}\phi_1 \mathsf{U} \phi_2] = \{q \in Q \ / \ \forall\, \sigma \in C_m(q),\ \exists\, i < |\sigma|,\ \sigma(i) \vDash \phi_2 \ \wedge\ (\forall\, j < i, \sigma(j) \vDash \phi_1)\}$$

Définition par point fixe Soit g la fonction $\psi \mapsto \phi_2 \cup ([\mathsf{AX}\psi] \cap \phi_1)$. Définissons également pour $k \in \mathbb{N}$ l'ensemble $[\mathsf{A}\phi_1 \mathsf{U}_k \phi_2]$ suivant :

$$[\mathsf{A}\phi_1 \mathsf{U}_k \phi_2] = \{q \in Q \ / \ \forall\, \sigma \in C_m(q),\ \exists\, i < k,\ \sigma(i) \vDash \phi_2 \ \wedge\ (\forall\, j < i, \sigma(j) \vDash \phi_1)\}$$

Nous allons montrer que $\mathsf{A}\phi_1 \mathsf{U} \phi_2$ est le plus petit point fixe de g.
Vérifions tout d'abord que $[\mathsf{A}\phi_1 \mathsf{U} \phi_2] = \bigcup_{k \geq 0} [\mathsf{A}\phi_1 \mathsf{U}_k \phi_2]$. On a tout d'abord pour tout entier k, $[\mathsf{A}\phi_1 \mathsf{U}_k \phi_2] \subseteq [\mathsf{A}\phi_1 \mathsf{U} \phi_2]$ donc $\bigcup_{k \geq 0} [\mathsf{A}\phi_1 \mathsf{U}_k \phi_2] \subseteq [\mathsf{A}\phi_1 \mathsf{U} \phi_2]$. Inversement, soit $q \in [\mathsf{A}\phi_1 \mathsf{U} \phi_2]$. Posons k_0 le nombre d'états de l'automate et supposons que $q \notin [\mathsf{A}\phi_1 \mathsf{U}_{k_0} \phi_2]$. Il existe un chemin σ dont les k_0 premiers états vérifient ϕ_1 et pas ϕ_2. Cette partie de σ comporte un cycle et permet de construire un chemin infini d'origine q qui ne vérifie pas $\phi_1 \mathsf{U} \phi_2$, d'où la contradiction.
On vérifie maintenant par récurrence que $g^k(\emptyset) = [\mathsf{A}\phi_1 \mathsf{U}_k \phi_2]$. Remarquons tout d'abord que $g^0(\emptyset) = \emptyset = \mathsf{A}\phi_1 \mathsf{U}_0 \phi_2$. Il nous suffit maintenant d'établir que $[\mathsf{A}\phi_1 \mathsf{U}_{k+1} \phi_2] = \phi_2 \cup ([\mathsf{AXA}\phi_1 \mathsf{U}_k \phi_2] \cap \phi_1)$. Si $q \vDash \phi_2 \cup ([\mathsf{AXA}\phi_1 \mathsf{U}_k \phi_2] \cap \phi_1)$ alors soit $q \vDash \phi_2$, et alors $q \vDash \mathsf{A}\phi_1 \mathsf{U}_{k+1} \phi_2$; soit $q \vDash (\phi_1 \cap (\mathsf{AXA}\phi_1 \mathsf{U}_k \phi_2))$. Dans ce cas, considérons un chemin maximal σ d'origine q. On a $|\sigma| > 1$, $\sigma(0) \vDash \phi_1$ et $\sigma(1) \vDash \mathsf{A}\phi_1 \mathsf{U}_k \phi_2$ donc $\sigma(1) \ldots \sigma(i) \ldots \vDash \phi_1 \mathsf{U}_k \phi_2$ donc $\sigma \vDash \phi_1 \mathsf{U}_{k+1} \phi_2$ ce qui donne une inclusion. Inversement, supposons que $q \vDash \mathsf{A}\phi_1 \mathsf{U}_{k+1} \phi_2$ et $q \nvDash \phi_2$ (q a alors nécessairement un successeur). Considérons un successeur q' de q et un chemin σ' d'origine q'. Soit σ le chemin obtenu à partir de σ' en ajoutant q en tête. σ satisfait $\phi_1 \mathsf{U}_{k+1} \phi_2$ et comme $\sigma(0)$ ne satisfait pas ϕ_2, $\sigma(0)$ satisfait ϕ_1 et $\sigma(1)$ satisfait $\mathsf{A}\phi_1 \mathsf{U}_k \phi_2$. On en déduit que $q \vDash \phi_1$ et que tous ses sucesseurs satisfont $\mathsf{A}\phi_1 \mathsf{U}_k \phi_2$ ce qui donne le résultat.
La fonction g étant continue, on en conclu que $\mathsf{A}\phi_1 \mathsf{U} \phi_2$ est le plus petit point fixe de la fonction g, ce qui permet d'implanter l'opérateur en *MEC*.

Implantation

```
function AU(phi1:state; phi2:state) return r:state;
begin
  r = phi2 \/ ((src(rtgt(*)) - src(rtgt(* - r))) /\ phi1)
end.
```

3.3.3 Un jour

EF

Relation avec EU On a, par définition, l'égalité suivante :

$$[\mathsf{EF}\phi] = [\mathsf{E}\,tt\,\mathsf{U}\phi]$$

On en déduit une implantation de cet opérateur en *MEC*.

Implantation

```
function EF(phi:state) return r:state;
begin
  r = phi \/ src(rtgt(r))
end.
```

AF

Relation avec AU On a, par définition, l'égalité suivante :

$$[\mathsf{AF}\phi] = [\mathsf{A}\,tt\,\mathsf{U}\phi]$$

On en déduit une implantation de cet opérateur en *MEC*.

Implantation

```
function AF(phi:state) return r:state;
begin
  r = phi \/ (src(rtgt(*)) - src(rtgt(* - r)))
end.
```

3.3.4 Toujours

EG

Relation avec AU

$$\begin{aligned}[\mathsf{EG}\phi] &= [\mathsf{E}\neg(\mathsf{F}\neg\phi)] \\ &= [\neg(\mathsf{AF}\neg\phi)] \\ &= [\neg(\mathsf{A}\,tt\,\mathsf{U}\neg\phi)]\end{aligned}$$

Grâce au théorême de dualité, on en déduit une définition par plus grand point fixe de l'opérateur EG.

Implantation

```
function EG(phi:state) return r:_state
begin
  r =  phi - (src(rtgt(*)) - src(rtgt(r))))
end.
```

AG

Relation avec EU

$$\begin{aligned}[\mathsf{AG}\phi] &= [\mathsf{A}\neg(\mathsf{F}\neg\phi)] \\ &= [\neg(\mathsf{EF}\neg\phi)] \\ &= [\neg(\mathsf{E}\,tt\,\mathsf{U}\neg\phi)]\end{aligned}$$

Grâce au théorême de dualité, on en déduit une définition par plus grand point fixe de l'opérateur AG.

Implantation

```
function AG(phi:state) return r:_state
begin
  r = phi - src(rtgt(* - r))
end.
```

3.3.5 Weak until

EW

Définition par point fixe On définit les opérateurs G_k et W_k, par analogie avec U_k par les relations suivantes :

$$\sigma \vDash \mathsf{G}_k\phi \;\Leftrightarrow\; (|\sigma| > k) \;\wedge\; (\forall\, i < k,\; \sigma(i) \vDash \phi)$$
$$\sigma \vDash \phi_1\mathsf{W}_k\phi_2 \;\Leftrightarrow\; (\sigma \vDash \phi_1\mathsf{U}_k\phi_2) \;\vee\; (\sigma \vDash \mathsf{G}_k\phi_1)$$

Considérons à nouveau la fonction $f : \psi \mapsto \phi_2 \cup ([\mathsf{EX}\psi] \cap \phi_1)$. Nous allons montrer que $\mathsf{E}\phi_1\mathsf{W}\phi_2$ est le plus grand point fixe de la fonction f.
Vérifions tout d'abord que $[\mathsf{E}\phi_1\mathsf{W}\phi_2] = \bigcap_{k\geq 0}[\mathsf{E}\phi_1\mathsf{W}_k\phi_2]$. Il suffit de montrer que $\sigma \vDash \phi_1\mathsf{W}\phi_2 \;\Leftrightarrow\; \sigma \vDash \bigwedge_{k\geq 0}\phi_1\mathsf{W}_k\phi_2$. Si $\sigma \vDash \phi_1\mathsf{W}\phi_2$ alors quel que soit k, $\sigma \vDash \phi_1\mathsf{W}_k\phi_2$ (si $s \vDash \mathsf{G}\phi_1$ et $|\sigma| = \infty$ alors quel que soit k, $s \vDash \mathsf{G}_k\phi_1$, sinon il existe k_0 tel que $s(k_0) \vDash \phi_2$ et $\forall k < k_0,\; s(k) \vDash \phi_1$; et, dans ce cas, pour $k \leq k_0$, $\sigma \vDash \mathsf{G}_k\phi_1$ et pour $k > k_0$, $\sigma \vDash \phi_1\mathsf{U}_k\phi_2$) donc $\sigma \vDash \bigwedge_{k\geq 0}\phi_1\mathsf{W}_k\phi_2$. Inversement, si $\sigma \vDash \bigwedge_{k\geq 0}\phi_1\mathsf{W}_k\phi_2$, soit pour tout k σ satisfait $\mathsf{G}_k\phi_1$ et dans ce cas σ est infini et satisfait $\mathsf{G}\phi_1$ et par conséquent $\phi_1\mathsf{W}\phi_2$; soit il existe un entier k tel que σ satisfasse $\phi_1\mathsf{U}_k\phi_2$ et alors σ satisfait $\phi_1\mathsf{W}\phi_2$ ce qui donne le résulat.
On vérifie par récurrence que $f^k(Q) = [\mathsf{E}\phi_1\mathsf{W}_k\phi_2]$:

$$\begin{array}{lll}
\text{Base :} & f^0(Q) & = Q = \mathsf{E}\phi_1\mathsf{W}_0\phi_2 \\
\text{Induction :} & f^{k+1}(Q) & = \phi_2 \cup ([\mathsf{EX}f^k(Q)] \cap \phi_1) \\
 & & = \phi_2 \cup ([\mathsf{EX}(\mathsf{E}\phi_1\mathsf{W}_k\phi_2)] \cap \phi_1) \\
 & & = \phi_2 \cup ([\mathsf{EX}(\mathsf{E}\phi_1\mathsf{U}_k\phi_2)] \cap \phi_1) \cup ([\mathsf{EX}(\mathsf{EG}_k\phi_2)] \cap \phi_1) \\
 & & = (\phi_2 \cup ([\mathsf{EX}(\mathsf{E}\phi_1\mathsf{U}_k\phi_2)] \cap \phi_1)) \cup ([\mathsf{EX}(\mathsf{EG}_k\phi_2)] \cap \phi_1) \\
 & & = \; [\mathsf{E}\phi_1\mathsf{U}_{k+1}\phi_2] \cup [\mathsf{EG}_{k+1}\phi_1] \\
 & & = \; [\mathsf{E}\phi_1\mathsf{W}_{k+1}\phi_2]
\end{array}$$

On en déduit que $[\mathsf{E}\phi_1\mathsf{W}\phi_2]$ est le plus grand point fixe de la fonction f.

```
function EW(phi1:state; phi2:state) return r:_state
begin
  r = phi2 \/ (src(rtgt(r)) /\ phi1)
end.
```

AW

Définition par point fixe Considérons à nouveau la fonction $g : \psi \mapsto \phi_2 \cup ([\mathsf{AX}\psi] \cap \phi_1)$.
Nous avons déjà montré que $\sigma \vDash \phi_1\mathsf{W}\phi_2 \;\Leftrightarrow\; \sigma \vDash \bigwedge_{k\geq 0}\phi_1\mathsf{W}_k\phi_2$. On en déduit que $[\mathsf{A}\phi_1\mathsf{W}\phi_2] = \bigcap_{k\geq 0}[\mathsf{A}\phi_1\mathsf{W}_k\phi_2]$.
On vérifie par récurrence que $g^k(Q) = [\mathsf{A}\phi_1\mathsf{W}_k\phi_2]$. Tout d'abord, remarquons que $g^0(Q) = [\mathsf{A}\phi_1\mathsf{W}_0\phi_2] = Q$. Il suffit ensuite de montrer que $[\mathsf{A}\phi_1\mathsf{W}_{k+1}\phi_2] = \phi_2 \cup ([\mathsf{AXA}\phi_1\mathsf{W}_k\phi_2] \cap \phi_1)$.
Si $q \in \phi_2 \cup ([\mathsf{AXA}\phi_1\mathsf{W}_k\phi_2] \cap \phi_1)$ alors soit $q \vDash \phi_2$ et dans ce cas $q \in [\mathsf{A}\phi_1\mathsf{W}_{k+1}\phi_2]$; soit $q \vDash \phi_1 \wedge \mathsf{AXA}\phi_1\mathsf{W}_k\phi_2$. Considérons alors un chemin maximal σ d'origine q. On a $|\sigma| > 1$, $\sigma(0) \vDash \phi_1$ et $\sigma(1) \vDash \mathsf{A}\phi_1\mathsf{W}_k\phi_2$ donc $\sigma \vDash \phi_1\mathsf{W}_{k+1}\phi_2$. Réciproquement, supposons que $q \in \mathsf{A}\phi_1\mathsf{W}_{k+1}\phi_2$ et que $q \not\vDash \phi_2$. Considérons un successeur q' de q (s'il n'en existait pas, le seul chemin maximal d'origine q serait réduit à q et ne pourrait satisfaire $\phi_1\mathsf{W}_{k+1}\phi_2$) et un chemin σ' d'origine q'. Soit σ le chemin obtenu à partir de σ' en ajoutant q en tête. σ satisfait $\mathsf{A}\phi_1\mathsf{W}_{k+1}\phi_2$ et comme $\sigma(0)$ ne satisfait pas ϕ_2, $\sigma(0)$ satisfait ϕ_1 et σ' satisfait $\phi_1\mathsf{W}_k\phi_2$. On en déduit que $q \vDash \phi_1$, q admet un succeseur et que tous ses successeurs satisfont $\mathsf{A}\phi_1\mathsf{W}_k\phi_2$ ce qui donne le résultat.
On en déduit que $[\mathsf{A}\phi_1\mathsf{W}\phi_2]$ est le plus grand point fixe de la fonction g.

Implantation

```
function AW(phi1:state; phi2:state) return r:_state
begin
  r = phi2 \/ ((src(rtgt(*)) - src(rtgt(* - r))) /\ phi1)
end.
```

3.4 Sémantique des chemins infinis ou équitables

On conserve les notations de la partie précédente.
On considère de plus une condition d'équité F et on note Q_F l'ensemble des états q de l'automate tels qu'il existe un chemin équitable d'origine q. On note également $C_F(q)$ l'ensemble des chemins équitables d'origine q.
Les fonctions écrites en MEC prennent l'ensemble Q_F comme argument. Il est expliqué dans la section 3.5.1 comment le calculer.

3.4.1 État suivant

EfX

Définition ensembliste

$$
\begin{aligned}
[\mathsf{EfX}\phi] &= \{q \in Q \;/\; \exists\, \sigma \in C_F(q),\ \sigma(1) \vDash \phi\} \\
&= \{q \in Q \;/\; \exists\, \sigma \in C(q),\ \sigma(1) \vDash (\phi \;\wedge\; Q_F))\} \\
&= \mathsf{EX}(\phi \;\wedge\; Q_F)
\end{aligned}
$$

Implantation

```
function EfX(phi:state; qf:state) return r:state;
begin
  r = src(rtgt(phi /\ qf))
end.
```

AfX

Définition ensembliste

$$[\mathsf{AfX}\phi] = \{q \in Q \;/\; \forall\, \sigma \in C_F(q),\ \sigma(1) \vDash \phi\}$$

Relation avec EfX

$$
\begin{aligned}
q \vDash \mathsf{AfX}\phi \;&\Leftrightarrow\; \forall\, \sigma \in C_F(q),\ \sigma(1) \vDash \phi \\
&\Leftrightarrow\; \neg(\exists \sigma \in C_F(q),\ \sigma(1) \vDash \neg\phi)
\end{aligned}
$$

On en déduit que $[\mathsf{AfX}\phi] = Q - [\mathsf{EfX}(Q_F - \phi)]$

Implantation

```
function AfX(phi:state; qf:state) return r:state;
begin
  r = * - src(rtgt(qf - phi))
end.
```

3.4.2 EfU et AfW

EfU

Relation avec EU

$$\begin{aligned} q \models \mathsf{Ef}\phi_1\mathsf{U}\phi_2 &\Leftrightarrow \exists\,\sigma \in C_F(q),\ \sigma \models \phi_1\mathsf{U}\phi_2 \\ &\Leftrightarrow \exists\,\sigma \in C(q),\ (\sigma \models \phi_1\mathsf{U}\phi_2)\ \wedge\ \sigma \text{ équitable} \\ &\Leftrightarrow \exists\,\sigma \in C(q),\ \sigma \models \phi_1\mathsf{U}(\phi_2 \cap Q_F) \\ &\Leftrightarrow q \models \mathsf{E}\phi_1\mathsf{U}(\phi_2 \cap Q_F) \end{aligned}$$

Implantation

```
function EfU(phi1:state; phi2:state; qf:state) return r:state;
begin
  r = (phi2 /\ qf) \/ (src(rtgt(r)) /\ phi1)
end.
```

AfW

Définition par point fixe Considérons la fonction $g : \psi \mapsto (\phi_2 \cup \neg Q_F) \cup ([\mathsf{AfX}\psi] \cap \phi_1)$.
Nous avons déjà montré que $\sigma \models \phi_1\mathsf{W}\phi_2 \Leftrightarrow \sigma \models \bigwedge_{k\geq 0}\phi_1\mathsf{W}_k\phi_2$. On en déduit que $[\mathsf{Af}\phi_1\mathsf{W}\phi_2] = \bigcap_{k\geq 0}[\mathsf{Af}\phi_1\mathsf{W}_k\phi_2]$.
On vérifie par récurrence que $g^k(Q) = [\mathsf{Af}\phi_1\mathsf{W}_k\phi_2]$. Tout d'abord, remarquons que $g^0(Q) = [\mathsf{Af}\phi_1\mathsf{W}_0\phi_2] = Q$. Il suffit ensuite de montrer que $[\mathsf{Af}\phi_1\mathsf{W}_{k+1}\phi_2] = (\phi_2 \cup \neg Q_F) \cup ([\mathsf{AfXAf}\phi_1\mathsf{W}_k\phi_2] \cap \phi_1)$.
Si $q \in (\phi_2 \cup \neg Q_F) \cup ([\mathsf{AfXAf}\phi_1\mathsf{W}_k\phi_2] \cap \phi_1)$ alors soit $q \models \phi_2 \cup \neg Q_F$ et dans ce cas $q \in [\mathsf{Af}\phi_1\mathsf{W}_{k+1}\phi_2]$; soit $q \models \phi_1 \wedge \mathsf{AfXAf}\phi_1\mathsf{W}_k\phi_2$. Considérons un chemin σ équitable d'origine q. On a $\sigma(0) \models \phi_1$ et $\sigma(1) \models \mathsf{Af}\phi_1\mathsf{W}_k\phi_2$ donc $\sigma \models \phi_1\mathsf{W}_{k+1}\phi_2$. Réciproquement, supposons que $q \in \mathsf{Af}\phi_1\mathsf{W}_{k+1}\phi_2$ et que $q \not\models \phi_2$. Considérons, s'il en existe un, un successeur q' de q (s'il n'en existe pas, le résultat est acquis puisque $q \in \neg Q_F$) et un chemin équitable σ' d'origine q'. Soit σ le chemin obtenu à partir de σ' en ajoutant q en tête. σ satisfait $\mathsf{Af}\phi_1\mathsf{W}_{k+1}\phi_2$ et comme $\sigma(0)$ ne satisfait pas ϕ_2, $\sigma(0)$ satisfait ϕ_1 et σ' satisfait $\phi_1\mathsf{W}_k\phi_2$. On en déduit que $q \models \phi_1$ et que tous ses sucesseurs sur des chemins équitaux satisfont $\mathsf{Af}\phi_1\mathsf{W}_k\phi_2$ ce qui donne le résultat.
On en déduit que $[\mathsf{Af}\phi_1\mathsf{W}\phi_2]$ est le plus grand point fixe de la fonction g.

Implantation

```
function AfW(phi1:state; phi2:state; qf:state) return r:_state
begin
  r = phi2 \/ (phi1 - src(rtgt(qf - r)))
end.
```

3.4.3 EfF et AfG

EfF

Relation avec EfU On a, de par la définition des opérateurs, l'égalité :

$$[\mathsf{EfF}\phi] = [\mathsf{Ef}\,tt\mathsf{U}\phi]$$

Implantation

```
function EfF(phi:state; qf:state) return r:state;
begin
  r = (phi /\ qf) \/ src(rtgt(r))
end.
```

AfG

Relation avec EfU

$$\begin{aligned}[\mathsf{AfG}\phi] &= [\neg\mathsf{Ef}\neg G\phi]\\ &= [\neg\mathsf{EfF}\neg\phi]\\ &= [\neg\mathsf{Ef}\,tt\mathsf{U}\neg\phi]\end{aligned}$$

Implantation

```
function AfG(phi:state; qf:state) return r:_state
begin
  r = * - ((qf - phi) \/ src(rtgt(* - r)))
end.
```

3.4.4 EfG et AfF

EfG

Définition par point fixe Soit σ un chemin équitable vérifiant $\mathsf{G}\phi$. σ étant infini, il existe un état q_0 qui y apparaît une infinité de fois. Soit k_0 l'indice de la première apparition de q_0. A partir de l'état k_0, les états de σ sont dans des cycles qui comportent des sommets équitables et dont tous les éléments vérifient ϕ. Ces sommets sont donc dans l'ensemble $L = rtgt(loop(tgt(F), tgt(\phi)))$.
Soit h la fonction $\psi \mapsto L \cup (\mathsf{EX}\psi \ \wedge \ \phi)$. On peut vérifier par récurrence (de manière analogue à ce que nous avons fait pour les fonctions f et g) que le plus petit point fixe de h est l'ensemble des états q tel qu'il existe un chemin menant de q à un état de L sur lequel ϕ est vérifié. Le plus petit point fixe de h est donc inclus dans $\mathsf{EfG}\phi$. Réciproquement, par ce qui précède, tout chemin satisfaisant $\mathsf{G}\phi$ est de cette forme. On en déduit que $\mathsf{EfG}\phi$ est le plus petit point fixe de h.

Implantation

```
function EfG(phi:state; L:state) return r:state;
begin
  r = L \/ (src(rtgt(r)) /\ phi);
end.
```

AfF

Relation avec EfG On a, de par la définition des opérateurs, les égalités :

$$\begin{aligned}[\mathsf{AfG}\phi] &= [\neg\mathsf{Ef}\neg G\phi]\\ &= [\neg\mathsf{EfF}\neg\phi]\end{aligned}$$

Implantation

```
function AfG(phi:state; L:state) return r:_state
begin
  r = * - (L \/ (src(rtgt(* - r)) /\ phi));
end.
```

3.4.5 EfW et AfU

EfW

Relation avec EfU et EfG La quantification existentielle étant distributive sur le ou logique, on a :

$$[\mathsf{Ef}\phi_1\mathsf{W}\phi_2] = [\mathsf{Ef}\phi_1\mathsf{U}\phi_2] \cup [\mathsf{EfG}\phi_1]$$

Implantation

```
function EfW(phi1:state; phi2:state; qf:state; L:state) return r:state;
var
  rEfU : state;
  rEfG : _state;
begin
  rEfG = L \/ (src(rtgt(rEfG)) /\ phi1);
  rEfU = (phi2 /\ qf) \/ (src(rtgt(rEfU)) /\ phi1);
  r = EfU \/ EfG
end.
```

AfU

Relation avec EfW Montrons tout d'abord que si σ est un chemin infini, $\sigma \models \neg(\phi_1 \mathsf{U} \phi_2) \Leftrightarrow \sigma \models (\phi_1 \wedge \neg\phi_2)\mathsf{W}(\neg\phi_1 \wedge \neg\phi_2)$.
Si $\sigma \models \neg(\phi_1 \mathsf{U} \phi_2)$, quel que soit l'entier i, soit $\sigma(i) \not\models \phi_2$, soit $\sigma(i) \models \phi_2 \wedge \exists\, j < i,\ \sigma(j) \not\models \phi_1$. Dans le cas où pour tout i $\sigma(i) \not\models \phi_2$, soit pour tout i, $\sigma(i) \models \phi_1$ et alors $\sigma \models \mathsf{G}(\phi_1 \wedge \neg\phi_2)$; soit il existe i tel que $\sigma(i) \not\models \phi_1$ et alors $\sigma \models (\phi_1 \wedge \neg\phi_2)\mathsf{U}(\neg\phi_1 \wedge \neg\phi_2)$. Sinon, dans le cas où il existe un entier i tel qu'il existe $j < i$ tel que $\sigma(j) \not\models \phi_1$, en considérant les plus petits entiers i et j, on montre que $\sigma \models (\phi_1 \wedge \neg\phi_2)\mathsf{U}(\neg\phi_1 \wedge \neg\phi_2)$.
Réciproquement, si $\sigma \models (\phi_1 \wedge \neg\phi_2)\mathsf{W}(\neg\phi_1 \wedge \neg\phi_2)$, soit $\sigma \models \mathsf{G}(\phi_1 \wedge \neg\phi_2)$ et alors $\sigma \models \mathsf{G}\neg\phi_2$ et σ ne peut satisfaire $\phi_1 \mathsf{U} \phi_2$; soit $\sigma \models (\phi_1 \wedge \neg\phi_2)\mathsf{U}(\neg\phi_1 \wedge \neg\phi_2)$. Notons alors i le plus petit entier tel que $\sigma(i) \models \neg\phi_1 \wedge \neg\phi_2$ et supposons que $\sigma \models \phi_1 \mathsf{U} \phi_2$. Notons i' l'entier correspondant. Si $i' \leq i$ c'est absurde car alors $\sigma(i') \not\models \phi_2$. Si $i' > i$, c'est également absurde puisque $\sigma(i) \not\models \phi_1$.
On en déduit les relations suivantes :

$$\begin{aligned}[\mathsf{Af}\phi_1 \mathsf{U} \phi_2] &= [\neg\mathsf{Ef}\neg(\phi_1 \mathsf{U} \phi_2)] \\ &= [\neg\mathsf{Ef}(\phi_1 \wedge \neg\phi_2)\mathsf{W}(\neg\phi_1 \wedge \neg\phi_2)]\end{aligned}$$

```
function AfU(phi1:state; phi2:state; qf:state; L:state) return r:state;
var
  rEfU : state;
  rEfG : _state;
begin
  rEfG = L \/ (src(rtgt(rEfG)) /\ (phi1 - phi2));
  rEfU = (qf - (phi1 \/ phi2)) \/ (src(rtgt(rEfU)) /\ (phi1 - phi2));
  r = * - (EfU \/ EfG)
end.
```

3.5 Utilisation

3.5.1 Utilisation directe

Il est possible d'utiliser directement les focntions *MEC* pour implanter les formules CTL. Ceci impose cependant de respecter les règles de syntaxe des équations de point fixe *MEC* qui interdit en particulier l'imbrication d'appels de fonctions.
Les opérateurs ont alors une syntaxe fonctionnelle.

Sémantique des chemins maximaux

Expression CTL	**Équivalent MEC**	**Expression CTL**	**Équivalent MEC**
$\mathsf{EX}\phi$	`EX(phi)`	$\mathsf{AX}\phi$	`AX(phi)`
$\mathsf{E}\phi_1\mathsf{U}\phi_2$	`EU(phi1;phi2)`	$\mathsf{A}\phi_1\mathsf{U}\phi_2$	`AU(phi1;phi2)`
$\mathsf{E}\phi_1\mathsf{W}\phi_2$	`EW(phi1;phi2)`	$\mathsf{A}\phi_1\mathsf{W}\phi_2$	`AW(phi1;phi2)`
$\mathsf{EF}\phi$	`EF(phi)`	$\mathsf{AF}\phi$	`AF(phi)`
$\mathsf{EG}\phi$	`EG(phi)`	$\mathsf{AG}\phi$	`AG(phi)`

Sémantique des chemins équitables

Les fonctions *MEC* correspondant à cette sémantique prennent un argument supplémentaire : Q_F, l'ensemble des états q tels qu'il existe un chemin équitable σ d'origine q. Cet ensemble doit être préalablement calculé par une équation de point fixe :

- Si on s'intéresse en fait aux chemins infinis, l'ensemble qe est l'ensemble des états origines d'un chemin infini. Il peut être calculé par la fonction suivante :

```
function infini() return r:state;
begin
  r = src(rtgt(r))
end.
```

- Si la condtion d'équité est définie par un ensemble E, l'ensemble qe peut être défini par :

```
function equitable(f:state) return r:state;
var
  l : state;
begin
  l = loop(tgt(f), *);
  r = l \/ src(rtgt(l))
end.
```

  L'expression $loop(x, y)$ désigne l'ensemble des états situés sur un cycle dont toutes les transitions sont dans y et au moins une est dans x.

Expression CTL	Équivalent MEC	Expression CTL	Équivalent MEC
EfXϕ	`EfX(phi ;qf)`	AfXϕ	`AfX(phi ;qf)`
Efϕ_1Uϕ_2	`EfU(phi1 ;phi2 ;qf)`	Afϕ_1Uϕ_2	`AfU(phi1 ;phi2 ;qf ;f)`
Efϕ_1Wϕ_2	`EfW(phi1 ;phi2 ;qf ;f)`	Afϕ_1Wϕ_2	`AfW(phi1 ;phi2 ;qf)`
EfFϕ	`EfF(phi ;qf)`	AfFϕ	`AfF(phi ;qf ;f)`
EfGϕ	`EfG(phi ;qf ;f)`	AfGϕ	`AfG(phi ;qf)`

3.5.2 Utilisation avec Autovar

Cette utilisation a été décrite dans la section 2.3.5.
Elle permet de directement spécifier les propriétés relatives au produit d'automates dans une forme très voisine de la syntaxe CTL usuelle. Le code est traduit directement en *SMV* puis en *MEC* en utilisant les fonctions décrites ci-dessus.

Conclusion

Pour conclure, je souhaiterais développer deux points : ce que ce stage m'a apporté d'une part et quelles continuations il ouvre d'autre part.
La première chose que ce travail m'a permis d'acquérir, ce sont de solides connaissances de base dans tout ce qui touche à la vérification. Je ne connaissais absolument pas ce domaine, et j'ai pu découvrir des applications intéressantes de sujets bien connus – comme la théorie des automates – ou moins comme la logique linéaire. Ce stage m'a confronté également pour la première fois à la question de la conception d'un langage de programmation original dans sa globalité et aux problèmes qu'elle pose.
Les perspectives ouvertes sont nombreuses. En ce qui concerne le langage de modification, on peut commencer par vouloir développer sa richesse. Cela peut aller de simples développements syntaxiques – valeurs initiales des files, amélioration du passage des arguments, nouveaux opérateurs – à l'ajout de notions – types structurés, piles, temporisation... On pourrait également ensuite imaginer d'établir de nouvelles passerelles vers d'autres langages. La partie spécification pourrait être prolongée par exemple en développant des outils permettant de traiter plusieurs conditions d'équité. Enfin, un sujet intéressant serait d'utiliser les passerelles réalisées pour comparer les trois outils de vérification et leurs performances.

Bibliographie

[1] André Arnold

Systèmes de transitions finis et sémantique des processus communiquants

Études et recherches en Informatique

[2] F. Laroussinie, A. Petit, Ph. Schoebelen

Le model-checking, une technique de vérification en plein essor

LSV

[3] Mec Hand Bool

Didier Begay

LaBRI, Université Bordeaux 1

[4] The SMV system

K. L. McMillan

Carnegi-Mellon University

[5] Basic Spin manual