Inférence de flots d'information pour ML

Vincent Simonet

Mémoire de DEA
sous la direction de François Pottier
INRIA Rocquencourt – Projet Cristal

Mars 2001

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

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

Partie **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 $\mathrm{ML_{re}}$

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 $\mathrm{ML_{re}}$ 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

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

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

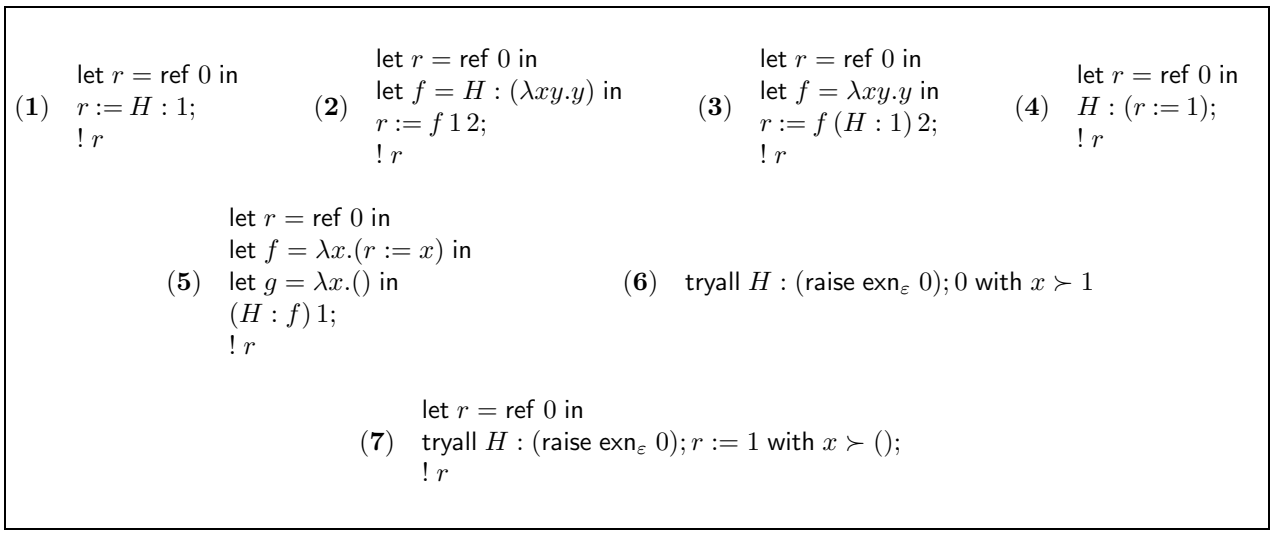

Fig. 2: Exemples d'expressions étiquetées

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 :

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

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 $\mathrm{ML_{re}}$:

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

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 $\mathrm{ML_{re}}$, notée $\mathrm{strip}\, e$, 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.

Partie **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 $\langle \dots \mid \dots \rangle$ aux classes syntaxiques des réponses, des valeurs et des expressions :

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

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 $\langle \dots \mid \dots \rangle$ ne sont pas imbriquées. Une expression est *simple* si elle ne comporte pas de construction $\langle \dots \mid \dots \rangle$. 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 :*
- $[\![e]\!]_1$ *la « projection gauche » de e obtenue à partir de e en ne conservant que les parties gauches des sous-expressions* $\langle \dots \mid \dots \rangle$.
- $[\![e]\!]_2$ *la « projection droite » de e obtenue à partir de e en ne conservant que les parties droites des sous-expressions* $\langle \dots \mid \dots \rangle$.

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

$$[\![\langle e_1 \mid e_2 \rangle]\!]_1 = e_1 \qquad\qquad [\![\langle e_1 \mid e_2 \rangle]\!]_2 = e_2$$

On note également $[\![e]\!]_\varnothing = e$ *et on étend les projections aux états mémoire :* $[\![\sigma]\!]_i = \{m \mapsto [\![\sigma(m)]\!]_i \mid m \in \mathrm{dom}(\sigma)$ *et* $[\![\sigma(m)]\!]_i \neq \varnothing\}$.

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 :

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

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 : $[\![e]\!]_1 = (\mathsf{let}\ a = 1\ \mathsf{in}\ (\lambda x.x)\, a) = \mathrm{strip}\, e_1$ et $[\![e]\!]_2 = (\mathsf{let}\ a = 2\ \mathsf{in}\ (\lambda x.x)\, a) = \mathrm{strip}\, e_2$.

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 :

$$\langle e_1 \mid e_2\rangle[x \Leftarrow v] = \langle e_1[x \Leftarrow [\![v]\!]_1] \mid e_2[x \Leftarrow [\![v]\!]_2]\rangle$$

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 $[\![e]\!]_i \,/\, [\![\sigma]\!]_i$ soit un état bloqué.*

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

$$(\lambda x.e)\, v \,/\, \sigma \xrightarrow{i} e[x \Leftarrow v] \,/\, \sigma \qquad (\beta)$$

$$\mathsf{let}\ x = v\ \mathsf{in}\ e \,/\, \sigma \xrightarrow{i} e[x \Leftarrow v] \,/\, \sigma \qquad (\text{let})$$

$$\mathsf{ref}\ v \,/\, \sigma \xrightarrow{i} m \,/\, \sigma \oplus [m \mapsto \mathrm{new}_i\, v] \qquad (\text{ref})$$

$$m := v \,/\, \sigma \xrightarrow{i} () \,/\, \sigma \ominus [m \mapsto \mathrm{update}_i(\sigma(m)/v)] \qquad (\text{assign})$$

$$!\, m \,/\, \sigma \xrightarrow{i} [\![\sigma(m)]\!]_i \,/\, \sigma \qquad (\text{deref})$$

$$\mathsf{raise}\ (\mathsf{exn}_\varepsilon\ v) \,/\, \sigma \xrightarrow{i} \mathsf{raised}_\varepsilon\ v \,/\, \sigma \qquad (\text{raise})$$

$$\mathsf{bind}\ x = v\ \mathsf{in}\ e \,/\, \sigma \xrightarrow{i} e[x \Leftarrow v] \,/\, \sigma \qquad (\text{bind})$$

$$\mathsf{try}_\varepsilon\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ e \,/\, \sigma \xrightarrow{i} e[x \Leftarrow v] \,/\, \sigma \qquad (\text{handle})$$

$$\mathsf{tryall}\ (\mathsf{raised}_\varepsilon\ v)\ \mathsf{with}\ x \succ e \,/\, \sigma \xrightarrow{i} e[x \Leftarrow (\mathsf{exn}_\varepsilon\ v)] \,/\, \sigma \qquad (\text{handle-all})$$

$$E[a] \,/\, \sigma \xrightarrow{i} a \,/\, \sigma \qquad \text{si } E \not\lll [\![a]\!]_1 \text{ et } E \not\lll [\![a]\!]_2 \qquad (\text{throw-context})$$

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

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

Règles « lift » :

$$\langle v_1 \mid v_2 \rangle\, v \,/\, \sigma \xrightarrow{\varnothing} \langle v_1\, [\![v]\!]_1 \mid v_2\, [\![v]\!]_2 \rangle \,/\, \sigma \qquad (\text{lift-app})$$

$$\langle v_1 \mid v_2 \rangle := v \,/\, \sigma \xrightarrow{\varnothing} \langle v_1 := [\![v]\!]_1 \mid v_2 := [\![v]\!]_2 \rangle \,/\, \sigma \qquad (\text{lift-assign})$$

$$!\, \langle v_1 \mid v_2 \rangle \,/\, \sigma \xrightarrow{\varnothing} \langle !\, v_1 \mid !\, v_2 \rangle \,/\, \sigma \qquad (\text{lift-deref})$$

$$\mathsf{raise}\ \langle v_1 \mid v_2 \rangle \,/\, \sigma \xrightarrow{\varnothing} \langle \mathsf{raise}\ v_1 \mid \mathsf{raise}\ v_2 \rangle \,/\, \sigma \qquad (\text{lift-raise})$$

$$E[\langle a_1 \mid a_2 \rangle] \,/\, \sigma \xrightarrow{\varnothing} \langle [\![E]\!]_1[a_1] \mid [\![E]\!]_2[a_2] \rangle \,/\, \sigma \qquad (\text{lift-context})$$

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

Primitives d'écriture dans l'état mémoire :

$$\mathrm{new}_1\, v = \langle v \mid \varnothing \rangle \qquad \mathrm{update}_1(v_0/v) = \langle v \mid [\![v_0]\!]_2 \rangle$$

$$\mathrm{new}_2\, v = \langle \varnothing \mid v \rangle \qquad \mathrm{update}_2(v_0/v) = \langle [\![v_0]\!]_1 \mid v \rangle$$

$$\mathrm{new}_\varnothing\, v = v \qquad \mathrm{update}_\varnothing(v_0/v) = v$$

Si $m \notin \mathrm{dom}(\sigma)$, $\sigma \oplus [m \mapsto v]$ dénote l'état mémoire qui étend σ et associe v à m. Si $m \in \mathrm{dom}(\sigma)$, $\sigma \ominus [m \mapsto v]$ 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

Preuve 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 $[\![v_1]\!]_i$ n'est pas de la forme $\lambda x.f$ pour $i = 1$ et 2 et donc que $[\![e \,/\, \sigma]\!]_i$ est un état bloqué.

$e = \mathsf{let}\ x = v\ \mathsf{in}\ e'$ **ou** $e = \mathsf{ref}\ v$ $e \,/\, \sigma$ 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 $[\![v_1]\!]_i$ n'est pas de la forme m pour $i = 1$ et 2 et donc que $[\![e \,/\, \sigma]\!]_i$ 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 $[\![v]\!]_i$ n'est pas de la forme m pour $i = 1$ et 2 et donc que $[\![e \,/\, \sigma]\!]_i$ 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 $[\![v]\!]_i$ n'est pas de la forme $\mathsf{exn}_\varepsilon\ w$ pour $i = 1$ et 2 et donc que $[\![e \,/\, \sigma]\!]_i$ 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 $e_1 \,/\, \sigma$ est bloqué et on obtient le résultat par hypothèse d'induction.

$e = \langle e_1 \mid e_2\rangle$ $e \,/\, \sigma$ est un état bloqué. $e_1 \,/\, \sigma$ et $e_2 \,/\, \sigma$ ne se réduisent pas et nécessairement e_i n'est pas une réponse pour $i = 1$ ou 2. On en déduit que $e_i \,/\, [\![\sigma]\!]_i$ 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 $e \,/\, \sigma \to e' \,/\, \sigma'$ alors $[\![e]\!]_i \,/\, [\![\sigma]\!]_i \to^{=} [\![e']\!]_i \,/\, [\![\sigma']\!]_i$ pour $i \in \{1, 2\}$.*

Preuve On vérifie successivement, en observant chaque règle de réduction, que si $e \,/\, \sigma \xrightarrow{i} e' \,/\, \sigma'$ pour $i \in \{1, 2\}$ alors $e \,/\, [\![\sigma]\!]_i \to e' \,/\, [\![\sigma']\!]_i$ puis que si $e \,/\, \sigma \to e' \,/\, \sigma'$ alors $[\![e]\!]_i \,/\, [\![\sigma]\!]_i \to^{=} [\![e']\!]_i \,/\, [\![\sigma']\!]_i$ pour $i \in \{1, 2\}$. □

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

Preuve On note, pour $i \in \{1, 2\}$, $e_i = [\![e]\!]_i$ et $\sigma_i = [\![\sigma]\!]_i$. 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 $e \,/\, \sigma$ alors on peut construire une réduction infinie à partir de $e_1 \,/\, \sigma_1$ ou de $e_2 \,/\, \sigma_2$, ce qui est impossible puisque $e_1 \,/\, \sigma_1$ et $e_2 \,/\, \sigma_2$ se réduisent vers des réponses. On en déduit que $e \,/\, \sigma$ peut se réduire en une forme normale $e' \,/\, \sigma'$.

Procédons par induction sur la longueur de la réduction $e \,/\, \sigma \to^* e' \,/\, \sigma'$. Si cette dérivation est de longueur nulle, on en déduit que e est une réponse et que $[\![e]\!]_i = a_i$ ce qui donne le résultat voulu.

Si $e \,/\, \sigma \to e_0 \,/\, \sigma_0 \to^* e' \,/\, \sigma'$ alors $e_i \,/\, \sigma_i \to [\![e_0]\!]_i \,/\, [\![\sigma_0]\!]_i$ (lemme 3.3). Pour $i \in \{1, 2\}$, il existe un renommage des adresses mémoire ϕ_i tel que $[\![e_0]\!]_i \,/\, [\![\sigma_0]\!]_i \to^* \phi_i(a_i) \,/\, \phi_i(\tau_i)$ (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 $e_0 \,/\, \sigma_0 \to^* a \,/\, \tau$ 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.

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 4.1 (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 4.2 (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 4.3 (Sous-typage sur les constructions) Les cinq formes de types précisées dans l'axiome 4.1 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 4.4 (Transitivité) Si $\ell \leq \ell'$, $\ell' \lhd s$ et $s \leq s'$ alors $\ell \lhd s'$

Axiome 4.5 (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 $\ell_0 \leq \ell$.

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

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

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

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

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

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 $\forall \varepsilon \in \mathrm{dom}(r),\ \mathsf{Pre}^{-}s \leq r(\varepsilon) \Rightarrow \mathsf{Pre}^{\ell} s \leq r(\varepsilon)$.

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.

Valeurs :

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

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

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

Expressions :

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

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

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

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

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

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

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

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

Fig. 5: Le système SB(S)

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 $\langle \dots \mid \dots \rangle$ 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 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 $s' \leq s$.

> **Preuve** 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 $pc', \Gamma \vdash e : s\ [r]$.*

Preuve 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 $pc, \Gamma \vdash \llbracket e \rrbracket_i : s\ [r]$. Soit v une valeur du calcul partagé. Si $\Gamma \vdash v : s$ alors, pour $i \in \{1, 2\}$ on a $\Gamma \vdash \llbracket v \rrbracket_i : s$.*

Informellement, pour obtenir une dérivation correspondant à $\llbracket e \rrbracket_i$ 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 4.1 alors $\ell \in L$.

Preuve 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 4.4). □

▶ **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 $pc, \Gamma \vdash e[x \Leftarrow v] : s\ [r]$.*

Preuve 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 4.8 (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 $pc, \Gamma' \vdash e' \,/\, \sigma' : s \ [r]$.

Preuve 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 $e' = f[x \Leftarrow (\mathsf{exn}_\varepsilon\ v)]$. ($\Delta$) 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 $e = \langle v_1 \mid v_2 \rangle\ v$. ($\Delta$) 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 $\Gamma \vdash [\![v]\!]_i : s'$ ($i \in \{1, 2\}$, lemme 4.4).

En appliquant E-App, il vient $pc \sqcup \ell, \Gamma \vdash v_i\ [\![v]\!]_i : s\ [r]$. 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 $e' = \langle v_1 := [\![v]\!]_1 \mid v_2 := [\![v]\!]_2 \rangle$. ($\Delta$) 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 $\Gamma \vdash [\![v]\!]_i : s'$ ($i \in \{1, 2\}$, lemme 4.4).

Par E-Assign, on obtient $pc \sqcup \ell, \Gamma \vdash v_i := [\![v]\!]_i : \mathsf{unit}^\ell\ [r]$ 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 $e' = \langle !\ v_1 \mid !\ v_2 \rangle$. ($\Delta$) 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 $e' = \langle \mathsf{raise}\ v_1 \mid \mathsf{raise}\ v_2 \rangle$. ($\Delta$) 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 $e' = \langle [\![E]\!]_1[a_1] \mid [\![E]\!]_2[a_2] \rangle$.

Dans le cas où $E = \mathsf{bind}\ x = [\ ]\ \mathsf{in}\ e_0$, ($\Delta$) 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 $\mathsf{bind}\ x = a_i\ \mathsf{in}\ [\![e_0]\!]_i{\Downarrow}$ pour $i = 1$ ou 2) on obtient $pc, \Gamma \vdash e : s\ [r]$.

Dans le cas où $E = \mathsf{tryall}\ [\ ]\ \mathsf{with}\ x \succ e_0$, ($\Delta$) 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 $e_i \mathbin{/} \sigma \xrightarrow{i} e_i' \mathbin{/} \sigma'$ 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 $e_i \mathbin{/} \sigma \xrightarrow{i} e_i' \mathbin{/} \sigma'$ 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 $f \mathbin{/} \sigma \xrightarrow{i} f' \mathbin{/} \sigma'$.

Envisageons le cas où $E = \mathsf{bind}\ x = [\ ]\ \mathsf{in}\ e_0$. ($\Delta$) 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 $pc, \Gamma' \vdash f' \mathbin{/} \sigma' : s'\ [r_1]$. On a également $pc, \Gamma'[x \mapsto s'] \vdash e_0 : s\ [r_2]$ (lemme 4.3). On en déduit que $pc, \Gamma' \vdash e' \mathbin{/} \sigma' : s\ [r]$

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 $e \,/\, \emptyset \to^* v$ alors $[\![e]\!]_1 \,/\, \emptyset \to^* v$ et $[\![e]\!]_2 \,/\, \emptyset \to^* v$.*

> **Preuve** On a $pc, \emptyset \vdash_L e : \mathsf{int}^\ell\ [r]$ et $e \,/\, \emptyset \to^* v \,/\, \sigma$. Il existe donc un environnement Γ tel que $pc, \Gamma \vdash_L v : \mathsf{int}^\ell\ [r]$ (théorème 4.8). 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 $[\![e]\!]_i \,/\, \emptyset \to^* v$. □

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.

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

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 $\mathrm{ML}_{\mathrm{re}}$ 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 ($\preccurlyeq$)** *Soit L un ensemble d'étiquettes. On dit que e_1* interfère moins que *e_2 pour L et on note $e_1 \preccurlyeq_L e_2$ 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 5.2** *Soient e_1 et e_2 deux expressions du calcul étiqueté telles que $e_1 \preccurlyeq_L e_2$ 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 $\|e\|_i = \mathrm{strip}(e_i)$ et $pc, \Gamma \vdash_L e : s\ [r]$.*

5.3 Théorème de non-interférence

▶ **Théorème 5.3 (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 $e_1 \preccurlyeq_L e_2$ et $pc, \emptyset \vdash e_1 : \mathsf{int}^\ell\ [r]$ avec $\ell \notin L$.*
Si $\mathrm{strip}\, e_i \,/\, \emptyset \to^* v_i$ *(pour $i \in \{1, 2\}$) alors $v_1 = v_2$.*

> **Preuve** On déduit de $e_1 \preccurlyeq_L e_2$ 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 $\|e\|_i = \mathrm{strip}(e_i)$ et $pc, \Gamma \vdash_L e : s\ [r]$ (lemme 5.2).
> On déduit de $\mathrm{strip}\, e_i / \emptyset \to^* v_i$ et $\|e\|_i = \mathrm{strip}\, e_i$ qu'il existe une valeur v telle que $e/\emptyset \to^* v$ et $v_i = \phi_i(\|v\|_i)$ (pour $i \in \{1, 2\}$, lemme 3.4). Par le théorème 5.1, on a $\phi_1(\|v\|_1) = \phi_2(\|v\|_2) = k$. On en déduit que $v_1 = v_2 = k$. □

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 $\preccurlyeq_L$.
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, $\mathrm{ML}_{\mathrm{re}}$, 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.

try e_1 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.

tryall e_1 with $x \succ e_2$ 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* :

Preuve On conserve toutes les notations employées dans la preuve du théorème 4.8.

(try-finally) On pose $e = \mathsf{try}\ a\ \mathsf{finally}\ e_0$ et $e' = \mathsf{bind}\ \_ = e_0\ \mathsf{in}\ a$. ($\Delta$) 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 $e' = \mathsf{bind}\ \_ = e\ \mathsf{in}\ (\mathsf{raised}_\varepsilon\ v)$. ($\Delta$) 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 4.8 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 5.3) restent vrais.

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

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

Fig. 7: Règles E-TryReraise et E-TryFinally

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

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

$$\mathsf{proj}_j\ (v_1,\ v_2)\ /\ \sigma \xrightarrow{i} v_j\ /\ \sigma \qquad \text{(proj)}$$

$$\mathsf{proj}_j\ \langle v_1 \mid v_2\rangle\ /\ \sigma \xrightarrow{\varnothing} \langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2\rangle \qquad \text{(lift-proj)}$$

$$\mathsf{match}\ (\mathsf{inj}_j\ v)\ \mathsf{with}\ v_1\ v_2\ /\ \sigma \xrightarrow{i} v_j\ v\ /\ \sigma \qquad \text{(match)}$$

$$\mathsf{match}\ \langle v_1 \mid v_2\rangle\ \mathsf{with}\ w_1\ w_2\ /\ \sigma \xrightarrow{\varnothing} \langle \mathsf{match}\ v_i\ \mathsf{with}\ [\![w_1]\!]_i\ [\![w_2]\!]_i\rangle_i\ /\ \sigma \qquad \text{(lift-match)}$$

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 6.1 (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*.

Preuve On conserve toutes les notations employées dans la preuve du théorème 4.8.

(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 $e' = \langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2\rangle$. ($\Delta$) 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 $pc, \Gamma \vdash v_j\ v : s\ [r]$.

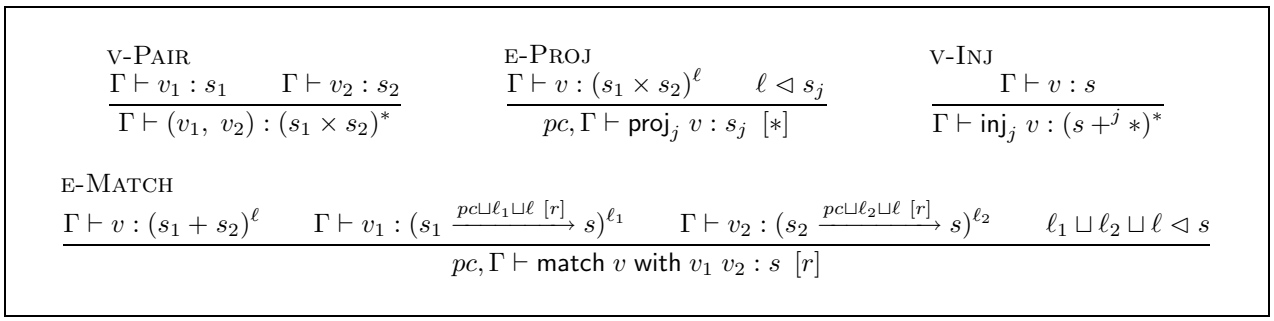

Fig. 8: Règles V-PAIR, E-PROJ, V-INJ et E-MATCH

(lift-match) On pose $e = \mathsf{match}\ \langle v_1 \mid v_2 \rangle\ \mathsf{with}\ w_1\ w_2$ et $e' = \langle \mathsf{match}\ w_i\ \mathsf{with}\ [\![v_1]\!]_i\ [\![v_2]\!]_i \rangle_i$. ($\Delta$) 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 $pc \sqcup \ell, \Gamma \vdash \mathsf{match}\ w_i\ \mathsf{with}\ [\![v_1]\!]_i\ [\![v_2]\!]_i : s\ \ [r]$. 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 : $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ devient $\mathsf{match}\ e_1\ \mathsf{with}\ (\lambda x.e_2)\ (\lambda x.\mathsf{inj}_2\ x)$ et $\mathsf{try}\ e_1\ \mathsf{with}\ x \succ e_2$ devient $\mathsf{match}\ e_1\ \mathsf{with}\ (\lambda x.\mathsf{inj}_1\ x)\ (\lambda x.e_2)$.
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 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 $\mathrm{ML_{re}}$ 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 $\mathrm{ML_{re}}$ 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.

$$f\ \bar{v}\ /\ \sigma \xrightarrow{i} [f](\bar{v})\ /\ \sigma \qquad \text{si } \bar{v} \text{ ne comporte pas de crochets} \qquad (\delta_f)$$

$$f\ \bar{v}\ /\ \sigma \xrightarrow{\varnothing} \langle [f](\llbracket \bar{v} \rrbracket_1) \mid [f](\llbracket \bar{v} \rrbracket_2) \rangle\ /\ \sigma \qquad \text{sinon} \qquad (\text{lift-}f)$$

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 6.2 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 6.3 La relation $\cdot \blacktriangleleft \cdot$ vérifie

$$\mathsf{int}^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \qquad \mathsf{unit}^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0$$

$$(r\ \mathsf{exn})^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \wedge (\forall \varepsilon \in \mathcal{E},\ r(\varepsilon) \blacktriangleleft \ell_0) \qquad (s_1 + s_2)^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \wedge s_1 \blacktriangleleft \ell_0 \wedge s_2 \blacktriangleleft \ell_0$$

$$(s_1 \times s_2)^\ell \blacktriangleleft \ell_0 \Rightarrow \ell \leq \ell_0 \wedge s_1 \blacktriangleleft \ell_0 \wedge s_2 \blacktriangleleft \ell_0 \qquad s \leq s' \wedge s' \blacktriangleleft \ell_0 \Rightarrow s \blacktriangleleft \ell_0$$

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 $\ell \in L$.*

Preuve 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.

Preuve On conserve toutes les notations employées dans la preuve du théorème 4.8.

(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 6.2, on en déduit que $pc, \Gamma \vdash [f](v) : s\ [r]$.

(δ-lift$_f$) On pose $e = f\,v$ et $e' = \langle [f](\llbracket \bar{v} \rrbracket_1) \mid [f](\llbracket \bar{v} \rrbracket_2) \rangle$. ($\Delta$) 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 $\Gamma \vdash \llbracket v \rrbracket_i : s'$ (pour $i \in \{1, 2\}$, lemme 4.4). On obtient $pc, \Gamma \vdash [f](\llbracket v \rrbracket_i) : s\ [r]$ (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). $\llbracket v \rrbracket_i$ étant une réponse, on en déduit que $pc \sqcup \ell, \Gamma \vdash [f](\llbracket v \rrbracket_i) : s\ [r]$. 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*.

Partie 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 :

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

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 :

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

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

Contraintes :

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

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

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 5.3 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 $e_i \,/\, \emptyset \rightarrow^* v_i$ (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 :

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

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

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

Expressions :

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

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

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

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

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

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

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

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

Fig. 11: Le système SHM(S)

Dans notre idée, ces instances doivent comporter les formes de types prévues par les axiomes 4.1, 4.2 et 6.1, 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 $\mathrm{ML}_{\mathrm{re}}$ 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.