

Université Paris 7 – Denis-Diderot
UFR d'Informatique

THÈSE
pour l'obtention du diplôme de
Docteur de l'Université Paris 7, spécialité informatique

UNE APPROCHE DE LA DÉTECTION STATIQUE D'EXCEPTIONS NON RATTRAPÉES EN APPEL PAR NOM

présentée et soutenue publiquement par

Sylvain LEBRESNE

le 5 Décembre 2008

devant le jury composé de

M. Thorsten	ALTENKIRCH	Rapporteur
M. Gilles	BARTHE	Rapporteur
M. Pierre-Louis	CURIEN	Directeur de thèse
M. Hugo	HERBELIN	Directeur de thèse
M. Alexandre	MIQUEL	
M. François	POTTIER	
M. Christophe	RAFFALLI	

Remerciements

À tout seigneur, tout honneur, je souhaiterais commencer par remercier Alexandre MIQUEL, Hugo HERBELIN et Pierre Louis CURIEN. Alexandre tout d'abord qui bien qu'il n'en soit pas crédité officiellement au générique a été un des encadrants de ces travaux. Son enthousiasme, sa sympathie et sa pédagogie sont autant de qualités qui en font un excellent encadrant. Merci aussi à Hugo pour sa disponibilité, pour sa gentillesse et pour avoir toujours soutenu et aiguillé mes travaux. Enfin merci à Pierre-Louis d'avoir accepté la direction officielle de cette thèse, mais surtout pour être en grande partie responsable du fait que PPS soit un endroit où il fait bon vivre. Je garderai aussi longtemps le souvenir ému de notre entraînement de Hip-Hop raté.

Un très grand merci également à Thorsten ALTENKIRCH et Gilles BARTHE qui m'ont fait le grand honneur d'accepter de rapporter cette thèse.

François POTTIER et Christophe RAFFALLI ont très gentiment acceptés de faire partie de mon jury, je les remercie et leur en est particulièrement reconnaissant. Merci à François aussi pour ses remarques sur le manuscrit.

Cette thèse aura duré quatre ans et j'y aurai fréquenté deux laboratoires, PPS et le LIX. Je voudrais ici remercier toutes les personnes que j'ai eu la chance d'y côtoyer et qui par leurs qualités scientifiques et personnelles ont contribué à rendre ces années autant agréables qu'enrichissantes. Merci notamment à tous mes collègues thésards pour leur grande sympathie et leur bonne humeur générale. Je voudrais particulièrement en remercier quelques uns des plus proches, qui sont devenus de réels amis : Fab (et Raph bien sûr, ex-PPSiène par alliance), Michel (même s'il est rarement disponible), Christine (qui a plus d'une fois tenté de me rendre moins moche, avec un certain succès d'ailleurs), Séverine (et son pull de labo en laine) et Sam et Nico (mes compères d'exil de fin de thèse). Sans oublier Marie qui bien qu'habitant en territoire ennemi est bien installée dans le camp des amis.

Merci aussi à Odile, Michèle, Catherine et Lydie qui non contentes d'être des secrétaires efficaces, sont également fort sympathiques.

Ces années de thèse n'auraient pas été les mêmes s'il n'y avait pas eu la « maison des thesards » et tous les gens qui y sont passés. Qu'ils en soient tous ici remerciés. Et parmi eux, il en est que je ne veux pas ne pas citer nommément. Alors pour l'amitié, les gueuletons et tant de moments mémorables, je souhaite remercier Guillaume (et sa « blague » de l'escalier), mój przyjaciel Ignacy (colocataire *honoris causa*), Bertrand (pour notre temps à RE4, et bien d'autres choses) et surtout, surtout, mon vieux binôme Kim, dont la taille est la meilleure des arnaques, tant c'est un grand personnage.

Je n'oublie pas non plus les autres amis qui ont également beaucoup contribué à cette thèse à leur façon. Merci d'abord à mes amis dits « de Malvoisine » (parfois également appelé « mes vrais amis », il doit y avoir une raison), puis merci à mes amis « de l'escalade » pour tant de mémorables week-ends et anecdotes ressassées jusqu'à épuisement.

Enfin, *last but not least*, je voudrais remercier ma famille, mes proches, pour leurs soutiens sans faille même quand il ne savait pas trop ce que je faisais.

Table des matières

Table des matières	v
Table des figures	ix
1 Introduction	1
1.1 Un mécanisme de programmation : les exceptions	1
1.2 Les langages de programmation fonctionnels	2
1.3 La théorie des types	4
1.4 Plan de la thèse	5
2 État de l’art	7
2.1 Positionnement	7
2.2 Typage des exceptions en appel par valeur	8
2.3 Exceptions et appel par nom	10
2.3.1 Coder les exceptions dans le calcul	10
2.3.2 Mécanisme d’exceptions primitif	12
2.4 Opérateurs de contrôle	14
2.5 Les exceptions en théorie des types	14
3 Exceptions en appel par nom	17
3.1 Les termes du calcul	17
3.2 Évaluation	19
3.2.1 La réduction	19
3.2.2 Confluence	21
3.2.3 Réduction vers une valeur	22
3.2.4 La réduction de tête faible	22
3.2.5 Standardisation	24
3.3 Exemples	30
4 Le système de types	33
4.1 Principe	33
4.1.1 Quel système de types ?	33
4.1.2 Trois niveaux de corruption	33
4.1.3 Des ensembles d’exceptions	37
4.1.4 Quel type pour rec ?	38
4.2 Typage et sous-typage	39

4.2.1	Syntaxe des types	39
4.2.2	Jugement de typage	39
4.2.3	Jugement de sous-typage	42
4.3	Exemples	45
4.4	Propriétés	49
4.4.1	Propriétés liminaires	49
4.4.2	Préservation du typage par réduction	68
4.4.3	Préservation du typage par η -réduction	70
4.4.4	Progression du calcul	71
4.4.5	Corruption des termes	73
4.4.6	Sûreté vis-à-vis des exceptions	74
5	Sémantique	77
5.1	Corruption et flèche	77
5.2	Un premier modèle de réalisabilité	78
5.2.1	Définition du modèle	78
5.2.2	Propriétés de l'interprétation	80
5.2.3	Correction	82
5.2.4	Propriétés dérivées	85
6	Extensions	87
6.1	Point fixe et « match »	87
6.1.1	Présentation	87
6.1.2	Exemples	89
6.1.3	Propriétés	90
6.2	Types de données	95
6.2.1	Les booléens	95
6.2.2	Le type <code>unit</code>	96
6.2.3	Le type produit	96
6.2.4	Le type somme	97
6.2.5	Les listes	98
6.2.6	Exemples d'utilisation	100
6.2.7	Propriétés	101
6.3	Des entiers courageux	105
6.4	Ajouter des arguments aux exceptions ?	106
7	Exceptions et second-ordre	109
7.1	Le λ -calcul du second-ordre avec sous-typage	109
7.2	Le système F_ϵ	111
7.2.1	Termes : syntaxe et réductions	111
7.2.2	Le système de types	111
7.2.3	Propriétés	117
7.3	Un Modèle de réalisabilité pour F_ϵ	120
7.3.1	L'orthogonalité	121
7.3.2	Définitions préliminaires	122
7.3.3	Définition et propriétés de l'interprétation	126
7.3.4	Correction du sous-typage vis-à-vis du modèle	132
7.3.5	Correction du typage vis-à-vis du modèle	136

7.3.6	Propriétés du calcul dérivées du modèle	138
7.4	Les codages du second-ordre	139
8	Perspectives	141
8.1	Types dépendants	141
8.2	Inférence de type	142
8.3	Intégration dans Haskell	142
8.4	Arguments d'exceptions	142
	Bibliographie	143

Table des figures

3.1	Règles de réduction de λ_ϵ .	20
3.2	Réduction parallèle de λ_ϵ .	21
3.3	Règles de réduction pour la réduction de tête faible dans λ_ϵ .	23
3.4	La réduction interne	24
4.1	Règles de typage du langage λ_ϵ .	40
4.2	Règles de sous-typage du langage λ_ϵ .	42
4.3	Schéma de dépendance pour la preuve de la préservation du typage par réduction	51
4.4	La simplification de type	52
4.5	Paires critiques de la simplification de type	54
4.6	Paires critiques de la simplification de type (suite)	55
4.7	La relation de corruption	73
5.1	Schéma de dépendance pour la preuve de correction du typage vis-à-vis de l'interprétation	80
6.1	Rappel des dépendances pour la preuve de la préservation du typage par réduction de λ_ϵ .	91
6.2	Rappel des dépendances pour la preuve de correction du typage de λ_ϵ vis-à-vis de l'interprétation	102
7.1	Rappel des termes et réductions de λ_ϵ réutilisées par F_ϵ .	112
7.2	Règles de typage du langage F_ϵ .	114
7.3	Règles de sous-typage du langage F_ϵ .	116
7.4	Rappel de la figure 4.7 définissant la relation de corruption	120
7.5	Règles de réduction du <i>daimon</i> .	122

1

Introduction

JE ne veux pas vous faire peur, mais les programmes informatiques sont partout. Et comme dirait ma grand-mère, « Vos machins informatiques là, ça marche jamais ! ». Ça n'est pas complètement faux. La bonne question est alors comment changer cela, comment éviter les « bugs » dans les programmes ? Un vaste pan de la recherche en informatique a cet objectif : rendre plus sûrs les programmes. L'étude et l'amélioration des langages de programmation est une façon d'y parvenir et cette thèse s'inscrit dans cet effort. Nous y développons un système de types pour la détection d'exceptions non rattrapées dans les langages fonctionnels en appel par nom et en analysons les principales propriétés. Explication de texte. . .

1.1 Un mécanisme de programmation : les exceptions

Le mécanisme des exceptions est utilisé par un grand nombre de langages de programmation. Son but est de faciliter la gestion des cas qui ne rentrent pas dans le cadre du comportement ordinaire d'un programme, des cas « exceptionnels ».

Il est commun que l'endroit où l'on détecte un tel cas exceptionnel ne soit pas le meilleur endroit pour le traiter. Interviennent alors les exceptions. Le mécanisme des exceptions consiste usuellement en deux constructions :

- La première permet de signaler que l'on a rencontré un cas exceptionnel que l'on ne désire pas traiter directement. On dit alors qu'on lève une exception et nous noterons cette opération *raise* dans la suite de cette thèse (certains langages nomment cette opération *throw*). Si lors de son exécution le programme évalue une opération *raise*, le comportement normal du programme, son flot de contrôle, est remplacé par la propagation de l'exception ;
- La seconde construction délimite une partie du programme et permet de rattraper une exception si celle-ci est propagée par la partie délimitée. Nous nommerons cette opération *try/with* (dans la tradition de OCaml [L⁺07]) mais certains langages utilisent la combinaison de mots clefs *try/catch* (par exemple Java [GJS96]) ou encore *handle* (par exemple SML [MTHM97]) pour cette opération. Si la partie de code délimitée par le *try* ne lève pas d'exception, cette construction est invisible pour le programme. En revanche, si le code propage une exception, celle-ci est rattrapée (on dit aussi capturée) par la construction qui passe alors le contrôle du programme à un gestionnaire spécifique.

La force des exceptions est de séparer la détection d’une erreur de sa gestion, et ce de façon transparente pour le programmeur. Cependant, en séparant ces deux opérations, apparaît la possibilité de signaler une exception sans jamais la gérer. Si l’exécution d’un programme lève une exception dans une partie de code qui n’est nulle part délimitée par une construction `try`, l’exception se propage jusqu’à la racine du programme qui échoue alors. L’exception est alors dite non rattrapée. Ce problème fait l’objet de cette thèse.

1.2 Les langages de programmation fonctionnels

Nous nous intéressons aux exceptions dans le cadre des langages de programmation fonctionnels. La programmation fonctionnelle est un paradigme de programmation où les fonctions sont des objets de première classe. Le modèle de ces langages est le λ -calcul [Bar84], et tous les langages fonctionnels modernes peuvent être vus comme des extensions du λ -calcul. Le λ -calcul met l’accent sur la notion de fonction et les deux principales constructions y sont la création de fonction (ou λ -abstraction), notée $\lambda x. t$, et l’application d’une fonction t à un argument u , notée $t u$. La principale règle de calcul est la β -réduction (notée $\rightarrow_\beta$) dont on distingue deux versions : l’appel par nom (ou *Call-By-Name*) et l’appel par valeur (ou *Call-By-Value*) :

Appel par nom		Appel par valeur
$(\lambda x. t) u \rightarrow_\beta t\{x := u\}$		$(\lambda x. t) v \rightarrow_\beta t\{x := v\}$ (si v est une valeur)

Dans les deux cas, la règle de β -réduction réduit l’application d’une fonction au corps de cette dernière où l’argument formel est remplacé par l’argument réel. Mais dans le cas de l’appel par valeur, la réduction n’est possible que si l’argument est une valeur, c’est-à-dire si il a été préalablement évalué. Ces deux versions de la β -réduction représentent deux stratégies d’évaluation différentes. En présence de non-terminaison ou d’effets (lecture/écriture, accès mémoire, ...), ces deux stratégies diffèrent fondamentalement et donnent lieu à deux grandes familles de langages fonctionnels : les langages en appel par valeur, ou stricts (ML [LW99], Lisp [Ste90], ...) et les langages en appel par nom¹ (Haskell [PJe⁺03], Miranda [Tur85], ...).

Exceptions et appel par valeur. Les langages fonctionnels en appel par valeur, aussi appelé langages fonctionnels stricts, s’accommodent parfaitement du mécanisme des exceptions. En effet, la propagation des exceptions est directement liée à la notion de flot de contrôle d’un programme, et la notion de flot de contrôle des langages fonctionnels stricts correspond à celle des langages de programmation impératifs (qui sont des langages stricts eux aussi).

Dans ces langages fonctionnels, où l’exécution d’un programme s’exprime en termes de règles de réduction, la propagation d’une exception s’exprime par les deux règles suivantes² :

$$\begin{array}{ll} (\text{raise } e) u & \rightarrow \text{raise } e & (a) \\ t (\text{raise } e) & \rightarrow \text{raise } e & (b) \end{array}$$

¹Dans la suite nous confondrons souvent, et sans perte de généralité pour notre propos, la notion théorique d’appel par nom et celle d’évaluation paresseuse.

²On ne présente ici que la propagation des exceptions vis-à-vis de l’application. Les langages réalistes possèdent d’autres constructions (filtrage par motif, `let... in, ...`) qui ont également des règles pour propager les exceptions. De plus, pour la plupart des langages fonctionnels strict, le terme t dans la règle (b) doit généralement être sous forme de valeur pour que la règle s’applique.

Dans ces règles, e représente une exception, la définition exacte que l'on donne à cette notion n'ayant pas d'importance ici. Ce peut en particulier être un simple nom, ou comme dans le cas de ML, un objet d'une structure de données algébrique.

Typage. Dans cette thèse, nous allons nous intéresser à une certaine classe de langages fonctionnels, ceux qui sont statiquement typés [Pie02]. Une des forces de ce typage est de permettre la détection statique (*i.e.*, à la compilation) d'un certain nombre d'erreurs de programmation, comme par exemple l'addition d'un entier à une chaîne de caractères, opération qui est certaine d'échouer à l'exécution (sauf bien sûr si le langage surcharge l'addition de façon à lui donner un sens dans ce cas précis).

Vis-à-vis du typage, l'opération consistant à lever une exception (l'expression *raise e*) est un peu particulière puisqu'elle doit pouvoir être utilisée n'importe où, et donc en particulier à la place d'un entier, d'un caractère, d'une fonction, ... La solution généralement adoptée (au moins par tous les dialectes de la famille ML) est de donner tous les types à cette opération³. Autrement dit, pour tout type A , le terme *raise e* est de type A .

Cette solution n'est pas anodine, elle est rendue *obligatoire* par un certain nombre de choix de design de langage. Ces choix peuvent être exprimés par les quatre conditions suivantes :

1. Le calcul possède la règle (*b*) ci-dessus ;
2. Les règles de typage de l'abstraction et de l'application y sont les règles standards ;
3. L'expression *raise e* a au moins un type, quel qu'il soit (nommé T par la suite) ;
4. Le calcul vérifie la propriété de préservation du typage par réduction.

Si un langage réunit ces quatre conditions, alors pour tout type habité A , l'opération *raise e* a le type A .

Pour montrer ce résultat, considérons un type habité A quelconque et un terme a de type A , et considérons le terme t suivant :

$$t = (\lambda x. a) (\text{raise } e)$$

La fonction $\lambda x. a$ a type $T \rightarrow A$ et donc par typage de l'application, t a le type A (conditions 2 et 3). Mais t se réduit sur *raise e* avec la règle (*b*) (condition 1) et donc par préservation du typage par réduction (condition 4), *raise e* a le type A .

Cette solution qui donne au terme *raise e* tous les types a un inconvénient majeur : elle affaiblit la sûreté qu'offre le typage. Un programme de type `int` ne calcule plus forcément un entier, il peut aussi lever une exception. Il existe des solutions à ce problème dans la littérature que nous décrirons dans la section 2.2.

Exceptions et appel par nom. Le cas des exceptions dans un langage en appel par nom est plus compliqué, justement parce que la propagation des exceptions repose sur la notion de flot de contrôle. En effet, en appel par nom, on ne peut être sûr qu'une expression est évaluée que lorsque la valeur qu'elle calcule est utilisée. En conséquence, le flot de contrôle d'un programme n'est pas prédictible par avance. Plus précisément, la règle (*b*) de propagation des exceptions en appel par valeur présentée ci-dessus (la règle $t (\text{raise } e) \rightarrow \text{raise } e$) n'a pas de sens en appel par nom puisqu'elle n'est applicable que dans un calcul où l'argument d'une fonction est évalué.

³On peut d'ailleurs noter que cette question se résout immédiatement dans les langages impératifs puisque les exceptions ne s'y propagent qu'à travers des instructions qui ont toutes le même type.

Une solution, introduite par S. Peyton Jones, A. Reid, F. Henderson, T. Hoare et S. Marlow [PJRH⁺99] pour le langage Haskell, consiste simplement à abandonner cette règle de réduction incompatible avec l’appel par nom. La seule règle de propagation des exceptions vis-à-vis de l’application étant alors la règle (a) (la règle (raise e) $u \rightarrow \text{raise } e$). Autrement dit, une exception n’est propagée que lorsqu’elle est utilisée. Ce changement impose de voir les exceptions comme liées aux valeurs du programme et non plus au flot de contrôle. Dans la suite, nous nous référerons à ce mécanisme d’exceptions sous le nom d’exceptions en appel par nom.

Notons que ce mécanisme d’exceptions en appel par nom diffère du mécanisme d’exceptions traditionnel indépendamment de la stratégie adoptée pour la β -réduction. Rien n’empêche en effet de l’utiliser en appel par valeur, mais là où avec le mécanisme d’exceptions traditionnel le programme $(\lambda x.0) (\text{raise } e)$ lève l’exception $\text{raise } e$, dans le cadre du mécanisme d’exceptions en appel par nom, ce même programme ne lève pas l’exception e et se réduit sur 0. Avec le mécanisme d’exceptions en appel par nom, les exceptions ne sont plus un effet du calcul.

Qu’en est-il du typage pour ce mécanisme d’exceptions en appel par nom ? La problématique est la même que pour le mécanisme traditionnel, à savoir que l’expression $\text{raise } e$ doit pouvoir être utilisée à la place de n’importe quelle expression. La même solution est alors envisageable : donner à cette expression tous les types (c’est d’ailleurs la solution adoptée dans [PJRH⁺99]), mais elle possède les mêmes inconvénients.

La principale contribution de cette thèse est de proposer un système de types pour les exceptions en appel par nom où le type d’une expression mentionne les exceptions que cette dernière peut lever et où un programme ayant le type des entiers ne peut par conséquent que s’évaluer sur un entier.

1.3 La théorie des types

Le λ -calcul et ses versions typées sont des modèles intéressants pour les langages de programmation, mais présentent un intérêt supplémentaire à la lumière de l’isomorphisme de Curry-Howard, qui met en relation les preuves mathématiques avec les programmes et les propositions logiques avec les types. Les programmes écrits en λ -calcul sont des preuves mathématiques, leurs types étant les propositions dont ils sont la preuve. Dans ce cadre, l’étude de systèmes de type expressifs pour le λ -calcul permet d’obtenir des systèmes qui autorisent à la fois l’écriture de preuves mathématiques et celle de programmes qui respectent des spécifications de haut niveau (ces deux activités étant équivalentes). Ces langages constituent l’étude de la théorie des types.

Un des points de départ des travaux de cette thèse est l’absence d’un mécanisme d’exceptions dans l’assistant d’aide à la preuve Coq [Tea06], basé sur le Calcul des Constructions Inductives (CCI) [Wer94]. Dans un langage de la théorie comme CCI, un mécanisme d’exceptions doit faire face à trois conditions :

1. Le Calcul des Constructions Inductives est pur, il ne considère pas les effets. En conséquence, aucune stratégie particulière pour la β -réduction n’y est fixée. D’un point de vue théorique, la règle de β -réduction y est donc exprimée sous la forme plus générale de l’appel par nom. Ainsi, y forcer l’appel par valeur constituerait une perte de généralité qui obligerait à revoir la théorie sous-jacente. Pour ces raisons, je pense que le mécanisme des exceptions en appel par nom est un meilleur candidat que le mécanisme d’exceptions traditionnel.

2. Dans les langages de programmation, le choix de typer l'opération qui lève une exception (*i.e.*, le terme raise e) avec tous les types implique une perte de sûreté du typage. En théorie des types, ce choix fait perdre la consistance logique du système. En effet, avoir une opération qui a tous les types signifierait avoir une preuve de n'importe quelle proposition. Plus généralement, on attend du typage d'un langage de la théorie des types qu'il capture plus de propriétés qu'un langage de programmation traditionnel. En particulier, les exceptions levées par une expression doivent se refléter dans le type de cette expression.

Le système de type présenté dans cette thèse répond à cette question dans l'optique du choix des exceptions en appel par nom (voir point précédent).

3. Les langages de la théorie des types manipulent des types d'ordre supérieur et le typage des exceptions doit donc être compatible avec cet ordre supérieur. Le chapitre 7 de ce document montre la compatibilité du typage que nous proposons avec le second-ordre et doit être vu comme une première étape vers l'objectif du calcul des constructions.

1.4 Plan de la thèse

Dans cette thèse, nous allons présenter principalement deux calculs : le premier, λ_ε , étend le λ -calcul simplement typé. Son étude fera l'objet des chapitres 3 à 6. Le second, F_ε , est une extension de λ_ε au second ordre et sera l'objet du chapitre 7. Dans son ensemble, le manuscrit est organisé de la façon suivante :

État de l'art (Chapitre 2) Nous présentons les travaux existants en relation avec cette thèse et les comparons avec notre approche.

Exceptions en appel par nom (Chapitre 3) Nous présentons la syntaxe des termes de λ_ε et leur sémantique opérationnelle. On prouve également la confluence de la réduction ainsi qu'une propriété de standardisation de cette réduction.

Le système de types (Chapitre 4) Nous présentons le système de types du calcul λ_ε et les deux notions sur lesquelles il se base : l'union et la corruption. Nous montrons alors les propriétés du calcul (préservation du typage par réduction, progression, ...).

Sémantique (Chapitre 5) Nous développons un modèle de réalisabilité du calcul λ_ε . On déduit de ce modèle une propriété de normalisation en forme normale de tête faible ainsi qu'une propriété de sûreté du typage pour les types de base.

Extensions (Chapitre 6) Nous présentons plusieurs extensions au calcul λ_ε : ajout d'un point fixe, ajout de types de données primitifs (produits, listes, ...) et encodage d'entiers stricts.

Exceptions et second-ordre (Chapitre 7) Nous présentons le calcul F_ε qui est une extension du calcul λ_ε avec la quantification de type du second-ordre. Nous développons

également un modèle de réalisabilité de ce nouveau calcul en utilisant des techniques d’orthogonalité. Nous y discutons également les encodages du second-ordre en présence d’exceptions. Ce chapitre constitue essentiellement une version étendue de l’article [Leb08] : *A System F with call-by-name exceptions*.

Perspectives (Chapitre 8) Nous décrivons les principales perspectives pour la continuation de ces travaux.

Nous présentons les travaux en rapport avec nos objectifs et les comparons à notre approche. Nous commençons par expliciter le positionnement de ces travaux (section 2.1) avant de rappeler les travaux sur l'analyse statique d'exceptions non rattrapées en appel par valeur dans la section 2.2. Nous nous intéressons ensuite au cas des exceptions en appel par nom en section 2.3 avant d'évoquer dans la section 2.4 les opérateurs de contrôle dont les exceptions sont souvent considérées comme un cas particulier. Un de nos objectifs est de développer un typage des exceptions compatible avec les types d'ordres supérieurs et nous évoquerons en section 2.5 les précédents travaux liés aux exceptions dans ce domaine.

2.1 Positionnement

Pour les langages de programmation fonctionnels, on peut distinguer deux mécanismes d'exceptions distincts :

1. Le mécanisme d'exception traditionnel où dans une application, une exception est propagée que ce soit le terme à gauche ou à droite de l'application qui la lève. En particulier, on a la réduction

$$(\lambda x. t) (\text{raise } e) \rightarrow \text{raise } e.$$

2. Le mécanisme d'exception en appel par nom¹ où une application ne propage que les exceptions levées par le terme de gauche. On a en particulier la réduction

$$(\lambda x. t) (\text{raise } e) \rightarrow t\{x := \text{raise } e\}.$$

Les langages de programmation fonctionnels où la β -réduction est en appel par valeur utilisent tous le mécanisme d'exception traditionnel (lorsqu'ils ont un mécanisme d'exceptions). Ce mécanisme n'est toutefois pas compatible avec la β -réduction en appel par nom et donc, dans les langages fonctionnels avec la β -réduction en appel par nom, le mécanisme d'exceptions en appel par nom est utilisé. Notons cependant qu'en principe, rien n'empêche d'utiliser les exceptions en appel par nom dans un langage où la β -réduction est en appel par valeur.

¹Ce vocabulaire est spécifique à cette thèse, nous ne connaissons pas de dénomination standard pour la notion exprimée ici.

Indépendamment du choix de mécanisme, on peut distinguer deux grands choix pour le typage des exceptions :

1. Les types ne tiennent pas compte des exceptions. La construction `raise e` a alors n’importe quel type ;
2. Le typage d’un terme rend compte des exceptions que ce dernier peut lever à l’exécution.

Dans le cas où les types rendent compte des exceptions levées par les termes, nous distinguons un dernier choix que l’on peut résumer à la question pragmatique suivante :

L’application du langage permet-elle d’appliquer une fonction f ayant le type $A \rightarrow B$, où A est un type *ne mentionnant pas* l’exception e , à un argument qui peut lever cette exception ? Autrement dit, le terme f (`raise  $e$`) est-il bien typé ?

Une réponse négative impose de disposer d’une opération spécifique pour réaliser cette application. Un exemple d’une telle opération est l’opérateur `>>=` d’Haskell dans le cadre d’une monade d’exception et nous reviendrons en section 2.3.1 sur les inconvénients d’une telle approche. Ce choix distingue une propagation implicite des exceptions (où on autorise f (`raise  $e$`)) d’une propagation explicite (où une opération doit être explicitement utilisée).

Le reste de ce chapitre discute les différents choix que nous venons de présenter. Le tableau qui suit situe, vis-à-vis de ces choix, les travaux que nous allons présenter et les travaux de cette thèse :

Exns	Les types mentionnent les exceptions ?		
	Non	Oui	
		Propagation explicite	Propagation implicite
CBN	<i>Imprecise exceptions</i> (section 2.3.2)	Type <code>option</code> , monade d’exception Haskell (section 2.3.1), EX_2 de Mounier et David (section 2.5)	Les systèmes présentés dans cette thèse : λ_ϵ et F_ϵ
CBV	OCaml [LW99], SML [MTHM97]		Leroy et Pessaux (Section 2.2)

Dans ce tableau, la ligne CBN désigne le mécanisme d’exceptions en appel par nom, alors que la ligne CBV désigne le mécanisme traditionnel.

2.2 Typage des exceptions en appel par valeur

Notre objectif est de proposer un typage précis des exceptions en appel par nom, mais nous avons vu que les exceptions en appel par nom s’obtiennent à partir des exceptions traditionnelles en abandonnant la règle de propagation des exceptions qui n’est pas compatible avec l’appel par nom (la règle (b) de la section 1.2) et en gardant l’autre qui elle est compatible à la fois avec l’appel par valeur et l’appel par nom (la règle (a) de la section 1.2). On peut donc, et à juste titre, se demander si les solutions existantes pour le typage des exceptions en appel par valeur ne peuvent pas être adaptées au cas des exceptions en appel par nom.

Dans les langages fonctionnels en appel par valeur statiquement typé, un programme qui a le type des entiers peut lever une exception et ainsi, le typage ne donne aucune information quant au fait qu'un terme peut lever ou non des exceptions à l'exécution. Une première solution pour régler ce problème, indépendante du typage, consiste à appliquer des méthodes d'analyse de flot de contrôle pour détecter les exceptions non rattrapées d'un programme [YR97, FFCA98]. Une seconde méthode est la modification du système de type afin de prendre en compte ces exceptions non rattrapées. Pour l'appel par valeur, cette méthode repose généralement sur la notion de *système d'effets*. Parmi ces travaux, on peut noter ceux de J.C. Guzmán et A. Suárez [GS94] pour ML et de X. Leroy et F. Pessaux [LP99, Pes00] plus spécifiquement pour le langage OCaml.

Le principe général d'un système d'effets est d'annoter le type d'une fonction avec les effets que celle-ci peut produire lors de son évaluation et de modifier les jugements de typage afin de tenir compte de ces effets. Avec un tel système, le type d'une fonction est $A \xrightarrow{\varphi} B$ où φ représente l'effet que la fonction peut produire lorsqu'elle est appliquée et le jugement de typage est

$$\Gamma \vdash t : A / \varphi$$

qui se lit « dans l'environnement de typage Γ , le terme t a le type A et peut produire l'effet φ lors de son évaluation ». Dans le cas de l'analyse d'exceptions, les effets sont des ensembles d'exceptions pouvant être levées. Le terme `raise  $e$` se type alors avec la règle

$$\frac{}{\Gamma \vdash \text{raise } e : A / \{e\}}$$

Ce terme est typable par n'importe quel type, mais est caractérisé par le fait qu'il produit l'effet e . La règle de typage de la construction `try` est

$$\frac{\Gamma \vdash t : A / \varphi \quad \Gamma \vdash u : A / \varphi'}{\Gamma \vdash \text{try } t \text{ with } e \mapsto u : A / (\varphi - \{e\}) \cup \varphi'}$$

où si t lève l'exception e , elle est capturée par la construction `try` comme l'indique l'effet final. Enfin, les règles de typage pour l'abstraction et l'application sont :

$$\frac{\Gamma, x : A \vdash t : B / \varphi}{\Gamma \vdash \lambda x. t : (A \xrightarrow{\varphi} B) / \varnothing} \qquad \frac{\Gamma \vdash t : (A \xrightarrow{\varphi} B) / \varnothing \quad \Gamma \vdash u : A / \varphi'}{\Gamma \vdash t u : B / \varphi \cup \varphi'}$$

Ces systèmes d'effets ne sont toutefois pas adaptés au cas des exceptions en appel par nom, et pour une bonne raison : les exceptions en appel par nom ne sont pas des effets mais de simples valeurs du calcul. Pour s'en convaincre, considérons dans un tel système le typage du terme $(\lambda x. \lambda y. x) (\text{raise } e)$ dont la dérivation de typage est :

$$\frac{\begin{array}{c} \vdots \\ \Gamma \vdash \lambda x. \lambda y. x : A \xrightarrow{\varnothing} (B \xrightarrow{\varnothing} A) / \varnothing \end{array} \quad \begin{array}{c} \vdots \\ \Gamma \vdash \text{raise } e : A / \{e\} \end{array}}{\Gamma \vdash (\lambda x. \lambda y. x) (\text{raise } e) : (B \xrightarrow{\varnothing} A) / \{e\}}$$

En appel par valeur et avec le mécanisme d'exceptions traditionnel, ce terme se réduit sur `raise  $e$` qui effectivement a le type $B \xrightarrow{\varnothing} A$ et produit l'effet $\{e\}$. En revanche, dans le cas

des exceptions en appel par nom, ce même terme se réduit sur $\lambda y. \text{raise } e$. Or ce terme ne lève aucune exception (il n’a pas d’effets) et, avec les notations des systèmes d’effets, a le type $B \xrightarrow{\{e\}} A$. Un système de types pour les exceptions en appel par nom doit être capable de rendre compte d’un tel phénomène où une fonction enfouit une exception dans son corps, ce dont les systèmes d’effets ne sont pas capables. Nous reviendrons sur ce phénomène d’enfouissement, spécifique aux exceptions en appel par nom, qui motivera l’introduction de la construction de type au cœur des travaux de cette thèse, la *corruption*.

2.3 Exceptions et appel par nom

Parce que le mécanisme traditionnel des exceptions est lié au flot de contrôle d’un programme et que les langages en appel par nom n’ont pas une notion de flot de contrôle aisément prédictible statiquement, l’ajout d’un tel mécanisme est compromis. En appel par nom, la seule façon productive de raisonner sur une expression est de considérer la *valeur qu’elle calcule* et non la *façon dont cette valeur est calculée*. Les exceptions doivent donc y être repensées en terme de valeurs et non de flot de contrôle. Dans cette section, nous allons voir les deux principales solutions pour la gestion des cas exceptionnels en appel par nom en considérant le cas du langage paresseux Haskell. La première solution est le codage des exceptions dans les valeurs du langage. La seconde est le mécanisme primitif des exceptions imprécises introduit dans [PJRH⁺99].

2.3.1 Coder les exceptions dans le calcul

L’idée de coder les exceptions dans les valeurs du langage n’est pas nouvelle [Wad85, Spi90]. Le principe est de définir un type des valeurs potentiellement exceptionnelles, par exemple le type `ExVal` suivant :

```
data ExVal a = OK a
              | Exn Exception
data Exception = DivideByZero
               | UserError
               ...
```

où le type `Exception` permet de définir différentes sortes d’exceptions. Ce type `ExVal` permet alors de définir une fonction renvoyant une valeur potentiellement exceptionnelle. Par exemple,

```
pred :: Integer -> ExVal Integer
pred n = if n > 0 then OK (n-1) else Exn UserError
```

Par ailleurs, `ExVal` forme une monade [Mog91, Wad90], permettant ainsi d’utiliser toute la machinerie monadique pour le manipuler. Cette approche possède plusieurs avantages, parmi lesquels on citera :

- Qui dit codage, dit non extension du langage. Toutes les propriétés du langage sont conservées ;
- Le type d’une expression indique si cette expression peut « lever une exception ». Notons cependant que le type n’indique que si l’expression est potentiellement exceptionnelle et non pas quelles exceptions en particulier peuvent être renvoyées. Cependant, au moins dans le cas où `ExVal` est considéré comme une monade, P. Wadler

et P. Thiemann [WT03] ont montré que l'on pouvait greffer par-dessus un système d'effets, permettant alors de résoudre ce problème.

Cependant, cette approche possède également un certain nombre d'inconvénients :

- En codant les exceptions, on en perd l'une des principales caractéristiques, la propagation implicite. Une exception, ou plus exactement une valeur exceptionnelle doit être explicitement propagée par le programmeur, polluant ainsi le code avec du « bruit » supplémentaire. Prenons l'exemple de l'application d'une fonction f sur les entiers ($f :: \text{Integer} \rightarrow \text{Integer}$) à un argument x qui est potentiellement exceptionnel ($x :: \text{ExVal Integer}$). Dans le cas où ExVal est un simple type algébrique, on doit écrire :

```
case x of
  Ok a -> Ok (f a)
  Exn e -> Exn e
```

Si ExVal est une monade d'exception, le traitement de l'exception est automatisé mais on doit tout de même écrire :

```
x >>= (\a -> return (f a))
```

ou, en utilisant la *do*-notation d'Haskell :

```
do a <- x
  return (f a)
```

Dans tous les cas, la propagation n'est pas aussi transparente pour le programmeur qu'avec les exceptions dans un langage en appel par valeur (où on utiliserait la simple application $f\ x$).

- Lorsqu'on utilise les exceptions, on rend artificiellement le langage strict (ceci est dû au fait que les monades d'Haskell sont en appel par valeur). La monade force l'évaluation d'un argument potentiellement exceptionnel pour vérifier s'il est exceptionnel ou non. Et même si dans la majorité des cas d'utilisation des monades (entrées-sorties, états mémoires, ...) cette évaluation est nécessaire, nous pensons qu'elle peut être évitée dans le cas des exceptions.
- La modularité et la réutilisation de code est en partie compromise, en particulier pour les fonctions d'ordre supérieur. Considérons l'exemple d'une fonction *sort* de tri de listes prenant en argument une fonction de comparaison qui renvoie un booléen. Son type en Haskell est :

```
sort :: (a -> a -> Bool) -> [a] -> [a]
```

où $[a]$ désigne le type des listes. Une telle fonction ne peut pas être utilisée avec une fonction de comparaison dont le résultat est potentiellement exceptionnel, c'est-à-dire avec une fonction *cmp* de type :

```
cmp :: a -> a -> ExVal Bool
```

Pour utiliser une telle fonction de comparaison, il faut changer le code (et donc le

type) de la fonction `sort`, du code devant y être ajouté pour propager l’exception lorsque la fonction de comparaison y est utilisée. Notons que ceci est en partie lié au fait que la propagation n’est pas implicite.

- Il n’y a pas de façon canonique de quitter la portée d’une monade, les effets représentés par une monade sont globaux. Dès qu’on utilise une exception, le code dans sa globalité devient impur et les opérations de la monade ne permettent pas de restaurer la pureté. Toutefois, des solutions à ce problème existent. La majeure partie de ces solutions consiste en l’extension du système monadique par de nouvelles constructions [Fil96, Kie98, Fil99]. Une autre solution est proposée par A. Nanevski [Nan05] qui utilise une co-monade notamment pour encoder les exceptions. Aucune de ces solutions ne règle cependant les inconvénients mentionnés dans les trois points précédents.

2.3.2 Mécanisme d’exceptions primitif

Puisqu’un codage des exceptions n’est pas pleinement satisfaisant, on peut envisager un mécanisme d’exceptions primitif pour l’appel par nom. Plusieurs solutions vont dans ce sens [DH90, Gov93, Rei98, PJRH⁺99]. Nous allons décrire la solution la plus aboutie, celles des *exceptions imprécises* de S. Peyton Jones, A. Reid, T. Hoare, S. Marlow et F. Henderson [PJRH⁺99].

Nous en avons déjà expliqué l’idée globale sous le nom d’exceptions en appel par nom. Tel qu’il est appliqué à Haskell, le mécanisme des exceptions imprécises consiste à considérer les exceptions comme des valeurs de *n’importe quel* type. Ainsi, pour *n’importe quel* type, une valeur est soit « normale », soit « exceptionnelle ». S. Peyton Jones *et al.* remarquent d’ailleurs que cette idée d’exceptions comme valeurs existe déjà dans le monde des langages impératifs avec les NaNs (Not-A-Number) et les infinités du standard IEEE pour l’arithmétique à virgule flottante [CCG⁺84].

Concrètement, le type `Exception` est défini par :

```
data Exception = DivideByZero
               | Overflow
               | UserError String
               ...
```

De plus, est ajouté à Haskell la fonction primitive `raise` qui pour tout type plonge une exception en une valeur exceptionnelle de ce type :

```
raise :: Exception -> a
```

Pour pouvoir rattraper une exception, la fonction primitive `getException` est également définie. Elle détermine si une valeur est exceptionnelle ou non² :

```
getException :: a -> ExVal a
```

où `ExVal` est simplement le type énuméré :

```
data ExVal a = OK a
             | Exn Exception
```

²Le type exact de `getException` dans [PJRH⁺99] est : `a -> IO (ExVal a)`.

Propagation. Avec ce mécanisme, les exceptions ont le même statut que les autres valeurs. En particulier, la notion de propagation doit être quelque peu réexaminée puisque les exceptions ne se propagent que de façon paresseuse, que lorsqu'elles sont utilisées (autrement dit, quand elles sont en tête de calcul). En particulier, elles peuvent se cacher dans un terme non évalué. Pour illustrer ce comportement, considérons la fonction `zipWith` définie comme suit :

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f []      []      = []
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
zipWith f xs      ys      = raise (UserError "Unequal lists")
```

Cette fonction Haskell applique la fonction `f` aux éléments de même indice dans les deux listes en entrée, renvoyant la liste résultat en sortie. Et si les listes n'ont pas la même taille, une exception est levée. Un appel à la fonction `zipWith` peut retourner diverses sortes de résultats :

- Il peut retourner une valeur exceptionnelle directement, c'est le cas par exemple avec `zipWith (+) [] [1]` ;
- Un appel à `zipWith` peut aussi retourner une liste qui n'est pas bien formée, la fin de cette liste étant une valeur exceptionnelle, par exemple `zipWith (+) [1] [1,2]`. Dans un tel résultat, l'exception joue le rôle d'une valeur de type liste. Et parce qu'Haskell est un langage paresseux, ce résultat ne s'évalue pas sur l'exception directement ;
- Un appel peut retourner une liste parfaitement formée, dont la structure est complètement définie, mais dont certains éléments sont des valeurs exceptionnelles. C'est le cas avec `zipWith (/) [1,2] [1,0]` dans lequel une exception joue le rôle d'une valeur de type entier.

Ce sont les *valeurs* qui peuvent être exceptionnelles et non les *appels* de fonctions, ces valeurs exceptionnelles pouvant se cacher dans des structures de données paresseuses. Donc pour être sûr qu'une structure de données ne contient pas de valeurs exceptionnelles, on doit forcer l'évaluation de tous les éléments de cette structure, ce qui s'effectue en Haskell grâce à la fonction primitive `seq`.

L'imprécision des exceptions. Dans cette thèse, nous allons reprendre le principe d'exceptions imprécises (mais sans l'imprécision). Nous avons précédemment désigné ce mécanisme sous le nom d'exceptions en appel par nom et le lecteur qui l'a découvert avec la présentation que nous venons d'en faire se demande probablement pourquoi Peyton Jones *et al.* l'ont nommé exceptions imprécises. La raison a à voir avec l'évaluation d'une expression telle que

```
(1/0) + (raise Overflow)
```

Avec les exceptions imprécises et contrairement au cas des exceptions en appel par valeur où généralement un ordre d'évaluation arbitraire est fixé, une telle expression renvoie l'ensemble des deux exceptions renvoyées à gauche et à droite du `+`. Une valeur exceptionnelle n'est alors plus une simple exception mais un ensemble d'exceptions. L'intérêt est qu'alors l'addition est commutative même en présence d'exceptions, et plus généralement, ce choix permet de conserver bon nombre de transformations de programmes utiles notamment pour la compilation (en particulier celles modifiant l'ordre d'évaluation).

Les calculs que nous présentons dans cette thèse seront munis d’entiers. Toutefois l’addition n’y sera pas primitive mais définie à partir d’un opérateur de récursion similaire à celui du Système T [GTL89] et ainsi un choix d’évaluation sera réellement effectué et ce choix dépendra du codage de l’addition. Nous verrons en particulier que l’addition ne sera pas commutative en présence d’exceptions. En tout cas, l’imprécision des exceptions de [PJRH⁺99] n’aura pas lieu d’être dans nos calculs et pour cette raison, nous conserverons dans la suite le nom d’exceptions en appel par nom.

2.4 Opérateurs de contrôle

Les exceptions (usuelles) sont des opérateurs de contrôle (au sens où elles modifient le flot de contrôle), mais elles ne doivent pas être confondues avec les continuations qui ont une sémantique statique. En particulier, exceptions et continuations ne peuvent pas s’exprimer l’une à partir de l’autre [RT99, Thi02].

Cependant, comme nous l’avons remarqué, le mécanisme d’exceptions usuel n’est pas adapté à l’appel par nom (voir section 1.2) et nous considérons dans cette thèse le mécanisme d’exceptions que nous avons nommé exceptions en appel par nom. Or, ce mécanisme n’est pas un opérateur de contrôle au sens où il ne capture pas, ni ne modifie le flot de contrôle du programme. Les exceptions n’y sont que des valeurs particulières du langage.

Par ailleurs, cette thèse traite de la détection statique des exceptions non rattrapées et nous ne connaissons pas de travaux sur les opérateurs de contrôle traitant de ce problème (hormis les travaux traitant spécifiquement du mécanisme des exceptions, travaux que nous avons déjà mentionnés en section 2.2).

2.5 Les exceptions en théorie des types

Un de nos objectifs est de développer un mécanisme d’exceptions adapté à la théorie des types (pris ici dans un sens très large), notamment dans son utilisation pour la preuve de programmes. Comme nous l’avons déjà précisé (section 1.3), les exceptions doivent y être précisément typées pour des raisons de cohérence. Leur gestion dans ce cadre a fait l’objet d’une attention plus récente.

Une première solution consiste à étudier un système faisant le choix de l’appel par valeur. C’est le cas du langage PML de C. Raffalli [Raf07] qui est une variante de ML destinée à la preuve de programme. PML possède un mécanisme d’exceptions précisément typé et celui-ci repose sur un système similaire à celui de X. Leroy et F. Pessaux [LP99] (voir section 2.2). Nous nous intéressons toutefois ici au cas de l’appel par nom. Comme pour les langages de programmation traditionnels, on distingue deux approches : le codage des exceptions dans le langage et les mécanismes d’exceptions primitifs.

Codage des exceptions. Le codage des exceptions dans les valeurs tel que nous l’avons présenté dans le cadre d’Haskell peut être adapté à la théorie des types. En particulier, G. Barthe, J. Hatcliff et P. Thiemann ont proposé de combiner les monades et les systèmes de types purs [BHT97], permettant ainsi un codage monadique des exceptions dans ces systèmes. Cette approche souffre néanmoins des mêmes inconvénients que ceux que nous avons décrits dans la section 2.3.1.

Exceptions primitives. La seule proposition d'ajout d'exceptions primitives dans un langage en appel par nom destiné à la preuve de programmes que nous connaissons est l'extension EX_2 d' AF_2 proposé par R. David et G. Mounier [DM04, Mou99]. Ce système propose un mécanisme d'exceptions « à la CAML » dont la logique induite est celle du système F . La réduction de EX_2 est en appel par nom, mais cependant aucune règle de typage ne permet l'application direct d'une fonction sur les entiers à une valeur exceptionnelle. En pratique, la propagation des exceptions est forcée à l'aide des opérateurs de mise en mémoire de J.L. Krivine [Kri90] permettant de simuler l'appel par valeur dans l'appel par nom. On y retrouve en particulier les inconvénients de la propagation non implicite (voir section 2.3.1). Par ailleurs, la notion de valeur exceptionnelle y est liée à celle de type de données et par exemple, la notion de valeur exceptionnelle de type $A \rightarrow B$ n'y existe pas.

3

Exceptions en appel par nom

Ce chapitre présente les termes du calcul λ_ε et leur sémantique opérationnelle. Comme nous l'avons motivé en introduction, les exceptions de ce calcul en sont des valeurs et se comportent comme telles (tant que la β -réduction est concernée). La dynamique des exceptions de λ_ε ne devra donc pas être pensée en termes de contrôle du flot d'exécution comme c'est habituellement le cas pour les exceptions.

Nous allons commencer par présenter la syntaxe des termes du calcul (section 3.1), avant de présenter la réduction et d'en démontrer les propriétés (section 3.2). Enfin, en section 3.3, nous donnerons quelques exemples.

3.1 Les termes du calcul

Définition 3.1 (Termes de λ_ε)

Un terme de λ_ε est défini comme une production de la grammaire suivante :

$t ::=$	x	(variable)
	$\lambda x. t$	(abstraction)
	$t t$	(application)
	$\text{raise } \varepsilon$	(levée d'exception)
	$\text{try } t \text{ with } \varepsilon \mapsto t$	(rattrapage d'exception)
	0	(zéro)
	S	(successeur)
	rec	(récursion sur les entiers)

À la syntaxe des termes du λ -calcul, on ajoute deux constructions manipulant les exceptions : $\text{raise } \varepsilon$ et $\text{try } t \text{ with } \varepsilon \mapsto t$. Dans ces constructions, ε représente un nom d'exception et l'on suppose donné un ensemble (infini) dénombrable de noms d'exceptions $\mathcal{E}$ (donc $\varepsilon \in \mathcal{E}$). La construction $\text{raise } \varepsilon$ lève l'exception de nom ε , alors que la construction $\text{try } t \text{ with } \varepsilon \mapsto u$ capture l'exception ε si t la lève (c'est à dire, si t se réduit sur $\text{raise } \varepsilon$), se

réduisant alors sur u . Le calcul est également muni d'entiers primitifs avec un opérateur de récursion semblable à celui du Système T de Gödel : *rec*.

Abréviation

On utilisera l'abréviation $S^n t$ (où $n \in \mathbb{N}$) pour désigner le terme

$$\underbrace{S (\dots S (t) \dots)}_{n \text{ fois}}.$$

Et en particulier, lorsque t est le terme 0, on notera plus simplement par n le terme $S^n 0$ (i.e. $1 = S 0, 2 = S (S 0), \dots$).

L'ensemble des termes de λ_ε est noté $\mathcal{T}$. Les notions de variables *libres* et *liées* sont définies comme d'habitude, de même que la *substitution* dans un terme t d'une variable x par un terme u , notée $t\{x := u\}$. L'ensemble des variables libres d'un terme t est noté $FV(t)$ et un terme est dit *clos* s'il ne possède aucune variable libre ($FV(t) = \emptyset$). De plus, nous considérerons toujours les termes modulo α -équivalence (c'est à dire modulo le choix des noms effectifs des variables liées). Enfin, remarquons que, comme il est d'usage avec les exceptions, la construction *try* t *with* $\varepsilon \mapsto u$ n'est pas un lieu (ni dans t , ni dans u) pour le nom d'exception ε , les exceptions ayant une portée globale. La substitution vérifie en particulier le lemme de substitution suivant :

Lemme 3.2 (Substitution) *Si t, u et v sont des termes quelconques et x et y sont deux variables de termes tels que $x \neq y$ et $x \notin FV(v)$, alors on a*

$$t\{x := u\}\{y := v\} = t\{y := v\}\{x := u\{y := v\}\}$$

Preuve :

La preuve de ce lemme est habituelle, les constructions spécifiques à λ_ε , à savoir les constructions *raise* et *try* ne posent pas de difficultés particulières.

L'application est associative à gauche : le terme $t u v$ doit être compris comme $(t u) v$. La portée de la lambda abstraction est la plus large possible : le terme $\lambda x. t u$ doit être compris comme $\lambda x. (t u)$.

Vocabulaire

On définit le vocabulaire suivant :

- Dans le terme *try* t *with* $\varepsilon \mapsto u$, on appellera t le *corps* de la construction *try* et u son *gestionnaire* ;
- Par abus de langage, nous appellerons *exception* à la fois le nom d'exception ε , mais aussi le terme *raise* ε .

Traditionnellement, l'évaluation du terme $\text{try } t \text{ with } \varepsilon \mapsto u$ consiste d'abord en l'évaluation du terme t , jusqu'à ce que celui-ci soit sous une forme permettant de décider s'il s'agit d'une exception ou non. Pour ce faire, on définit la notion de *valeur régulière* représentant les valeurs du langage qui ne sont pas des exceptions.

Définition 3.3 (Valeurs régulières de λ_ε)

Une valeur régulière est un terme dont la forme est définie comme une production de la grammaire suivante :

$$v ::= \lambda x. t \quad | \quad 0 \quad | \quad S \quad | \quad S t \quad | \quad \text{rec} \quad | \quad \text{rec } t \quad | \quad \text{rec } t t$$

On remarquera en particulier que le terme $S t$ est une valeur régulière, même si t n'est pas une valeur régulière. Le terme S ($\text{raise } \varepsilon$) est par exemple une valeur régulière.

La notion de *valeur* est alors définie à partir de celle de valeur régulière par :

Définition 3.4 (Valeur)

Une valeur est un terme dont la forme est définie comme une production de la grammaire suivante :

$$h ::= v \mid \text{raise } \varepsilon$$

où v représente une valeur régulière au sens donnée par la définition 3.3.

On nommera *valeur close* une valeur qui ne contient aucune variable libre.

3.2 Évaluation

3.2.1 La réduction

Définition 3.5 (Notion de réduction)

La notion de réduction est la relation $\rightarrow$ définie par les règles suivantes :

$$\begin{array}{llll} (\lambda x. t) u & \rightarrow & t\{x := u\} & (\beta) \\ (\text{raise } \varepsilon) t & \rightarrow & \text{raise } \varepsilon & (\text{rapp}) \\ \text{try } (\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto u & \rightarrow & u & (\text{catch}) \\ \text{try } (\text{raise } \varepsilon') \text{ with } \varepsilon \mapsto u & \rightarrow & \text{raise } \varepsilon' & (\text{trai}) \quad (\text{si } \varepsilon \neq \varepsilon') \\ \text{try } v \text{ with } \varepsilon \mapsto u & \rightarrow & v & (\text{tval}) \quad (\text{si } v \text{ valeur régulière}) \\ \text{rec } z f 0 & \rightarrow & z & (\text{recz}) \\ \text{rec } z f (S n) & \rightarrow & f n (\text{rec } z f n) & (\text{recs}) \\ \text{rec } z f (\text{raise } \varepsilon) & \rightarrow & \text{raise } \varepsilon & (\text{recr}) \end{array}$$

Définition 3.6 (Réduction)

La relation de réduction $\rightarrow$ est définie comme la clôture contextuelle de la notion de réduction. Plus précisément, elle est définie comme la plus petite relation dérivable en utilisant les règles d'inférences de la figure 3.1.

De plus,

- On note $\rightarrow^*$ la clôture réflexive et transitive de la réduction.
- On note $=_\beta$ la clôture réflexive, transitive et symétrique de la réduction et si deux termes t et t' vérifie $t =_\beta t'$, on dira qu'ils sont convertibles ou encore β -équivalents¹.

$$\begin{array}{c}
\frac{t \rightarrow t'}{t \rightarrow t'} (red) \qquad \frac{t \rightarrow t'}{\lambda x. t \rightarrow \lambda x. t'} (red_\lambda) \\
\\
\frac{t \rightarrow t'}{t u \rightarrow t' u} (red_{app1}) \qquad \frac{u \rightarrow u'}{t u \rightarrow t u'} (red_{app2}) \\
\\
\frac{t \rightarrow t'}{\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow \text{try } t' \text{ with } \varepsilon \mapsto u} (red_{try1}) \\
\\
\frac{u \rightarrow u'}{\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow \text{try } t \text{ with } \varepsilon \mapsto u'} (red_{try2})
\end{array}$$

FIG. 3.1 – Règles de réduction de λ_ε .

On peut remarquer que :

- Le terme `raise  $\varepsilon$` est une valeur comme une autre. Il est donc en particulier effaçable (par exemple $(\lambda x. 0) (\text{raise } \varepsilon) \rightarrow 0$) et peut être enfoui sous des abstractions ou des successeurs (par exemple, $S (\text{raise } \varepsilon)$ n'est pas réductible);
- Le terme `try  $t$  with  $\varepsilon \mapsto u$` se réduit (sur t ou u) seulement si t est une valeur régulière ou une exception.

On dira qu'un terme t lève l'exception ε si ce terme se réduit (au sens de $\rightarrow^*$) sur l'exception de nom ε (c'est à dire si $t \rightarrow^* \text{raise } \varepsilon$).

Remarquons que l'opérateur `try` est à liaison dynamique. Par exemple, lors de la réduction du terme

$$(\lambda x. \text{try } x \text{ with } \varepsilon \mapsto 0) (\text{raise } \varepsilon),$$

l'exception ε va être capturée par la construction `try` (ce terme se réduit sur 0). La construction `try` n'est pas un lieur pour ε .

Parce que la réduction est confluente (théorème 3.7 qui fait l'objet de la sous-section suivante), deux termes t et t' sont convertibles ($t =_\beta t'$) si et seulement si ils ont un réduit commun (i.e. il existe un terme u tel que $t \rightarrow^* u$ et $t' \rightarrow^* u$).

¹Remarquons que le terme β -équivalents et la notation $=_\beta$ sont des abus de langages au sens où la réduction de λ_ε ne contient pas que la règle β .

3.2.2 Confluence

La réduction de λ_ϵ est confluente :

Théorème 3.7 (Confluence de la réduction) Si t , u et u' sont des termes tels que $t \rightarrow^* u$ et $t \rightarrow^* u'$, alors il existe un terme v tel que

$$u \rightarrow^* v \quad \text{et} \quad u' \rightarrow^* v.$$

Preuve :

La preuve de confluence du calcul se fait en utilisant la méthode de Tait et Martin-Löf pour le λ -calcul que l'on pourra trouver par exemple dans [Bar84]. On définit pour cela une notion de *réduction parallèle* adaptée à notre calcul. La *réduction parallèle* d'un terme t vers un terme u notée $t \Rightarrow u$ est définie inductivement comme la plus petite relation définie par les règles d'inférences de la figure 3.2. On note $\Rightarrow^*$ la fermeture réflexive et transitive de la réduction parallèle. On utilise le lemme 3.8 pour montrer que la réduction parallèle vérifie la propriété du diamant et on conclut alors en utilisant le lemme 3.10 qui nous dit que $\rightarrow^* = \Rightarrow^*$.

$$\begin{array}{c}
\frac{}{t \Rightarrow t} \text{ (pid)} \qquad \frac{t \Rightarrow t' \quad u \Rightarrow u'}{(\lambda x. t) u \Rightarrow t' \{x := u'\}} \text{ (p}\beta\text{)} \\
\\
\frac{t \Rightarrow t'}{\lambda x. t \Rightarrow \lambda x. t'} \text{ (plam)} \qquad \frac{t \Rightarrow t' \quad u \Rightarrow u'}{t u \Rightarrow t' u'} \text{ (papp)} \\
\\
\frac{}{(\text{raise } \epsilon) t \Rightarrow \text{raise } \epsilon} \text{ (prapp)} \\
\\
\frac{u \Rightarrow u'}{\text{try } (\text{raise } \epsilon) \text{ with } \epsilon \mapsto u \Rightarrow u'} \text{ (pcatch)} \\
\\
\frac{\epsilon \neq \epsilon'}{\text{try } (\text{raise } \epsilon') \text{ with } \epsilon \mapsto u \Rightarrow \text{raise } \epsilon'} \text{ (prai)} \\
\\
\frac{t \Rightarrow t' \quad u \Rightarrow u'}{\text{try } t \text{ with } \epsilon \mapsto u \Rightarrow \text{try } t' \text{ with } \epsilon \mapsto u'} \text{ (ptry)} \\
\\
\frac{v \Rightarrow v' \quad v \text{ valeur régulière}}{\text{try } v \text{ with } \epsilon \mapsto u \Rightarrow v'} \text{ (ptval)} \\
\\
\frac{z \Rightarrow z'}{\text{rec } z f 0 \Rightarrow z'} \text{ (prz)} \qquad \frac{z \Rightarrow z' \quad f \Rightarrow f' \quad n \Rightarrow n'}{\text{rec } z f (S n) \Rightarrow f' n' (\text{rec } z' f' n')} \text{ (prs)} \\
\\
\frac{}{\text{rec } z f (\text{raise } \epsilon) \Rightarrow \text{raise } \epsilon} \text{ (prr)}
\end{array}$$

FIG. 3.2 – Réduction parallèle de λ_ϵ .

Lemme 3.8 (Propriété du diamant pour la réduction parallèle)

Soit t , u et u' des termes tel que $t \Rightarrow u$ et $t \Rightarrow u'$, alors il existe un terme v tel que

$$u \Rightarrow v \quad \text{et} \quad u' \Rightarrow v.$$

Preuve :

On procède par induction simultanée sur les dérivations de $t \Rightarrow u$ et $t \Rightarrow u'$. La majorité des cas sont soit triviaux (lorsque l'une des deux règles est (pid)), soit se résolvent par une simple application de l'hypothèse d'induction. Seuls les cas où l'une des règles est la règle $(p\beta)$ nécessitent en plus l'utilisation du lemme 3.9 afin de conclure.

Lemme 3.9 Si t , t' , u et u' sont des termes tels que $t \Rightarrow t'$ et $u \Rightarrow u'$, alors

$$t\{x := u\} \Rightarrow t'\{x := u'\}.$$

Preuve :

La preuve s'effectue par une simple induction sur la structure de t et en raisonnant par cas sur la règle ayant été utilisée pour dériver $t \Rightarrow t'$. Aucun cas ne présente de difficulté particulière.

Lemme 3.10 Soit t et u deux termes, alors $t \rightarrow^* u$ si et seulement si $t \Rightarrow^* u$.

Preuve :

Il suffit de montrer que $\Rightarrow \subseteq \rightarrow^*$ et $\rightarrow \subseteq \Rightarrow$. Les preuves de ces deux inclusions sont de simples inductions sur les définitions de $\rightarrow$ et $\Rightarrow$.

3.2.3 Réduction vers une valeur**Définition 3.11 (Avoir une valeur)**

On dira qu'un terme t a une valeur s'il se réduit sur une valeur, c'est à dire si il existe une valeur v telle que $t \rightarrow^* v$.

Remarquons que si t est un terme clos et si t a une valeur v , alors $\text{try } t \text{ with } \varepsilon \mapsto u$ se réduit ou bien sur v , ou bien sur u .

3.2.4 La réduction de tête faible

Définition 3.12 (Réduction de tête faible)

La réduction de tête faible $\rightarrow_{wh}$ est définie comme la plus petite relation dérivable en utilisant les règles d'inférences données en figure 3.3. La clôture réflexive et transitive de la réduction de tête faible est notée $\rightarrow_{wh}^*$.

$$\begin{array}{c}
 \frac{t \rightarrow t'}{t \rightarrow_{wh} t'} (hred) \qquad \frac{t \rightarrow_{wh} t'}{t u \rightarrow_{wh} t' u} (hred_{app}) \\
 \\
 \frac{t \rightarrow_{wh} t'}{\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow_{wh} \text{try } t' \text{ with } \varepsilon \mapsto u'} (hred_{try}) \\
 \\
 \frac{t \rightarrow_{wh} t'}{\text{rec } z \text{ f } t \rightarrow_{wh} \text{rec } z \text{ f } t'} (hred_{rec})
 \end{array}$$

FIG. 3.3 – Règles de réduction pour la réduction de tête faible dans λ_ε .

Remarquons que cette réduction possède une règle spécifique pour le récursur `rec` puisque la règle de réduction pour l'application ($hred_{app}$) ne prend plus ce cas en compte.

Lemme 3.13 Si t, t' et u sont trois termes tels que $t \rightarrow_{wh}^* t'$ alors

$$t\{x := u\} \rightarrow_{wh}^* t'\{x := u\}$$

Preuve :

On prouve que si $t \rightarrow_{wh} t'$ alors $t\{x := u\} \rightarrow_{wh} t'\{x := u\}$, le passage à la clôture réflexive et transitive étant ensuite direct. Cette preuve s'effectue par une simple induction sur la dérivation de $t \rightarrow_{wh} t'$. Le cas de la β -réduction utilise le lemme de substitution (lemme 3.2).

Définition 3.14 (Avoir une valeur par la réduction de tête faible)

On dira qu'un terme t a une valeur par la réduction de tête faible s'il se réduit sur une valeur par la réduction de tête faible, c'est à dire si il existe une valeur v telle que $t \rightarrow_{wh}^* v$.

On peut remarquer que les valeurs sont des formes normales pour la réduction de tête faible. La réciproque n'est pas vraie en général, même pour les termes clos, puisque `rec 0 0` ($\lambda x. x$) est par exemple une forme normale pour la réduction de tête faible mais n'est pas une valeur. Ce terme est cependant mal typé et lorsque nous aurons introduit le typage, le théorème de progression du calcul de λ_ε (théorème 4.47) montrera que les formes normales closes pour la réduction de tête faible qui sont bien typées sont des valeurs.

On va maintenant montrer la propriété de standardisation de la réduction (théorème 3.21) à partir de laquelle on va montrer qu'un terme a une valeur si et seulement si il a une valeur par la réduction de tête faible.

3.2.5 Standardisation

Le but de cette section est de montrer qu'avoir une valeur (définition 3.11) équivaut à avoir une valeur par la réduction de tête faible (définition 3.14).

Remarquons d'abord que le sens 'avoir une valeur par la réduction de tête faible implique avoir une valeur' est évident puisque la réduction contient la réduction de tête faible. Pour montrer l'autre sens de l'équivalence, on va montrer que la réduction vérifie une forme de standardisation [Mit79]. On va pour cela s'inspirer de la preuve de standardisation proposée par J. McKinna et R. Pollack dans [MP99] utilisant une notion de *réduction parallèle interne*.

Définition 3.15 (Réduction parallèle interne)

La réduction parallèle interne (ou plus simplement réduction interne) pour le calcul λ_ϵ est définie comme la plus petite relation dérivable à l'aide des règles d'inférence de la figure 3.4.

$$\begin{array}{c} \frac{}{t \Rightarrow_i t} \text{ (iid)} \quad \frac{t \Rightarrow t'}{\lambda x. t \Rightarrow_i \lambda x. t'} \text{ (ilam)} \quad \frac{t \Rightarrow_i t' \quad u \Rightarrow u'}{t u \Rightarrow_i t' u'} \text{ (iapp)} \\[10pt] \frac{t \Rightarrow_i t' \quad u \Rightarrow u'}{\text{try } t \text{ with } \epsilon \mapsto u \Rightarrow_i \text{try } t' \text{ with } \epsilon \mapsto u'} \text{ (itry)} \end{array}$$

FIG. 3.4 – La réduction interne

Cette réduction autorise la réduction parallèle de tous les radicaux à l'exclusion de ceux en tête.

Commençons par montrer que la réduction interne implique la réduction parallèle :

Lemme 3.16 Si t et u sont deux termes tels que $t \Rightarrow_i u$, alors

$$t \Rightarrow u$$

Preuve :

Il est facile de vérifier que la réduction interne est bien un cas particulier de la réduction parallèle.

On montre également les deux lemmes suivants :

Lemme 3.17 Si t, t', u, u' et v sont des termes tels que $t \Rightarrow_i t'$ et $u \xrightarrow{*}_{wh} v \Rightarrow_i u'$ et $u \Rightarrow u'$, alors il existe un terme w tel que

$$t\{x := u\} \xrightarrow{*}_{wh} w \Rightarrow_i t'\{x := u'\}$$

Preuve :

On raisonne par induction sur la dérivation de $t \Rightarrow_i t'$. Si la dernière règle utilisée est :

(iid) On est exactement dans le cas du lemme 3.18.

(ilam) On est dans le cas où il existe s et s' tel que $t = \lambda x.s$, $t' = \lambda x.s'$ et $s \Rightarrow s'$. Alors $t\{x := u\} = \lambda x.s\{x := u\}$ et $t'\{x := u'\} = \lambda x.s'\{x := u'\}$ (par définition de la substitution). Mais puisque l'on a $u \Rightarrow u'$, le lemme 3.9 nous donne $s\{x := u\} \Rightarrow s'\{x := u'\}$ et finalement $\lambda x.s\{x := u\} \Rightarrow_i \lambda x.s'\{x := u'\}$ avec la règle (ilam).

(iapp) On est dans le cas où il existe r, r', s et s' tel que $t = r s$ et $t' = r' s'$ avec $r \Rightarrow_i r'$ et $s \Rightarrow s'$. Par hypothèse d'induction il existe r_0 tel que $r\{x := u\} \xrightarrow{*}_{wh} r_0 \Rightarrow_i r'\{x := u'\}$ et donc $t \xrightarrow{*}_{wh} r_0 s\{x := u\}$. Mais comme $s \Rightarrow s'$ et $u \Rightarrow u'$, le lemme 3.9 nous donne $s\{x := u\} \Rightarrow s'\{x := u'\}$ et donc finalement, en appliquant la règle (iapp), $r_0 s\{x := u\} \Rightarrow_i r'\{x := u'\} s'\{x := u'\}$.

(itry) On est dans le cas où il existe r, r', s et s' tel que $t = \text{try } r \text{ with } \varepsilon \mapsto s$ et $t' = \text{try } r' \text{ with } \varepsilon \mapsto s'$ avec $r \Rightarrow_i r'$ et $s \Rightarrow s'$. Par hypothèse d'induction il existe r_0 tel que $r\{x := u\} \xrightarrow{*}_{wh} r_0 \Rightarrow_i r'\{x := u'\}$ et donc $t \xrightarrow{*}_{wh} \text{try } r_0 \text{ with } \varepsilon \mapsto s\{x := u\}$. Mais comme $s \Rightarrow s'$ et $u \Rightarrow u'$, le lemme 3.9 nous donne $s\{x := u\} \Rightarrow s'\{x := u'\}$ et donc finalement, en appliquant la règle (iapp), $\text{try } r_0 \text{ with } \varepsilon \mapsto s\{x := u\} \Rightarrow_i \text{try } r'\{x := u'\} \text{ with } \varepsilon \mapsto s'\{x := u'\}$.

Lemme 3.18 Si t, u, u' et v sont des termes tels que $u \xrightarrow{*}_{wh} v \Rightarrow_i u'$ et $u \Rightarrow u'$, alors il existe un terme w tel que

$$t\{x := u\} \xrightarrow{*}_{wh} w \Rightarrow_i t\{x := u'\}$$

Preuve :

On procède par induction sur la structure du terme t :

- Si $t = x$, alors on conclut directement avec l'hypothèse $u \xrightarrow{*}_{wh} v \Rightarrow_i u'$.
- Si $t = y$ (où $y \neq x$) ou $t = \text{raise } \varepsilon$ ou $t = 0$ ou $t = S$ ou $t = \text{rec}$, alors $t\{x := u\} = t\{x := u'\}$ et on conclut directement.
- Si $t = \lambda y.s$, alors puisque $s \Rightarrow s$ et $u \Rightarrow u'$, le lemme 3.9 nous donne $s\{x := u\} \Rightarrow s\{x := u'\}$. Donc finalement $\lambda x.s\{x := u\} \Rightarrow_i \lambda x.s\{x := u'\}$.
- Si $t = r s$, alors par hypothèse d'induction, il existe r_0 tel que $r\{x := u\} \xrightarrow{*}_{wh} r_0 \Rightarrow_i r\{x := u'\}$ et donc $t\{x := u\} \xrightarrow{*}_{wh} r_0 s\{x := u\}$. Par ailleurs, puisque $s \Rightarrow s$ et $u \Rightarrow u'$, le lemme 3.9 nous donne $s\{x := u\} \Rightarrow s\{x := u'\}$. Donc finalement $r_0 s\{x := u\} \Rightarrow_i r\{x := u'\} s\{x := u'\}$.
- Si $t = \text{try } r \text{ with } \varepsilon \mapsto s$, alors par hypothèse d'induction, il existe r_0 tel que $r\{x := u\} \xrightarrow{*}_{wh} r_0 \Rightarrow_i r\{x := u'\}$ et donc $t\{x := u\} \xrightarrow{*}_{wh} \text{try } r_0 \text{ with } \varepsilon \mapsto s\{x := u\}$. Par ailleurs, puisque $s \Rightarrow s$ et $u \Rightarrow u'$, le lemme 3.9 nous donne $s\{x := u\} \Rightarrow s\{x := u'\}$. Donc finalement $\text{try } r_0 \text{ with } \varepsilon \mapsto s\{x := u\} \Rightarrow_i \text{try } r\{x := u'\} \text{ with } \varepsilon \mapsto s\{x := u'\}$.

On va montrer que toute suite de réductions peut se réécrire comme une suite de réductions de tête faible suivie d'une suite de réductions internes. On commence par montrer

le lemme suivant (qui utilise la réduction parallèle plutôt que la réduction, mais rappelons que $\Rightarrow^* = \rightarrow^*$ (lemme 3.10)) :

Lemme 3.19 (Semi-Standardisation) *Si t et u sont deux termes tels que $t \Rightarrow u$, alors il existe un terme t_0 tel que*

$$t \rightarrow_{wh}^* t_0 \Rightarrow_i u$$

Preuve :

La preuve procède d'une induction sur la dérivation de $t \Rightarrow u$. Si la dernière règle utilisée est :

(*pid*) C'est vrai car $t \Rightarrow_i t$.

(*pβ*) On a $(\lambda x. t) u \Rightarrow t' \{x := u'\}$ car $t \Rightarrow t'$ et $u \Rightarrow u'$. Par hypothèse d'induction sur $t \Rightarrow t'$, il existe t_0 tel que $t \rightarrow_{wh}^* t_0 \Rightarrow_i t$ et donc $(\lambda x. t) u \rightarrow_{wh} t' \{x := u\} \rightarrow_{wh}^* t_0 \{x := u\}$ (en utilisant le lemme 3.13). Ensuite, comme par hypothèse d'induction sur $u \Rightarrow u'$, il existe u_0 tel que $u \rightarrow_{wh}^* u_0 \Rightarrow_i u$, on conclut en appliquant le lemme 3.17.

(*plam*) On a $\lambda x. t \Rightarrow \lambda x. t'$ car $t \Rightarrow t'$. Donc on a directement $\lambda x. t \Rightarrow_i \lambda x. t'$.

(*papp*) On a $t u \Rightarrow t' u'$ car $t \Rightarrow t'$ et $u \Rightarrow u'$. Par hypothèse d'induction sur $t \Rightarrow t'$, il existe t_0 tel que $t \rightarrow_{wh}^* t_0 \Rightarrow_i t'$. Alors $t u \rightarrow_{wh}^* t_0 u \Rightarrow_i t' u'$.

(*prapp*) On conclut directement puisque $(\text{raise } \varepsilon) t \rightarrow_{wh} \text{raise } \varepsilon$.

(*pcatch*) On a $\text{try } (\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto u \Rightarrow u'$ car $u \Rightarrow u'$. Or $\text{try } (\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto u \rightarrow_{wh} u$ et on conclut donc avec l'hypothèse d'induction appliquée à $u \Rightarrow u'$.

(*prai*) On a $\text{try } (\text{raise } \varepsilon') \text{ with } \varepsilon \mapsto u \Rightarrow \text{raise } \varepsilon'$, mais on conclut alors directement puisque $\text{try } (\text{raise } \varepsilon') \text{ with } \varepsilon \mapsto u \rightarrow_{wh} \text{raise } \varepsilon'$.

(*ptry*) On a $\text{try } t \text{ with } \varepsilon \mapsto u \Rightarrow \text{try } t' \text{ with } \varepsilon \mapsto u'$ car $t \Rightarrow t'$ et $u \Rightarrow u'$. Par hypothèse d'induction sur $t \Rightarrow t'$, il existe t_0 tel que $t \rightarrow_{wh}^* t_0 \Rightarrow_i t'$. Alors $\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow_{wh} \text{try } t_0 \text{ with } \varepsilon \mapsto u \Rightarrow_i \text{try } t' \text{ with } \varepsilon \mapsto u'$.

(*ptval*) On a $\text{try } v \text{ with } \varepsilon \mapsto u \Rightarrow v'$ car $v \Rightarrow v'$ et v est une valeur régulière. Mais alors $\text{try } v \text{ with } \varepsilon \mapsto u \rightarrow_{wh} v$ et on conclut par hypothèse d'induction appliqué à $v \Rightarrow v'$.

(*prz*) On a $\text{rec } z f 0 \Rightarrow z'$ car $z \Rightarrow z'$. On applique alors l'hypothèse d'induction sur $z \Rightarrow z'$ et on obtient qu'il existe z_0 tel que $z \rightarrow_{wh}^* z_0 \Rightarrow_i z'$. Mais alors $\text{rec } z f 0 \rightarrow_{wh} z \rightarrow_{wh}^* z_0 \Rightarrow_i z'$.

(*prs*) On a $\text{rec } z f (S n) \Rightarrow f' n' (\text{rec } z' f' n')$ car $z \Rightarrow z'$, $f \Rightarrow f'$ et $n \Rightarrow n'$. On a d'abord $\text{rec } z f (S n) \rightarrow_{wh} f n (\text{rec } z f)$. Ensuite, par hypothèse d'induction sur $f \Rightarrow f'$, il existe f_0 tel que $f \rightarrow_{wh}^* f_0 \Rightarrow_i f'$ et donc, $f n (\text{rec } z f) \rightarrow_{wh}^* f_0 n (\text{rec } z f)$. Enfin, puisque $n \Rightarrow n'$ et qu'on dérive facilement des hypothèses que $\text{rec } z f n \Rightarrow \text{rec } z' f' n'$, on a $f_0 n (\text{rec } z f n) \Rightarrow_i f' n' (\text{rec } z' f' n')$.

(*prr*) On conclut directement puisque $\text{rec } z f (\text{raise } \varepsilon) \rightarrow_{wh} \text{raise } \varepsilon$.

On montre alors que les pas de réduction de tête faible peuvent toujours s'effectuer avant ceux de la réduction interne :

Lemme 3.20 *Si t , u et v sont des termes tels que $t \Rightarrow_i v \rightarrow_{wh} u$, alors il existe v' tel que*

$$t \rightarrow_{wh}^* v' \Rightarrow_i u$$

Preuve :

Commençons par remarquer que si la règle appliquée pour dériver $t \Rightarrow_i v$ est (iid) alors $v = t$ et on conclut directement en prenant $v' = u$ puisque $u \Rightarrow_i u$. On considérera donc dans la suite seulement les cas où ce n'est pas la règle (iid) qui a été utilisée. On procède alors par induction sur la dérivation de $v \rightarrow_{wh} u$. Si la dernière règle utilisée est :

(hred) On est forcément dans l'un des cas suivants :

- Le terme v est une application et donc $v = r' s'$. Mais alors on a forcément utilisé (iapp) pour dériver $t \Rightarrow_i v$ et donc $t = r s$ avec $r \Rightarrow_i r'$ et $s \Rightarrow s'$. Les cas possibles sont alors :
 - $r' = \lambda x. r_1$ et $u = r_1 \{x := s'\}$. Mais puisque $r \Rightarrow_i \lambda x. r_1$, on a forcément $r = \lambda x. r_0$ et $r_0 \Rightarrow r_1$. Comme de plus $s \Rightarrow s'$, le lemme 3.9 nous donne $r_0 \{x := s\} \Rightarrow r_1 \{x := s'\}$. Alors, $(\lambda x. r_0) s \rightarrow_{wh} r_0 \{x := s\}$ et on conclut en appliquant le lemme de semi-standardisation (lemme 3.19) sur $r_0 \{x := s\} \Rightarrow r_1 \{x := s'\}$.
 - $r' = \text{raise } \varepsilon$ et $u = \text{raise } \varepsilon$. Mais puisque $r \Rightarrow_i \text{raise } \varepsilon$, on a forcément $r = \text{raise } \varepsilon$ et donc $r s \rightarrow_{wh} \text{raise } \varepsilon = u$.
 - $r' = \text{rec } z_1 f_1$, $s' = 0$ et $u = z_1$. Mais puisque $r \Rightarrow_i \text{rec } z_1 f_1$, on vérifie facilement que l'on a forcément $r = \text{rec } z_0 f_0$ avec $z_0 \Rightarrow z_1$ et $f_0 \Rightarrow f_1$. Comme $s \Rightarrow s'$, par le lemme de semi-standardisation (lemme 3.19) on a $s \xrightarrow{*}_{wh} s_0 \Rightarrow_i s'$ et comme $s' = 0$, on a forcément $s_0 = 0$. Donc finalement $r s = \text{rec } z_0 f_0 s \xrightarrow{*}_{wh} \text{rec } z_0 f_0 0 \rightarrow_{wh} z_0$ et on conclut avec le lemme de semi-standardisation (lemme 3.19) sur $z_0 \Rightarrow z_1$.
 - $r' = \text{rec } z_1 f_1$, $s' = S n_1$ et $u = f_1 n_1 (\text{rec } z_1 f_1 n_1)$. Mais puisque $r \Rightarrow_i \text{rec } z_1 f_1$, on vérifie facilement que l'on a forcément $r = \text{rec } z_0 f_0$ avec $z_0 \Rightarrow z_1$ et $f_0 \Rightarrow f_1$. Comme $s \Rightarrow s'$, par le lemme de semi-standardisation (lemme 3.19) on a $s \xrightarrow{*}_{wh} s_0 \Rightarrow_i s'$ et comme $s' = S n_1$, on a forcément $s_0 = S n_0$ avec $n_0 \Rightarrow n_1$. Donc $r s = \text{rec } z_0 f_0 s \xrightarrow{*}_{wh} \text{rec } z_0 f_0 (S n_0) \rightarrow_{wh} f_0 n_0 (\text{rec } z_0 f_0 n_0)$. De plus, avec le lemme de semi-standardisation (lemme 3.19) sur $f_0 \Rightarrow f_1$, on a $f_0 n_0 (\text{rec } z_0 f_0 n_0) \xrightarrow{*}_{wh} f'_0 \Rightarrow_i f_1 n_1 (\text{rec } z_1 f_1 n_1)$.
 - $r' = \text{rec } z_1 f_1$, $s' = \text{raise } \varepsilon$ et $u = \text{raise } \varepsilon$. Mais puisque $r \Rightarrow_i \text{rec } z_1 f_1$, on vérifie facilement que l'on a forcément $r = \text{rec } z_0 f_0$ avec $z_0 \Rightarrow z_1$ et $f_0 \Rightarrow f_1$. Comme $s \Rightarrow s'$, par le lemme de semi-standardisation (lemme 3.19) on a $s \xrightarrow{*}_{wh} s_0 \Rightarrow_i s'$ et comme $s' = \text{raise } \varepsilon$, on a forcément $s_0 = \text{raise } \varepsilon$. Donc finalement $r s = \text{rec } z_0 f_0 s \xrightarrow{*}_{wh} \text{rec } z_0 f_0 \text{raise } \varepsilon \rightarrow_{wh} \text{raise } \varepsilon$ et on conclut puisque $\text{raise } \varepsilon \Rightarrow_i \text{raise } \varepsilon$.
- Le terme v est un try et donc $v = \text{try } r' \text{ with } \varepsilon \mapsto s'$. Mais alors on a forcément utilisé (itry) pour dériver $t \Rightarrow_i v$ et donc $t = \text{try } r \text{ with } \varepsilon \mapsto s$ avec $r \Rightarrow_i r'$ et $s \Rightarrow s'$. Les cas possible sont alors :
 - $r' = \text{raise } \varepsilon$ et $u = s'$. Puisque $r \Rightarrow_i \text{raise } \varepsilon$, on a forcément $r = \text{raise } \varepsilon$ et donc $\text{try } r \text{ with } \varepsilon \mapsto s \rightarrow_{wh} s$ et on conclut avec le lemme de semi-standardisation (lemme 3.19) appliqué à $s \Rightarrow s'$.
 - $r' = \text{raise } \varepsilon'$ (où $\varepsilon \neq \varepsilon'$) et $u = \text{raise } \varepsilon'$. Puisque $r \Rightarrow_i \text{raise } \varepsilon'$, on a forcément $r = \text{raise } \varepsilon'$ et donc $\text{try } r \text{ with } \varepsilon \mapsto s \rightarrow_{wh} \text{raise } \varepsilon'$ et on conclut puisque $\text{raise } \varepsilon' \Rightarrow_i \text{raise } \varepsilon'$.
 - r' est une valeur régulière, et $u = r'$. On raisonne par cas sur r' :
 - Si r' est 0, S ou rec, alors on a forcément $r = r'$ et donc $\text{try } r \text{ with } \varepsilon \mapsto s \rightarrow_{wh} r$ et on conclut puisque $r \Rightarrow_i r'$.
 - Si r' est $\lambda x. r_1$, S r_1 ou rec r_1 , alors on a forcément que r est $\lambda x. r_0$, S r_0 ou rec r_0 avec $r_0 \Rightarrow r_1$. Mais alors $\text{try } r \text{ with } \varepsilon \mapsto s \rightarrow_{wh} r$ et on conclut car dans tous les cas $r \Rightarrow_i r'$.
 - Si $r' = \text{rec } r_1 r'_1$, alors on a forcément $r = \text{rec } r_0 r'_0$ avec $r'_0 \Rightarrow r'_1$ et $\text{rec } r_0 \Rightarrow_i \text{rec } r_1$. Mais alors $\text{try } r \text{ with } \varepsilon \mapsto s \rightarrow_{wh} r$ et on conclut car dans tous les cas $r \Rightarrow_i r'$.

(hred_{app}) On est dans le cas où $v = r_1 s_1$ et $u = r_2 s_1$ avec $r_1 \rightarrow_{wh} r_2$. Mais alors on a forcément utilisé (iapp) pour dériver $t \Rightarrow_i v$ et donc $t = r_0 s_0$ avec $r_0 \Rightarrow_i r_1$ et $s_0 \Rightarrow s_1$. Donc $r_0 \Rightarrow_i r_1 \rightarrow_{wh} r_2$ et on peut appliquer l'hypothèse d'induction pour obtenir $r_0 \xrightarrow{*}_{wh} r'_1 \Rightarrow_i r_2$ et alors $r_0 s_0 \xrightarrow{*}_{wh} r'_1 s_0 \Rightarrow_i r_2 s_1$.

(*hred_{rec}*) On est dans le cas où $v = \text{rec } z_1 f_1 n_1$ et $u = \text{rec } z_1 f_1 n_2$ avec $n_1 \rightarrow_{wh} n_2$. Mais alors on a forcément utiliser (*iapp*) pour dériver $t \Rightarrow_i v$ et donc $t = \text{rec } z_0 f_0 n_0$ avec $\text{rec } z_0 f_0 \Rightarrow_i \text{rec } z_1 f_1$ et $n_0 \Rightarrow n_1$. En appliquant le lemme de semi-standardisation (lemme 3.19) sur $n_0 \Rightarrow n_1$, on obtient $n_0 \xrightarrow{*}_{wh} n'_0 \Rightarrow_i n_1$. Donc $n'_0 \Rightarrow_i n_1 \rightarrow_{wh} n_2$ et on peut appliquer l'hypothèse d'induction pour obtenir $n'_0 \xrightarrow{*}_{wh} n'_1 \Rightarrow_i n_2$. De plus, puisque $n'_1 \Rightarrow_i n_2$, avec le lemme 3.16 on a $n'_1 \Rightarrow n_2$. Finalement, $\text{rec } z_0 f_0 n_0 \xrightarrow{*}_{wh} \text{rec } z_0 f_0 n'_0 \xrightarrow{*}_{wh} \text{rec } z_0 f_0 n'_1 \Rightarrow_i \text{rec } z_1 f_1 n_2$.

(*hred_{try}*) On est dans le cas où $v = \text{try } r_1 \text{ with } \varepsilon \mapsto s_1$ et $u = \text{try } r_2 \text{ with } \varepsilon \mapsto s_1$ avec $r_1 \rightarrow_{wh} r_2$. Mais alors on a forcément utiliser (*itry*) pour dériver $t \Rightarrow_i v$ et donc $t = \text{try } r_0 \text{ with } \varepsilon \mapsto s_0$ avec $r_0 \Rightarrow_i s_0$ et $s_0 \Rightarrow s_1$. Donc $r_0 \Rightarrow_i r_1 \rightarrow_{wh} r_2$ et on peut appliquer l'hypothèse d'induction pour obtenir $r_0 \xrightarrow{*}_{wh} r'_1 \Rightarrow_i r_2$ et alors $\text{try } r_0 \text{ with } \varepsilon \mapsto s_0 \xrightarrow{*}_{wh} \text{try } r'_1 \text{ with } \varepsilon \mapsto s_0 \Rightarrow_i \text{try } r_2 \text{ with } \varepsilon \mapsto s_1$.

Enfin, on montre le théorème de standardisation de λ_ε :

Théorème 3.21 (Standardisation) *Si t et u sont des termes tels que $t \rightarrow^* u$, alors il existe un terme v tel que*

$$t \xrightarrow{*}_{wh} v \Rightarrow_i^* u$$

Preuve :

On utilise d'abord le lemme 3.10 pour montrer que $t \Rightarrow^* u$. On procède alors par induction sur la longueur de la réduction $t \Rightarrow^* u$:

- Soit $t = u$, la réduction est de longueur 0 et on conclut directement.
- Sinon, c'est qu'il existe t' tel que $t \Rightarrow t' \Rightarrow^* u$. Alors le lemme de semi-standardisation (lemme 3.19) nous dit qu'il existe t_0 tel que $t \xrightarrow{*}_{wh} t_0 \Rightarrow_i t'$ et l'hypothèse d'induction nous dit qu'il existe t'_0 tel que $t' \xrightarrow{*}_{wh} t'_0 \Rightarrow_i u$. Alors, il est facile avec le lemme 3.20 de montrer par induction sur la longueur de $t_0 \Rightarrow_i t' \xrightarrow{*}_{wh} t'_0$ qu'il existe v tel que $t_0 \xrightarrow{*}_{wh} v \Rightarrow_i t'_0$ et finalement $t \xrightarrow{*}_{wh} t_0 \xrightarrow{*}_{wh} v \Rightarrow_i t'_0 \Rightarrow_i u$, c'est à dire $t \xrightarrow{*}_{wh} v \Rightarrow_i u$.

Par définition de la réduction interne il est alors facile de vérifier que si t et u sont deux termes tel que $t \Rightarrow_i^* u$ et u est une valeur, alors t est également une valeur et de même forme (*i.e.* si u est une λ -abstraction alors t aussi, si u est raise ε alors t aussi, etc...). On a donc le corollaire suivant au théorème de standardisation :

Corollaire *Un terme t a une valeur si et seulement si il a une valeur par la réduction de tête faible.*

Ce théorème de standardisation et plus précisément son corollaire nous permettent en particulier de prouver les lemmes suivants :

Lemme 3.22 *Si t et u sont des termes clos tels que $t \rightarrow^* u$ ait une valeur alors*

$$t \text{ a une valeur}$$

Preuve :

Grâce au théorème de standardisation (théorème 3.21), puisque $t u$ a une valeur, $t u$ a une valeur par la réduction de tête faible. On va alors montrer que t a une valeur par la réduction de tête faible, d'où on conclura trivialement qu'il a une valeur.

Puisque $t u$ a une valeur par la réduction de tête faible, il existe une valeur v tel que $t u \rightarrow_{wh}^n v$ pour $n \geq 0$. On raisonne alors par induction sur la longueur n de cette dérivation :

- Si $n = 0$, alors $t u$ est une valeur. Mais alors par définition des valeurs, on a forcément $t = S$ ou $t = \text{rec}$ ou $t = \text{rec } t'$ (où t' est un terme quelconque). Dans tout ces cas on peut conclure puisque t est alors lui-même une valeur.
- Sinon, il existe z tel que $t u \rightarrow_{wh} z \rightarrow_{wh}^{n-1} v$. Alors, on raisonne par cas sur la réduction utilisée pour dériver $t u \rightarrow_{wh} z$:

(*hred*) Mais alors on a appliqué l'une des règles (β), (*rapp*), (*recz*), (*recs*) ou (*recr*). Dans tous ces cas, t est une valeur et on conclut directement.

(*hred_{app}*) Alors c'est que $z = t' u$ avec $t u \rightarrow_{wh} t' u$ car $t \rightarrow_{wh} t'$. Alors par hypothèse d'induction, t' a une valeur par la réduction de tête faible. Et comme $t \rightarrow_{wh} t'$, on a finalement que t a une valeur par la réduction de tête faible.

(*hred_{rec}*) Alors c'est que $t = \text{rec } z f$ (où z et f sont des termes quelconques) et donc t est une valeur et on conclut directement.

Les autres règles de la réduction de tête faible ne sont pas applicables.

Lemme 3.23 Si t et u sont des termes clos tels que $\text{try } t \text{ with } \varepsilon \mapsto u$ ait une valeur, alors

t a une valeur

Preuve :

Grâce au théorème de standardisation (théorème 3.21), puisque $\text{try } t \text{ with } \varepsilon \mapsto u$ a une valeur, $\text{try } t \text{ with } \varepsilon \mapsto u$ a aussi une valeur par la réduction de tête faible. On va alors montrer que t a une valeur par la réduction de tête faible, d'où on conclura trivialement qu'il a une valeur.

Puisque $\text{try } t \text{ with } \varepsilon \mapsto u$ a une valeur par la réduction de tête faible, il existe une valeur v tel que $\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow_{wh}^n v$ pour $n \geq 0$. On raisonne alors par induction sur la longueur n de cette dérivation :

- Si $n = 0$, alors $\text{try } t \text{ with } \varepsilon \mapsto u$ est une valeur, ce qui est impossible.
- Sinon, il existe z tel que $\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow_{wh} z \rightarrow_{wh}^{n-1} v$. Alors, on raisonne par cas sur la réduction utilisée pour dériver $\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow_{wh} z$:

(*hred*) Mais alors on a appliqué l'une des règles (*catch*), (*trai*) ou (*tval*). Dans tous ces cas, t est une valeur et on conclut directement.

(*hred_{try}*) Alors c'est que $z = \text{try } t' \text{ with } \varepsilon \mapsto u$ avec $\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow_{wh} \text{try } t' \text{ with } \varepsilon \mapsto u$ car $t \rightarrow_{wh} t'$. Alors par hypothèse d'induction, t' a une valeur par la réduction de tête faible. Et comme $t \rightarrow_{wh} t'$, on a finalement que t a une valeur par la réduction de tête faible.

Les autres règles de la réduction de tête faible ne sont pas applicables.

Lemme 3.24 Si t , z et f sont des termes tels que $\text{rec } z f t$ ait une valeur, alors

t a une valeur

Preuve :

Grâce au théorème de standardisation (théorème 3.21), puisque $\text{rec } z \ f \ t$ a une valeur, $\text{rec } z \ f \ t$ a une valeur par la réduction de tête faible. On va alors montrer que t a une valeur par la réduction de tête faible, d'où on conclura trivialement qu'il a une valeur.

Puisque $\text{rec } z \ f \ t$ a une valeur par la réduction de tête faible, il existe une valeur v tel que $\text{rec } z \ f \ t \rightarrow_{wh}^n v$ pour $n \geq 0$. On raisonne alors par induction sur la longueur n de cette dérivation :

- Si $n = 0$, alors $\text{rec } z \ f \ t$ est une valeur, ce qui n'est pas possible.
- Sinon, il existe u tel que $\text{rec } z \ f \ t \rightarrow_{wh} u \rightarrow_{wh}^{n-1} v$. Alors, on raisonne par cas sur la réduction utilisée pour dériver $\text{rec } z \ f \ t \rightarrow_{wh} u$:

(*hred*) Mais alors on a appliqué l'une des règles (*recz*), (*recs*) ou (*recr*). Dans tous ces cas, t est une valeur et on conclut directement.

(*hred_{app}*) C'est impossible puisque $\text{rec } z \ f$ n'est pas réductible par la réduction de tête faible.

(*hred_{rec}*) Alors c'est que $u = \text{rec } z \ f \ t'$ avec $\text{rec } z \ f \ t \rightarrow_{wh} \text{rec } z \ f \ t'$ car $t \rightarrow_{wh} t'$. Alors par hypothèse d'induction, t' a une valeur par la réduction de tête faible. Et comme $t \rightarrow_{wh} t'$, on a finalement que t a une valeur par la réduction de tête faible.

Les autres règles de la réduction de tête faible ne sont pas applicables.

3.3 Exemples

Dans cette section, nous allons présenter quelques exemples de codage dans λ_ϵ . Nous n'avons pas encore présenté le typage de λ_ϵ et nous nous contentons donc de donner des termes et leurs réductions. Toutefois, nous ne présenterons que des termes bien typés. Ces exemples seront repris en section 4.3 où nous en donnerons le type.

On commence par le prédécesseur sur les entiers naturels, mais renvoyant une exception lorsqu'il est appliqué à 0 :

$$\text{pred} \equiv \text{rec } (\text{raise } \text{argis0}) \ (\lambda n. \lambda r. n)$$

Il est alors facile de vérifier que l'on a les réductions suivantes :

- $\text{pred } 0 \rightarrow \text{raise } \text{argis0}$
- $\text{pred } (S \ n) \rightarrow^* n$

A partir de cette fonction et en utilisant la construction `try`, on peut définir une autre fonction prédécesseur ne pouvant plus lever d'exception (et renvoyant 0 lorsqu'elle est appliquée à 0) :

$$\text{safe_pred} \equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \text{argis0} \mapsto 0$$

Le prédécesseur peut alors être utilisé pour coder la soustraction, où l'on renverra une exception lorsque le résultat n'est pas un entier positif :

$$\text{minus} \equiv \lambda a. \lambda b. \text{rec } a \ (\lambda n. \lambda r. \text{pred } r) \ b$$

La fonction *minus* vérifie les réductions suivantes :

- $\text{minus } 6 \ 2 \rightarrow^* 4$
- $\text{minus } 3 \ 3 \rightarrow^* 0$
- $\text{minus } 2 \ 5 \rightarrow^* \text{raise } \text{argis0}$

Définissons également la fonction d'addition :

$$add \equiv \lambda a. \lambda b. \text{rec } b \ (\lambda n. \lambda r. S \ r) \ a$$

On peut alors vérifier que l'on a les réductions suivantes :

- $add \ 3 \ (minus \ 3 \ 1) \rightarrow^* 5$
- $add \ 3 \ (minus \ 1 \ 3) \rightarrow^* S \ (S \ (S \ (raise \ argis0)))$
- $add \ (minus \ 1 \ 3) \ 3 \rightarrow^* raise \ argis0$

Remarquons que l'addition n'est plus symétrique dès lors que l'un des arguments est une exception. Le résultat reflète le choix fait lors du codage de l'addition, à savoir utiliser (et donc évaluer) le premier argument pour la récursion. Cet exemple nous montre également (avec le second point se réduisant sur $S \ (S \ (S \ (raise \ argis0)))$) que nos exceptions sont des valeurs pouvant être encapsulées dans une structure de donnée.

Si l'on veut être sûr de ne pas se retrouver dans le cas où une exception est encapsulée dans l'entier renvoyé, on peut écrire une fonction qui évalue l'entier et si celui-ci contient l'exception $raise \ \varepsilon$, la fait remonter à la racine. Une telle fonction *eval* est codée par :

$$eval \equiv \lambda n. (\text{rec } (\lambda a. a) \ (\lambda _ . \lambda r. \lambda a. r \ (S \ a)) \ n) \ 0$$

et on vérifie que l'on a les réductions suivantes :

- $eval \ (add \ 3 \ (minus \ 3 \ 1)) \rightarrow^* 5$
- $eval \ (add \ 3 \ (minus \ 1 \ 3)) \rightarrow^* raise \ argis0$
- $eval \ (add \ (minus \ 1 \ 3) \ 3) \rightarrow^* raise \ argis0$

Nous reviendrons sur une telle fonction, particulièrement utile dans λ_ε , lors de la présentation du système de types.

Les quelques exemples que nous venons de montrer n'ont pour but que de familiariser le lecteur avec la réduction des exceptions dans λ_ε . Dans le chapitre 4, le système de type de λ_ε sera présenté et des exemples plus conséquents seront alors donnés (section 4.3).

4

Le système de types

C E chapitre présente le système de type du langage λ_ϵ dont nous avons présenté la syntaxe et le comportement calculatoire dans le chapitre 3. La suite de ce chapitre s'organise de la façon suivante : on commence par présenter les motivations qui ont abouti au présent système de types et on donne les principales intuitions derrière les notions qu'il utilise (section 4.1). On présente alors formellement ce système de type dans la section 4.2 avant d'en donner des exemples d'utilisations dans la section 4.3. Enfin, on démontre dans la section 4.4 les principales propriétés du langage λ_ϵ que son système de types lui assure (préservation du typage par réduction, progression du calcul, ...).

4.1 Principe

4.1.1 Quel système de types ?

La conception du système de types que nous allons présenter a été guidée par la volonté de lui faire respecter un certain nombre de principes :

Principe 1 Dans tout terme bien typé, on peut remplacer n'importe quel sous-terme par une exception, le résultat restant bien typé (mais pas nécessairement de même type) ;

Principe 2 Les types d'un terme indiquent les exceptions qui peuvent s'échapper durant l'évaluation de ce dernier. En particulier, si le type d'un terme ne mentionne pas une certaine exception, il ne pourra pas lever cette exception lors de son évaluation ;

Principe 3 Toute dérivation de typage du λ -calcul simplement typé est une dérivation valide de notre système de type ;

Principe 4 Le typage respecte la modularité et favorise la ré-utilisabilité du code. Le principe 1 s'applique même en présence de fonctions dont seul le type est connu.

4.1.2 Trois niveaux de corruption

Le typage des exceptions dans λ_ϵ repose principalement sur une notion de corruption de type. Intuitivement et pour un type donné, on distingue trois niveaux possibles de corruption de ce type que nous allons maintenant décrire.

1^{er} niveau : l'absence de corruption

Un type du λ -calcul simplement typé est un type de λ_ε et c'est un type qui n'est pas corrompu. Les termes de ce type ne peuvent pas lever d'exceptions à l'exécution. En particulier, un terme de type nat est un terme qui calcule effectivement un entier naturel, comme

- $S (S 0)$
- $(\lambda x. S x) (S (S 0))$

On notera cependant qu'un terme de type non corrompu peut quand même utiliser des exceptions, soit parce qu'elles sont rattrapées au cours du calcul, comme dans le terme

- $\text{try} (\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto 0$

qui est de type nat , soit parce qu'elles sont effacées au cours du calcul :

- $(\lambda x. 0) (\text{raise } \varepsilon)$

qui est également de type nat .

2^{ème} niveau : la corruption à la racine, le type $A \uplus \{\varepsilon\}$

Pour tout type A et pour tout nom d'exceptions ε , on introduit le type $A \uplus \{\varepsilon\}$, dit type *union*. Il s'agit du type des termes qui soit se réduisent sur une valeur de type A , soit lèvent l'exception ε (et on rappellera que lever une exception ε est se réduire vers le terme $\text{raise } \varepsilon$). Cette construction est donc une simple union bien qu'hétérogène et dissymétrique.

Exemple 4.1

Les termes suivants sont de type $\text{nat} \uplus \{\varepsilon\}$:

- $\text{raise } \varepsilon$
- $S (S 0)$
- $(\lambda x. \text{raise } \varepsilon) 0$
- $\text{try} (\text{raise } \varepsilon') \text{ with } \varepsilon' \mapsto \text{raise } \varepsilon$

La construction de type $A \uplus \{\varepsilon\}$ est essentielle pour typer la construction try . En effet, dans l'expression $\text{try } t \text{ with } \varepsilon \mapsto u$, l'exception ε est capturée si et seulement si t la lève. Ainsi la règle de typage de la construction try dans λ_ε est :

$$\frac{\Gamma \vdash t : A \uplus \{\varepsilon\} \quad \Gamma \vdash u : A}{\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A}$$

On notera que tout terme de type A est également un terme de type $A \uplus \{\varepsilon\}$. Il n'y a donc pas d'unicité de type dans le calcul. Cette inclusion de type est modélisée dans le calcul par l'introduction de la relation de sous-typage $\leq$ dont le premier axiome est :

$$A \leq A \uplus \{\varepsilon\}$$

De plus, la relation $\leq$ est réflexive, transitive et au regard du type des fonctions, contravariante à gauche et covariante à droite.

Enfin, le terme $\text{raise } \varepsilon$ a le type $A \uplus \{\varepsilon\}$ pour tout type A et sa règle de typage est donc

$$\overline{\Gamma \vdash \text{raise } \varepsilon : A \uplus \{\varepsilon\}}$$

3^{ème} niveau : la corruption généralisée, le type $A^{\{\varepsilon\}}$

La construction $A \uplus \{\varepsilon\}$ ne permet pas de donner un type à une valeur telle que $S \text{ (raise } \varepsilon)$, qui ne se réduit ni sur un entier, ni sur une exception. Dans un tel terme, l'exception ε est en quelque sorte enfouie sous la fonction successeur.

Notons par ailleurs que ce problème d'enfouissement n'est pas spécifique au type des entiers mais apparaît également avec les fonctions. En effet, considérons la fonction $\lambda x. \lambda y. x$ de type $A \rightarrow (B \rightarrow A)$. Appliquée à une exception, cette fonction va enfouir cette dernière sous une abstraction. En effet $(\lambda x. \lambda y. x) \text{ (raise } \varepsilon)$ se réduit sur $\lambda y. \text{raise } \varepsilon$ qui n'est ni une fonction de type $B \rightarrow A$, ni ne lève l'exception ε .

Pour typer les termes où les exceptions sont enfouies, on introduit la construction de type $A^{\{\varepsilon\}}$, que l'on nommera type A corrompu par ε . Intuitivement, cette construction désigne le type des termes de type A dans lesquels on a remplacé un certain nombre de sous-termes (éventuellement aucun) par le terme $\text{raise } \varepsilon$. Ainsi le terme $S \text{ (raise } \varepsilon)$ est-il de type $\text{nat}^{\{\varepsilon\}}$. La corruption est la construction générale pour le typage des exceptions au sens où tout terme (bien typé) qui mentionne une exception ε est de type $A^{\{\varepsilon\}}$ pour un certain A (on respecte ainsi le principe 2 donnée à la section 4.1.1). Notons également qu'une caractérisation équivalente reliant la corruption à l'union est qu'alors que les termes de type $A \uplus \{\varepsilon\}$ sont les termes pouvant lever l'exception ε à la racine, ceux de type $A^{\{\varepsilon\}}$ sont ceux qui peuvent lever l'exception ε dans n'importe quel contexte d'évaluation valable pour un terme de type A (cette caractérisation sera d'ailleurs à la base de l'interprétation de l'union et la corruption dans le modèle de réalisabilité du chapitre 7).

Exemple 4.2

Les termes suivants sont de type $\text{nat}^{\{\varepsilon\}}$:

- $S \text{ (} S \text{ 0)}$
- $\text{raise } \varepsilon$
- $S \text{ (} S \text{ raise } \varepsilon)$

Les termes suivants sont de type $(\text{nat} \rightarrow \text{nat})^{\{\varepsilon\}}$:

- $\lambda x. S \text{ 0}$
- $\text{raise } \varepsilon$
- $\lambda x. \text{raise } \varepsilon$
- $\lambda x. S \text{ (raise } \varepsilon)$
- $(\lambda x. \lambda y. S \text{ (} S \text{ (} x))) \text{ (raise } \varepsilon)$

Les termes de type A sont en particulier des termes de types $A^{\{\varepsilon\}}$ (c'est le cas où on ne remplace aucun sous-terme par $\text{raise } \varepsilon$). De plus, les termes de type $A \uplus \{\varepsilon\}$ sont également des termes de type $A^{\{\varepsilon\}}$. Ce sont les termes où on a remplacé un ou plusieurs sous-termes par $\text{raise } \varepsilon$ de sorte que le terme entier se réduise sur $\text{raise } \varepsilon$. Ces inclusions de types sont modélisées dans le calcul par l'ajout de l'axiome de sous-typage

$$A \uplus \{\varepsilon\} \leq A^{\{\varepsilon\}}.$$

L'axiome déjà introduit $A \leq A \uplus \{\varepsilon\}$ et la transitivité du sous-typage permettent alors en particulier d'obtenir $A \leq A^{\{\varepsilon\}}$.

Dans le calcul, le sens de la construction de corruption lui est donné par le fait qu'elle vérifie l'axiome de sous-typage suivant :

$$(A \rightarrow B)^{\{\varepsilon\}} \leq A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}} \quad (1)$$

Une conséquence directe de cet axiome, et l'idée fondamentale du système de type, est que pour tout type A et B , n'importe quelle fonction de type $A \rightarrow B$ peut toujours être utilisée en présence d'exceptions. Plus précisément, on a la dérivation de sous-typage suivante :

$$A \rightarrow B \leq (A \rightarrow B) \uplus \{\varepsilon\} \leq (A \rightarrow B)^{\{\varepsilon\}} \leq A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}$$

Autrement dit, une fonction (et en particulier une fonction qui n'a pas été écrite de façon à gérer les exceptions) peut toujours être appliquée à un argument mentionnant l'exception ε . Bien sûr, dans ce cas, le résultat de l'application mentionnera probablement également cette exception (cette fonction ne sachant pas à priori gérer l'exception ε , la seule façon pour que le résultat ne mentionne pas ε est que la fonction efface son argument). Notons qu'apparaît ici plus clairement l'intérêt de la corruption, un tel sous-typage ne serait en effet pas valable pour la construction d'union, puisque l'on ne sait pas ce que fait la fonction de type $A \rightarrow B$ de son argument. Donc, en particulier, si son argument contient une exception, il est tout à fait possible que la fonction enfouisse cette exception (sous des abstractions, des successeurs) et qu'elle ne soit donc pas de type $A \uplus \{\varepsilon\} \rightarrow B \uplus \{\varepsilon\}$. Remarquons en revanche qu'une fonction de type $A \rightarrow B$ est une fonction de type $A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}$ et donc, par contravariance à gauche des types flèches, est également une fonction de type $A \uplus \{\varepsilon\} \rightarrow B^{\{\varepsilon\}}$.

De plus, la règle de sous-typage (1) permet également d'utiliser le terme `raise  $\varepsilon$` comme une fonction. En effet, pour tout type A et B , `raise  $\varepsilon$` a le type $(A \rightarrow B) \uplus \{\varepsilon\}$ et donc, en réutilisant la dérivation ci-dessus, a aussi le type $A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}$.

Cette règle de sous-typage permet au système de type d'assurer le premier principe donné en section 4.1.1, c'est à dire permet l'utilisation des exceptions à la place de n'importe quel autre terme (cette affirmation sera justifiée par le théorème de corruption (théorème 4.50) dans la section 4.4.5). En effet, comme on vient de le montrer, `raise  $\varepsilon$` peut toujours être utilisé aussi bien comme une fonction que comme un argument de fonction.

Le fait que toute fonction de type $A \rightarrow B$ puisse aussi être vue comme une fonction de type $A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}$ est une propriété de sous-typage. Elle ne dépend donc pas du code de la fonction, apportant ainsi une forme de modularité au typage des exceptions dans λ_ε (respect du principe 4 de la section 4.1.1). D'autre part, ceci permet le typage des types de données (les entiers naturels) en présence d'exceptions. En effet, les constructeurs de ces types de données (le successeur pour les entiers) sont des fonctions primitives et n'ont donc pas de définition. Ainsi, grâce aux règles de sous-typage relatives à la corruption, le successeur S qui est de type $\text{nat} \rightarrow \text{nat}$ est également de type $\text{nat}^{\{\varepsilon\}} \rightarrow \text{nat}^{\{\varepsilon\}}$. Et donc finalement, le terme S (`raise  $\varepsilon$`) est bien de type $\text{nat}^{\{\varepsilon\}}$ comme nous l'avions anticipé précédemment.

Avec les règles que nous avons présenté jusqu'à présent, le type $A^{\{\varepsilon\}}$ ne correspond pas exactement à la définition intuitive que nous en avons donnée, à savoir celui des termes de type A pour lesquels on peut avoir remplacé certains sous-termes par le terme `raise  $\varepsilon$` . En effet, considérons le terme $\lambda x. \text{raise } \varepsilon$. Ce terme est de type $B \rightarrow A^{\{\varepsilon\}}$ mais n'est pas (pour l'instant) de type $(B \rightarrow A)^{\{\varepsilon\}}$. Ceci est en fait révélateur d'un problème plus sérieux :

l'absence de préservation du typage par réduction, ainsi que le montre l'exemple qui suit. Considérons le terme

$$t = \lambda x. \lambda y. x.$$

Ce terme est de type $A \rightarrow (B \rightarrow A)$ et donc aussi de type $A^{\{\varepsilon\}} \rightarrow (B \rightarrow A)^{\{\varepsilon\}}$. Par conséquent, le terme t (raise ε) est de type $(B \rightarrow A)^{\{\varepsilon\}}$ par simple application de la règle de typage de l'application. Or t (raise ε) se réduit sur $\lambda y. \text{raise } \varepsilon$ qui, comme nous l'avons dit n'est pas de type $(B \rightarrow A)^{\{\varepsilon\}}$. Pour récupérer la propriété de préservation du typage par réduction, il faut dans cet exemple que

$$B \rightarrow A^{\{\varepsilon\}} \leq (B \rightarrow A)^{\{\varepsilon\}}$$

On règle ce problème en ajoutant au sous-typage le nouvel axiome suivant :

$$A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}} \leq (A \rightarrow B)^{\{\varepsilon\}}$$

Avec ce nouvel axiome,

$$B \rightarrow A^{\{\varepsilon\}} \leq (B \rightarrow A^{\{\varepsilon\}})^{\{\varepsilon\}} \leq B^{\{\varepsilon\}} \rightarrow (A^{\{\varepsilon\}})^{\{\varepsilon\}} \leq B^{\{\varepsilon\}} \rightarrow A^{\{\varepsilon\}} \leq (B \rightarrow A)^{\{\varepsilon\}}$$

Notons par ailleurs que cette dérivation utilise la règle non encore présentée mais intuitivement simple :

$$(B^{\{\varepsilon\}})^{\{\varepsilon\}} \leq B^{\{\varepsilon\}}$$

Pour résumer, si on note $\doteq$ l'égalité de sous-typage, l'égalité suivante est un axiome du sous-typage :

$$(A \rightarrow B)^{\{\varepsilon\}} \doteq A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}$$

Nous verrons dans la section 5.1 que cette égalité change profondément le sens du type des flèches et qu'en particulier la flèche usuelle de la réalisabilité n'est pas une sémantique valide pour la flèche de λ_ε .

4.1.3 Des ensembles d'exceptions

Nous n'avons jusqu'à présent utilisé qu'un seul nom d'exception ε avec les constructions $A \uplus \{\varepsilon\}$ et $A^{\{\varepsilon\}}$. Toutefois, afin de gérer le fait qu'un terme peut mentionner plusieurs exceptions, ces deux constructions manipulent en réalité des ensembles d'exceptions. On parlera donc dorénavant des constructions $A \uplus \Delta$ et A^Δ où Δ est un ensemble de noms d'exceptions.

L'utilisation d'ensembles d'exceptions nécessite d'effectuer un certain nombre d'identifications de types au moyen des équations (de sous-typage) suivantes :

$$\begin{array}{ll} (A \uplus \Delta') \uplus \Delta & \doteq A \uplus (\Delta \cup \Delta') \\ (A^{\Delta'})^\Delta & \doteq A^{(\Delta \cup \Delta')} \\ A \uplus \emptyset & \doteq A \\ A^\emptyset & \doteq A \end{array}$$

4.1.4 Quel type pour `rec` ?

Le calcul λ_ε est muni d'entiers primitifs, construits à partir des deux constructeurs habituels, 0 et la fonction successeur S , avec leurs règles de typage usuelles :

$$\overline{\Gamma \vdash 0 : \text{nat}} \qquad \overline{\Gamma \vdash S : \text{nat} \rightarrow \text{nat}}$$

Se pose alors la question du typage de l'opération de récursion `rec` de λ_ε . Cet opérateur a les réductions du récursur du système T de Gödel [GTL89] (plus le fait de propager les exceptions). Le type usuel de ce récursur est le suivant :

$$\overline{\Gamma \vdash \text{rec} : A \rightarrow (\text{nat} \rightarrow A \rightarrow A) \rightarrow \text{nat} \rightarrow A}$$

Munis de ce type et grâce au mécanisme de la corruption, on peut alors montrer que l'opérateur `rec a`, par sous-typage, le type suivant :

$$\text{rec} : A^\Delta \rightarrow (\text{nat}^\Delta \rightarrow A^\Delta \rightarrow A^\Delta) \rightarrow \text{nat}^\Delta \rightarrow A^\Delta$$

Autrement dit, comme pour toute autre fonction, cet opérateur peut être utilisé en présence d'exceptions. Cependant, parce qu'il est obtenu par corruption qui est un mécanisme général, ce type ne rend pas compte suffisamment précisément du comportement calculatoire de `rec`. On peut en particulier remarquer deux sources d'imprécision :

Première source d'imprécision. Considérons la fonction suivante :

$$\text{eval} \equiv \lambda n. (\text{rec } (\lambda a. a) (\lambda x. \lambda r. \lambda a. r (S a)) n) 0$$

Appliquée à un entier naturel (donc intuitivement à un terme de la forme $S^n 0$), cette fonction parcourt ce dernier (le déconstruit) en le reconstruisant petit à petit dans un accumulateur (la variable a dans le code ci-dessus). Autrement dit, sur un entier, cette fonction est l'identité, mais codée de façon à recopier son entrée dans un accumulateur qui est renvoyé en fin de calcul. Cette fonction est de type $\text{nat} \rightarrow \text{nat}$ et donc également de type $\text{nat}^{\{\varepsilon\}} \rightarrow \text{nat}^{\{\varepsilon\}}$. On peut donc l'appliquer à un entier corrompu, disons $S (S (\text{raise } \varepsilon))$ et le résultat, $\text{eval } (S (S (\text{raise } \varepsilon)))$ est de type $\text{nat}^{\{\varepsilon\}}$. Cependant $\text{eval } (S (S (\text{raise } \varepsilon)))$ se réduit sur `raise  $\varepsilon$` , et plus généralement, on peut vérifier que si l'argument d' eval est n , un entier potentiellement corrompu par ε (c'est à dire un terme de type $\text{nat}^{\{\varepsilon\}}$), alors $\text{eval } n$ renverra soit n dans le cas où c'est un entier licite, soit directement le terme `raise  $\varepsilon$` si n le mentionne effectivement. Autrement dit, le résultat sera toujours un terme de type $\text{nat} \uplus \{\varepsilon\}$ bien que le système de type lui donne le type moins précis $\text{nat}^{\{\varepsilon\}}$. Pour régler ce problème, on va donner à l'opérateur de récursion le type plus précis suivant :

$$\overline{\Gamma \vdash \text{rec} : A \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow A \uplus \Delta \rightarrow A \uplus \Delta) \rightarrow \text{nat}^\Delta \rightarrow A \uplus \Delta}$$

Dans ce type, Δ représente un ensemble de noms d'exceptions quelconque et cette règle est donc un schéma de règle de typage dans lequel on retrouve le typage habituel du récursur en prenant $\Delta = \emptyset$. Nous verrons dans la section 4.3 que grâce à ce typage du récursur le terme eval a le type $\text{nat}^\Delta \rightarrow \text{nat} \uplus \Delta$ (quel que soit l'ensemble de nom d'exceptions Δ). Une telle fonction est particulièrement utile dans λ_ε puisqu'elle permet de capturer une exception enfouie dans un entier naturel.

Seconde source d'imprécision. Rappelons la règle de réduction (*recr*) (voir définition 3.5) :

$$\text{rec } z \ f \ (\text{raise } \varepsilon) \rightarrow \text{raise } \varepsilon$$

Si l'argument principal de *rec* (celui sur lequel la récursion est effectuée) est une exception, alors cette exception est simplement propagée. Ce comportement calculatoire est orthogonal à celui que nous évoquons dans le paragraphe précédent et le type pour *rec* qui y est proposé n'en tient pas compte. Pour le prendre en compte, on doit donner au récursur la règle de typage suivante :

$$\frac{}{\Gamma \vdash \text{rec} : A \rightarrow (\text{nat} \rightarrow A \rightarrow A) \rightarrow \text{nat} \uplus \Delta \rightarrow A \uplus \Delta}$$

Et ici encore, Δ représente un ensemble de noms d'exceptions quelconque et cette règle est donc un schéma de règle de typage dans lequel on retrouve le typage habituel du récursur en prenant $\Delta = \emptyset$. Alors, en combinant ce typage à celui du paragraphe précédent, on obtient la règle finale de typage du récursur dans le calcul λ_ε , à savoir :

$$\frac{}{\Gamma \vdash \text{rec} : A \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow A \uplus \Delta \rightarrow A \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow A \uplus (\Delta \cup \Delta')}$$

4.2 Typage et sous-typage

Cette section présente formellement le système de type du calcul λ_ε .

4.2.1 Syntaxe des types

La syntaxe des types de λ_ε est construite à partir de celles des types du λ -calcul simplement typé en y ajoutant les deux constructions de types que nous avons introduites dans la section précédente, l'union et la corruption.

Définition 4.3 (Types de λ_ε)

Un type de λ_ε est défini comme une production (finie) de la grammaire suivante :

$$\begin{array}{ll} A ::= & \text{nat} \quad (\text{entiers naturels}) \\ & | \quad A \rightarrow A \quad (\text{fonctions}) \\ & | \quad A \uplus \Delta \quad (\text{union}) \\ & | \quad A^\Delta \quad (\text{corruption}) \end{array}$$

Dans les constructions $A \uplus \Delta$ et A^Δ , Δ représente un ensemble fini de noms d'exceptions (et donc $\Delta \subseteq \mathcal{E}$). Les précédences de ces deux constructions sont supérieures à celle du type flèche, si bien que $A \rightarrow B^\Delta$ doit se comprendre comme $A \rightarrow (B^\Delta)$. De plus, comme il est d'usage, la flèche est associative à droite : le type $A \rightarrow B \rightarrow C$ doit se comprendre comme $A \rightarrow (B \rightarrow C)$.

4.2.2 Jugement de typage

Définition 4.4 (Environnement de typage)

Un environnement de typage Γ est un ensemble fini de déclarations ayant la forme

$$\Gamma \equiv x_1 : A_1, \dots, x_n : A_n,$$

où $x_1, \dots, x_n$ sont des variables de termes deux à deux distinctes et $A_1, \dots, A_n$ sont des types.

Définition 4.5 (Jugement de typage)

Le système de types de λ_ϵ est défini par le jugement de typage

$$\Gamma \vdash t : A$$

qui se lit : « Dans l'environnement de typage Γ , le terme t a le type A ». Ce jugement est défini inductivement par les règles de déduction de la figure 4.1.

Règles du λ -calcul simplement typé

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} (t\text{-ax})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} (t\text{-abs}) \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B} (t\text{-app})$$

Règles de typage des entiers naturels

$$\frac{}{\Gamma \vdash 0 : \text{nat}} (t\text{-zero}) \quad \frac{}{\Gamma \vdash S : \text{nat} \rightarrow \text{nat}} (t\text{-succ})$$

$$\frac{}{\Gamma \vdash \text{rec} : A \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow A \uplus \Delta \rightarrow A \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow A \uplus (\Delta \cup \Delta')} (t\text{-rec})$$

Règles spécifiques au typage des exceptions

$$\frac{}{\Gamma \vdash \text{raise } \epsilon : A \uplus \{\epsilon\}} (t\text{-raise}) \quad \frac{\Gamma \vdash t : A \uplus \{\epsilon\} \quad \Gamma \vdash u : A}{\Gamma \vdash \text{try } t \text{ with } \epsilon \mapsto u : A} (t\text{-try})$$

$$\frac{\Gamma \vdash t : A \quad A \leq B}{\Gamma \vdash t : B} (t\text{-subtyp})$$

FIG. 4.1 – Règles de typage du langage λ_ϵ .

Les règles du λ -calcul simplement typé sont reprises dans λ_ε sans aucune modification. Ainsi, les règles de typage pour la λ -abstraction et l'application sont les règles usuelles. Les règles de typage des constructeurs pour les entiers naturels sont également standards et seule la règle pour le typage du récursur prend en compte les exceptions (on se reportera à la section 4.1.4 pour une discussion détaillée et une justification du typage de cet opérateur). Notons par ailleurs que, faute de polymorphisme, les règles de typage pour le *raise* et le récursur sont en fait des schémas de règles de typage où A représente un type quelconque et Δ un ensemble de noms d'exceptions quelconque. Les autres règles sont spécifiques à la gestion des exceptions :

- (*t-raise*) Rappelons que le type $A \uplus \Delta$ contient les termes qui sont soit de type A , soit se réduisent sur une exception dont le nom appartient à Δ . En particulier, *raise* ε se réduit sur lui-même et est donc de type $A \uplus \{\varepsilon\}$ quel que soit le type A .
- (*t-try*) Le corps d'un *try* peut lever l'exception ε , auquel cas celle-ci est rattrapée comme l'indique le type de résultat où ε n'apparaît pas. Autrement, le corps et le gestionnaire du *try* doivent avoir le même type A , qui est alors le type du *try*.
- (*t-subtyp*) Comme expliqué précédemment, le typage des exceptions et en particulier la corruption repose fortement sur une notion de sous-typage. On remarquera d'ailleurs qu'aucune règle de typage ne permet d'introduire de type corrompu (seul le récursur permet d'éliminer la corruption sur les entiers naturels). La présentation de cette relation de sous-typage fait l'objet de la prochaine section.

Lemme 4.6 Si t et u sont deux termes, A et B deux types, Γ un environnement de typage et x une variable de terme tel que $\Gamma, x : A \vdash t : B$ et $\Gamma \vdash u : A$, alors

$$\Gamma \vdash t\{x := u\} : B$$

Preuve :

On prouve en fait une légère généralisation de ce lemme pour traiter le cas où la règle de l'abstraction rajoute dans sa prémisse une variable à l'environnement de typage et on prouve donc que si $\Gamma, x : A, \Omega \vdash t : B$ et $\Gamma, \Omega \vdash u : A$ alors $\Gamma, \Omega \vdash t\{x := u\} : B$. La preuve est une simple induction sur $\Gamma, x : A, \Omega \vdash t : B$ dont aucun cas ne présente de difficultés.

Lemme 4.7 Si t est un terme, x est une variable de terme, A et B sont des types et Γ est un environnement de typage tel que $\Gamma, x : A \vdash t : B$ et $x \notin \text{FV}(t)$, alors

$$\Gamma \vdash t : B$$

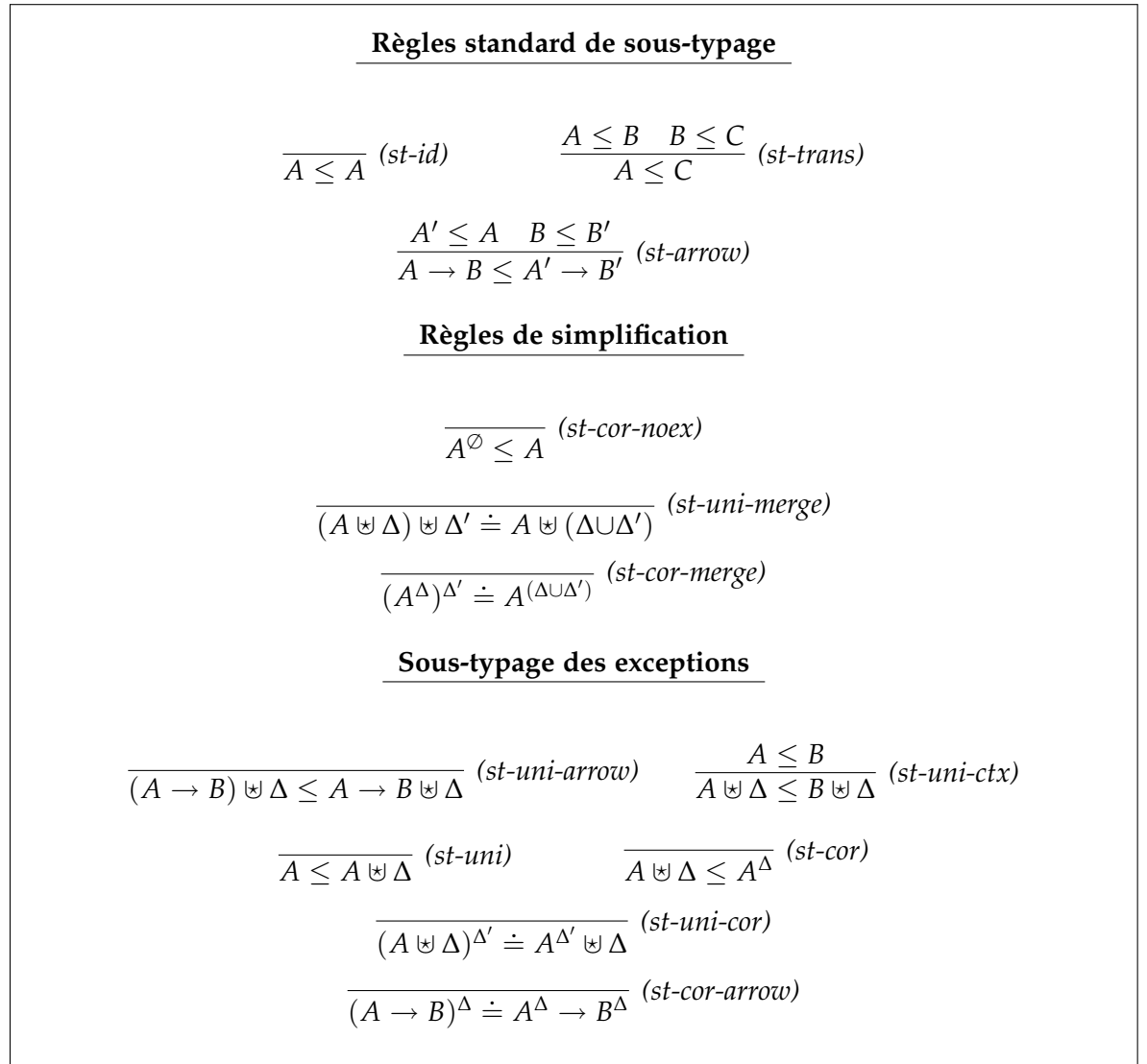
Preuve :

La preuve de ce lemme est une simple induction sur la dérivation de $\Gamma, x : A \vdash t : B$.

4.2.3 Jugement de sous-typage

Définition 4.8 (Relation de sous-typage)

La relation de sous-typage entre deux types A et B est notée $A \leq B$. À partir de cette relation, on définit la notion d'égalité de sous-typage $A \doteq B$ comme la composition des deux règles $A \leq B$ et $B \leq A$. La relation de sous-typage est définie comme la plus petite relation dérivable en utilisant les règles de déduction de la figure 4.2.

FIG. 4.2 – Règles de sous-typage du langage λ_e .

Dans la figure 4.2, le sens de la notation $A \doteq B$ dépend de sa position :

- En position de prémisses, elle figure pour deux prémisses :

$$\frac{A \dot{=} B}{C} \quad \text{est une notation pour} \quad \frac{A \leq B \quad B \leq A}{C}$$

- En conclusion, elle exprime deux règles :

$$\frac{C}{A \dot{=} B} \quad \text{est une notation pour} \quad \frac{C}{A \leq B} \quad \text{et} \quad \frac{C}{B \leq A}$$

Cette relation est une relation de sous-typage au sens usuel du terme : elle est réflexive (*st-id*), transitive (*st-trans*) et les flèches y sont contravariantes à gauche et covariantes à droite (*st-arrow*). Les règles dites « de simplification » dans la figure 4.2 sont les règles utilisées pour manipuler les ensembles d'exceptions dans les constructions $A \uplus \Delta$ et A^Δ . Pour les autres règles :

(st-uni-arrow) Cette règle rend compte du comportement calculatoire suivant : Un terme t de type $(A \rightarrow B) \uplus \Delta$ est soit un terme de type $A \rightarrow B$, soit un terme se réduisant sur une exception de Δ . Et une exception est une fonction qui jette ses arguments. On peut donc toujours appliquer t à un terme u de type A . Le résultat sera soit un terme de type B , soit une exception de Δ si t en était déjà une.

Autrement dit, cette règle de sous-typage rend compte du comportement calculatoire induit par la règle de réduction (*rapp*) (définition 3.5) en permettant, quels que soient les types A et B , de dériver le jugement de typage suivant :

$$\frac{\vdots}{\Gamma \vdash \text{raise } \varepsilon : A \rightarrow B \uplus \{\varepsilon\}}$$

(st-uni-ctx) Cette règle exprime la stabilité du sous-typage par passage à l'union. Le sous-typage est également stable par passage à la corruption, mais cette dernière propriété est admissible comme le montre le lemme 4.10.

(st-uni) Cette règle modèle l'inclusion des termes de type A dans ceux de type $A \uplus \Delta$ (voir section 4.1.2 pour plus de détails sur cet inclusion).

(st-cor) Cette règle modèle l'inclusion des termes de type $A \uplus \Delta$ dans ceux de type A^Δ (voir section 4.1.2 pour plus de détails sur cet inclusion).

(st-uni-cor) Cette règle exprime la commutation des opérations d'union et de corruption. On peut noter que, comme le montre le lemme 4.11, la règle

$$\overline{(A \uplus \Delta')^\Delta \dot{=} A^\Delta \uplus (\Delta' - \Delta)}$$

qui est plus fine que la règle (*st-uni-cor*), est dérivable.

(st-cor-arrow) Cette règle est la règle fondamentale de la notion de corruption. On se reportera à la section 4.1.2 pour une discussion sur l'utilité et la justification de cette règle.

Règles dérivables. Comme pour la construction de corruption, l'union d'un type avec un ensemble vide d'exceptions est l'identité :

Lemme 4.9 Quel que soit le type A ,

$$A \uplus \emptyset \leq A$$

Preuve :

Cette propriété est évidente puisque $A \uplus \emptyset \leq A^\emptyset \leq A$.

D'autre part, le sous-typage est stable par corruption :

Lemme 4.10 Si A et B sont deux types tel que $A \leq B$, alors pour tout ensemble de noms d'exceptions Δ ,

$$A^\Delta \leq B^\Delta$$

Preuve :

La preuve est une simple induction sur la dérivation de $A \leq B$. Tous les cas se résolvent facilement puisque la construction de corruption commute avec toutes les constructions de type. Par exemple, si on prend le cas de la règle (*st-uni-arrow*), on veut montrer que

$$((A \rightarrow B) \uplus \Delta')^\Delta \leq (A \rightarrow B \uplus \Delta')^\Delta.$$

Or, en utilisant la règle (*st-uni-arrow*),

$$(A^\Delta \rightarrow B^\Delta) \uplus \Delta' \leq A^\Delta \rightarrow B^\Delta \uplus \Delta'$$

et on conclut avec le fait que

$$(A^\Delta \rightarrow B^\Delta) \uplus \Delta' \doteq ((A \rightarrow B) \uplus \Delta')^\Delta \quad \text{et} \quad A^\Delta \rightarrow B^\Delta \uplus \Delta' \doteq (A \rightarrow B \uplus \Delta')^\Delta.$$

On peut également prouver les deux égalités de sous-typage suivantes :

Lemme 4.11 Si A est un type et Δ et Δ' deux ensembles de noms d'exceptions, alors

$$(A \uplus \Delta')^\Delta \doteq A^\Delta \uplus (\Delta' - \Delta)$$

Preuve :

On montre les deux sens de l'égalité :

- Montrons que $A^\Delta \uplus (\Delta' - \Delta) \leq (A \uplus \Delta')^\Delta$:
Par (*st-uni*), $A^\Delta \uplus (\Delta' - \Delta) \leq (A^\Delta \uplus (\Delta' - \Delta)) \uplus \Delta$ et par (*st-uni-merge*) $(A^\Delta \uplus (\Delta' - \Delta)) \uplus \Delta \leq A^\Delta \uplus \Delta'$. On conclut alors en appliquant (*st-uni-cor*).
- Montrons que $(A \uplus \Delta')^\Delta \leq A^\Delta \uplus (\Delta' - \Delta)$:
Avec (*st-uni-cor*), $(A \uplus \Delta')^\Delta \doteq A^\Delta \uplus \Delta'$. Alors en utilisant (*st-uni-merge*), $A^\Delta \uplus \Delta' \doteq (A^\Delta \uplus (\Delta' - \Delta)) \uplus \Delta$. On utilise alors (*st-cor*) pour obtenir $(A^\Delta \uplus (\Delta' - \Delta)) \uplus \Delta \leq (A^\Delta \uplus (\Delta' - \Delta))^\Delta$. Mais alors $(A^\Delta \uplus (\Delta' - \Delta))^\Delta \doteq (A^\Delta)^\Delta \uplus (\Delta' - \Delta) \doteq A^\Delta \uplus (\Delta' - \Delta)$ (par (*st-uni-cor*) et (*st-cor-merge*)).

Lemme 4.12 Si A et B sont des types et Δ un ensemble de noms d'exceptions, alors

$$(A \rightarrow B \uplus \Delta) \uplus \Delta \doteq A \rightarrow B \uplus \Delta$$

Preuve :

Le fait que $A \rightarrow B \uplus \Delta \leq (A \rightarrow B \uplus \Delta) \uplus \Delta$ est une simple instance de la règle (*st-uni*). Pour l'autre sens de l'égalité, avec la règle (*st-uni-arrow*), $(A \rightarrow B \uplus \Delta) \uplus \Delta \leq A \rightarrow (B \uplus \Delta) \uplus \Delta$ et avec (*st-uni-merge*) et (*st-arrow*), $A \rightarrow (B \uplus \Delta) \uplus \Delta \doteq A \rightarrow B \uplus \Delta$.

4.3 Exemples

On va commencer par reprendre les exemples de la section 3.3 et en donner leur typage dans λ_ϵ .

Le prédécesseur. Le type de *pred* rend compte de son comportement calculatoire, à savoir que si on l'applique à un entier, il renverra soit un entier, soit un exception :

$$\begin{aligned} \text{pred} & : \text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\} \\ & \equiv \text{rec } (\text{raise argis0}) (\lambda n. \lambda r. n) \end{aligned}$$

Le typage de cette fonction utilise le fait que *rec* a pour type

$$\text{nat} \uplus \{\text{argis0}\} \rightarrow (\text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\} \rightarrow \text{nat} \uplus \{\text{argis0}\}) \rightarrow \text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\}$$

en prenant $A = \text{nat} \uplus \{\text{argis0}\}$ et $\Delta = \Delta' = \emptyset$ dans la règle (*t-rec*).

Le système de types capture correctement le fait que la fonction *safe_pred*, elle, ne va jamais lever d'exception :

$$\begin{aligned} \text{safe_pred} & : \text{nat} \rightarrow \text{nat} \\ & \equiv \lambda n. \text{try pred } n \text{ with argis0} \mapsto 0 \end{aligned}$$

La soustraction. La fonction *pred* a le type $\text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\}$ et donc par sous-typage a aussi le type $\text{nat}^{\{\text{argis0}\}} \rightarrow \text{nat}^{\{\text{argis0}\}}$. En effet,

$$\begin{aligned} \text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\} & \leq (\text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\})^{\{\text{argis0}\}} \\ & \leq \text{nat}^{\{\text{argis0}\}} \rightarrow (\text{nat} \uplus \{\text{argis0}\})^{\{\text{argis0}\}} \\ & \leq \text{nat}^{\{\text{argis0}\}} \rightarrow \text{nat}^{\{\text{argis0}\}} \end{aligned}$$

En conséquence, $\lambda n. \lambda r. \text{pred } r$ a le type $\text{nat} \rightarrow \text{nat}^{\{\text{argis0}\}} \rightarrow \text{nat}^{\{\text{argis0}\}}$. Par ailleurs, *rec* a le type

$$\text{nat}^{\{\text{argis0}\}} \rightarrow (\text{nat} \rightarrow \text{nat}^{\{\text{argis0}\}} \rightarrow \text{nat}^{\{\text{argis0}\}}) \rightarrow \text{nat} \rightarrow \text{nat}^{\{\text{argis0}\}}$$

en prenant $A = \text{nat}^{\{\text{argis0}\}}$ et $\Delta = \Delta' = \emptyset$ dans la règle (*t-rec*). Il s'ensuit que l'on a le typage suivant pour la fonction *minus* :

$$\begin{aligned} \text{minus} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}^{\{\text{argis0}\}} \\ & \equiv \lambda a. \lambda b. \text{rec } a \ (\lambda n. \lambda r. \text{pred } r) \ b \end{aligned}$$

Remarquons que le type du résultat de cette fonction *minus* est corrompu, alors même que l'on peut facilement vérifier que cette fonction renverra toujours un terme de type $\text{nat} \uplus \{\text{argis0}\}$. Ceci est une conséquence du fait que le type que l'on a utilisé pour *pred* ne rend pas compte (une fois corrompu) du fait que cette fonction n'enfouit jamais une exception. On peut cependant typer plus finement la fonction *pred* puisque quel que soit l'ensemble de noms d'exceptions Δ on peut vérifier (en prenant Δ' non vide dans la règle de typage (*t-rec*)) que

$$\text{pred} : \text{nat} \uplus \Delta \rightarrow \text{nat} \uplus (\Delta \cup \{\text{argis0}\})$$

En utilisant ce type pour la fonction *pred* et en y instanciant Δ par $\{\text{argis0}\}$, on vérifie que

$$\text{minus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\}$$

On retrouve ici un argument de modularité du système de types. Même si *pred* est déclarée dans un module séparé avec un type qui n'est pas son type le plus précis, le système fonctionne correctement (mais en propageant la perte de précision au fonction utilisant *pred*). De plus, on verra un peu plus loin (page 48) que même si l'on utilise pour *pred* le type $\text{nat} \rightarrow \text{nat} \uplus \text{argis0}$, on peut tout de même écrire une variante de *minus* ayant le type $\text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\}$.

L'addition. La fonction *add* n'utilise pas les exceptions et son type est celui qu'elle a dans le Système T, à savoir :

$$\begin{aligned} \text{add} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \\ & \equiv \lambda a. \lambda b. \text{rec } b \ (\lambda n. \lambda r. S \ r) \ a \end{aligned}$$

Le mécanisme de la corruption nous permet toutefois d'utiliser l'addition en présence d'exceptions (et ce implicitement) et par exemple le terme *add* 3 (*minus* 1 3) a le type $\text{nat}^{\{\text{argis0}\}}$. Ce terme se réduit d'ailleurs sur *S* (*S* (*S* (*raise argis0*))), terme qui correspond bien à l'intuition d'un entier corrompu.

La fonction « eval ». Dans la section 4.1.4, nous avons utilisé la fonction *eval* pour justifier notre typage de l'opérateur de récursion et avons affirmé que l'on avait :

$$\begin{aligned} \text{eval} & : \text{nat}^\Delta \rightarrow \text{nat} \uplus \Delta \\ & \equiv \lambda n. (\text{rec } (\lambda a. a) \ (\lambda _ . \lambda r. \lambda \text{acc}. r \ (S \ \text{acc})) \ n) \ 0 \end{aligned}$$

Pour dériver ce type, on utilise le fait que le récursur a le type

$$\begin{aligned} \text{rec} & : (\text{nat} \rightarrow \text{nat} \uplus \Delta) \uplus \Delta \rightarrow \left(\text{nat}^\Delta \rightarrow (\text{nat} \rightarrow \text{nat} \uplus \Delta) \uplus \Delta \rightarrow (\text{nat} \rightarrow \text{nat} \uplus \Delta) \uplus \Delta \right) \\ & \rightarrow \text{nat}^\Delta \rightarrow (\text{nat} \rightarrow \text{nat} \uplus \Delta) \uplus \Delta \end{aligned}$$

en prenant $A = \text{nat} \rightarrow \text{nat} \uplus \Delta$, $\Delta = \Delta$ et $\Delta' = \emptyset$ dans la règle (*t-rec*). En remarquant de plus (lemme 4.12) que

$$(\text{nat} \rightarrow \text{nat} \uplus \Delta) \uplus \Delta \doteq \text{nat} \rightarrow \text{nat} \uplus \Delta,$$

et en notant χ le type $\text{nat} \rightarrow \text{nat} \uplus \Delta$, on a plus simplement que

$$\text{rec} : \chi \rightarrow (\text{nat}^\Delta \rightarrow \chi \rightarrow \chi) \rightarrow \text{nat}^\Delta \rightarrow \chi$$

et on vérifie que d'une part

$$\lambda a. a : \chi$$

et d'autre part,

$$\lambda _ . \lambda r. \lambda acc. r (S \ acc) : \text{nat}^\Delta \rightarrow \chi \rightarrow \chi.$$

Donc finalement, si n a le type nat^Δ , on a

$$\text{rec} (\lambda a. a) (\lambda _ . \lambda r. \lambda acc. r (S \ acc)) n : \chi$$

ce qui permet de conclure que $eval$ a effectivement le type $\text{nat}^\Delta \rightarrow \text{nat} \uplus \Delta$.

Généraliser « $eval$ ». La fonction $eval$ force l'évaluation d'un entier. Elle peut être généralisée afin d'obtenir la fonction $cbind_{\text{nat}}$ qui applique une fonction sur les entiers à un entier potentiellement corrompu, mais en commençant par forcer l'évaluation de ce dernier, et n'appliquant la fonction que si c'est un entier licite. Sinon, l'exception corrompant est simplement propagée à la racine. Le codage (et le typage) de cette fonction s'obtient comme une simple variation de la fonction $eval$, à savoir :

$$\begin{aligned} cbind_{\text{nat}} & : \text{nat}^\Delta \rightarrow (\text{nat} \rightarrow B) \rightarrow B \uplus \Delta \\ & = \lambda n. \lambda f. (\text{rec} (\lambda a. f \ a) (\lambda e. \lambda r. \lambda a. r (S \ a)) \ n) \ 0 \end{aligned}$$

Cette fonction est plus générale que la fonction $eval$ et en particulier on a :

$$eval \equiv \lambda n. cbind_{\text{nat}} \ n \ (\lambda x. x)$$

Remarquons que par contravariance à gauche de la flèche, la fonction $cbind_{\text{nat}}$ a le type $\text{nat} \uplus \Delta \rightarrow (\text{nat} \rightarrow B) \rightarrow B \uplus \Delta$. Cependant, par définition $cbind_{\text{nat}}$ évalue toujours complètement son argument, et donc dans le cas où elle est appliquée à un argument de type $\text{nat} \uplus \Delta$, elle n'exploite pas le fait que l'exception ne peut se trouver qu'à la racine de l'argument.

Dans le cas où l'argument est de type $\text{nat} \uplus \Delta$, on utilisera alors plutôt la fonction plus efficace $bind_{\text{nat}}$:

$$\begin{aligned} bind_{\text{nat}} & : \text{nat} \uplus \Delta \rightarrow (\text{nat} \rightarrow B) \rightarrow B \uplus \Delta \\ & = \lambda n. \lambda f. \text{rec} (f \ 0) (\lambda x. \lambda _ . f (S \ x)) \ n \end{aligned}$$

Notons que le typage de $bind_{\text{nat}}$ est plus simple que celui de $cbind_{\text{nat}}$ et utilise le fait que rec a le type

$$B \rightarrow (\text{nat} \rightarrow B \rightarrow B) \rightarrow \text{nat} \uplus \Delta \rightarrow B \uplus \Delta.$$

Dans la suite (et principalement dans le chapitre 6), nous utiliserons les notations suivantes :

Notation

Si n et t sont deux termes et a une variable de terme,

- on notera $\text{let } a = n \text{ in } t$ le terme $bind_{\text{nat}} \ n \ (\lambda a. t)$.
- on notera $\text{clet } a = n \text{ in } t$ le terme $cbind_{\text{nat}} \ n \ (\lambda a. t)$.

Retour sur la soustraction. En utilisant les notations ci-dessus, on peut réécrire la fonction *minus* comme suit :

$$\begin{aligned} \text{minus} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \uplus \{\text{argis0}\} \\ & \equiv \lambda a. \lambda b. \\ & \quad \text{let } x = \text{rec } a \ (\lambda n. \lambda r. \text{pred } r) \ b \text{ in } x \end{aligned}$$

Ici la fonction *pred* est utilisée avec le type $\text{nat} \rightarrow \text{nat} \uplus \text{argis0}$ qui, nous l'avons vu, n'est pas son type le plus général.

La récursion terminale. Pour conclure sur la manipulation des entiers corrompus, on peut remarquer que les fonctions récursives terminales sur les entiers ont la propriété remarquable que si elles sont utilisées avec des entiers réellement corrompu (*i.e.* de type nat^Δ), elles renverront toujours un résultat de type $A \uplus \Delta$ (*i.e.* où l'exception est à la racine). Autrement dit, l'opérateur de récursion terminale sur les entiers, qui comme habituellement est définissable depuis *rec*, a le type suivant dans λ_ϵ :

$$\begin{aligned} \text{tail_rec} & : A \rightarrow (A \rightarrow A) \rightarrow \text{nat}^\Delta \rightarrow A \uplus \Delta \\ & \equiv \lambda z. \lambda f. \lambda n. \\ & \quad (\text{rec } (\lambda a. a) \ (\lambda _ . \lambda r. \lambda a. r \ (f \ a)) \ n) \ z \end{aligned}$$

Avec une telle fonction, la fonction *eval* peut se coder directement par :

$$\text{eval} \equiv \text{tail_rec } 0 \ S$$

La division euclidienne. Nous ne pouvions pas terminer cette section d'exemple sur les entiers et les exceptions sans parler de la fonction où ces exceptions apparaissent le plus naturellement, la division euclidienne. On va coder la division euclidienne avec l'algorithme naïf par soustraction successives. Cet algorithme requiert une induction avec un pas non unitaire. On commence donc par définir la fonction rec_n suivante :

$$\begin{aligned} \text{rec}_n & : \text{nat} \rightarrow A \rightarrow (\text{nat} \rightarrow A \rightarrow A) \rightarrow \text{nat} \rightarrow A \\ & \equiv \lambda n. \lambda z. \lambda f. \lambda m. \\ & \quad (\text{rec } (\lambda i. \text{rec } (f \ 0 \ z) \ (\lambda _ . \lambda _ . z) \ i) \\ & \quad (\lambda m'. \lambda r. \lambda n'. \text{rec } (f \ (S \ m') \ (r \ (\text{safe_pred } n))) \ (\lambda n''. \lambda _ . r \ n'') \ n') \ m \\ & \quad) \ n \end{aligned}$$

Cette fonction vérifie plus ou moins les réductions suivantes :

$$\begin{aligned} \text{rec}_n \ n \ z \ f \ 0 & \rightarrow z \\ \text{rec}_n \ n \ z \ f \ m & \rightarrow f \ (n - m) \ (\text{rec}_n \ n \ z \ f \ (n - m)) \quad (\text{Si } m \neq 0) \end{aligned}$$

À l'aide de cette fonction, on définit alors la division euclidienne *div* :

$$\begin{aligned} \text{div} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \uplus \{\text{divBy0}\} \\ & \equiv \lambda m. \lambda n. \text{rec } (\text{raise divBy0}) \ (\lambda _ . \lambda _ . \text{rec}_n \ n \ 0 \ S \ m) \ n \end{aligned}$$

Notons que le codage de la fonction rec_n est difficilement lisible. Ceci est principalement dû au manque de souplesse de l'opérateur de récursion. Dans la section 6.1, nous présenterons une variante de λ_ϵ muni d'un opérateur de point fixe et verrons à cette occasion que la division euclidienne peut y être codée plus simplement.

Pour conclure, précisons que dans le chapitre 6 nous ajouterons à λ_ϵ de nouveaux types de données primitifs, et notamment les listes. D'autres exemples d'utilisations des exceptions seront alors donnés.

4.4 Propriétés

Dans cette section nous allons présenter les propriétés vérifiées par λ_ϵ et son système de types. Les propriétés que nous allons prouver sont :

- La préservation du typage par réduction (théorème 4.45) ;
- La préservation du typage par η -réduction (théorème 4.46) ;
- La progression (théorème 4.47) qui montre que les termes clos bien typés qui ne sont pas des valeurs sont toujours réductibles : le calcul ne peut pas être bloqué ;
- Le théorème de corruption (théorème 4.50) qui montre formellement que les exceptions peuvent être utilisées en place de n'importe quel autre terme ;
- Le théorème de sûreté du typage vis-à-vis des exceptions (théorème 4.52) qui montre que les termes dont le type ne mentionne pas d'exceptions ne peuvent en lever durant leur exécution.

Afin de prouver ces propriétés, nous allons commencer par démontrer un certain nombre de lemmes sur le typage et le sous-typage de λ_ϵ .

4.4.1 Propriétés liminaires

Pour montrer la propriété de préservation du typage par réduction, on a besoin de montrer un certain nombre de propriétés du sous-typage et du typage de λ_ϵ . La relation de sous-typage de λ_ϵ est relativement complexe et les dérivations n'y sont pas dirigées par la syntaxe. Raisonner directement sur cette relation est alors difficile. Par exemple, on aimerait montrer que si deux types $A \rightarrow B$ et $A' \rightarrow B'$ sont en relation, alors les types A et A' et les types B et B' le sont aussi¹. Mais dans λ_ϵ ni le sous-type, ni le sur-type d'une flèche est une flèche. Pour contourner ce problème et plus généralement faciliter le raisonnement sur la relation de sous-typage de λ_ϵ , on va commencer par définir l'*équivalence de type*, puis on raisonnera sur le sous-typage modulo cette équivalence.

Équivalence de type

Supposons que l'on ait certains types A , B et C vérifiant la relation de sous-typage $C \leq A \rightarrow B$. Quelle peut alors être la forme du type C ? En observant les règles de sous-typage, on s'aperçoit que C n'est pas forcément directement une flèche, mais une flèche avec par dessus une suite d'unions et de corruptions entremêlées. Schématiquement, on peut avoir

$$C = (\dots ((A' \rightarrow B')^{\Delta_1} \boxtimes \Delta_1)^{\Delta_2} \boxtimes \Delta_2 \dots)^{\Delta_k} \boxtimes \Delta'_k.$$

Il n'est pas aisé de raisonner avec un tel type. On peut toutefois remarquer que l'on aura toujours $C \doteq (A'' \rightarrow B'') \boxtimes \Delta$ car l'égalité de sous-typage permet toujours de pousser les corruptions vers les feuilles du type et d'accumuler les unions. Il est plus simple de raisonner sur ce terme mais, parce que la relation de sous-typage est complexe, l'égalité de sous-typage l'est également. Par exemple, il est difficile de montrer que si $A \rightarrow B \doteq A' \rightarrow B'$ alors $A \doteq A'$ et $B \doteq B'$.

On définit donc l'*équivalence de type* qui est une relation d'équivalence entre types plus grossière que l'égalité de sous-typage car ne vérifiant que les égalités qui nous sont nécessaires.

¹On peut d'ailleurs tout de suite noter que la propriété souvent déduite dans ce genre de cas, à savoir que si $A \rightarrow B \leq A' \rightarrow B'$, alors $A' \leq A$ et $B \leq B'$, est fausse dans λ_ϵ . Il suffit de considérer le sous-typage $\text{nat} \rightarrow \text{nat} \leq \text{nat}^{\{\epsilon\}} \rightarrow \text{nat}^{\{\epsilon\}}$ pour s'en convaincre.

Définition 4.13 (Équivalence de types)

L'équivalence $\approx$ entre types de λ_ε est la plus petite relation d'équivalence (qui passe donc au contexte) vérifiant les équations suivantes :

$$\begin{array}{ll}
 (A \rightarrow B)^\Delta & \approx \quad A^\Delta \rightarrow B^\Delta \\
 (A \uplus \Delta) \uplus \Delta' & \approx \quad A \uplus (\Delta \cup \Delta') \\
 (A^\Delta)^{\Delta'} & \approx \quad A^{\Delta \cup \Delta'} \\
 (A \uplus \Delta)^{\Delta'} & \approx \quad A^{\Delta'} \uplus \Delta \\
 A^\emptyset & \approx \quad A \\
 A \uplus \emptyset & \approx \quad A
 \end{array}$$

L'équivalence de type est incluse dans l'égalité de sous-typage mais la réciproque est fausse en général. Par exemple, les égalités $(A \uplus \Delta')^\Delta \doteq A^\Delta \uplus (\Delta' - \Delta)$ (lemme 4.11) et $A \rightarrow B \uplus \Delta \doteq (A \rightarrow B \uplus \Delta) \uplus \Delta$ (lemme 4.12) sont dérivables, alors que les équivalences correspondantes ne sont pas vraies.

Lemme 4.14 Si A et B sont deux types tels que $A \approx B$, alors

$$A \doteq B$$

Preuve :

Chaque égalité de l'équivalence correspond soit directement à une égalité de sous-typage, soit (pour $A \uplus \emptyset \approx A$ et $A^\emptyset \approx A$) est évidente à montrer.

La suite de la preuve de préservation du typage par réduction s'articule de la façon suivante :

- On va montrer un certain nombre de propriétés de l'équivalence de type ;
- Ces propriétés nous permettront de montrer des propriétés du sous-typage modulo l'équivalence de type ;
- On en déduira alors des propriétés d'inversion du sous-typage de λ_ε ;
- On montrera par ailleurs les propriétés d'inversion du typage pour les différentes constructions du calcul ;
- Les propriétés d'inversion du typage et du sous-typage nous permettront alors de montrer la préservation du typage par réduction.

Plus précisément, la figure 4.3 donne les dépendances des principaux lemmes que nous allons montrer pour aboutir à la preuve de préservation du typage par réduction. Notons que nous avons simplifié l'énoncé de certains lemmes dans cette figure et que nous n'y avons pas mentionné certains lemmes auxiliaires (en particulier, nous n'y mentionnons pas les lemmes sur la notion de simplification qui est définie juste après et qui n'a pour but que de montrer les propriétés de l'équivalence de type).

L'équivalence de type est une relation d'équivalence, elle est donc symétrique. Ceci rend fastidieuse la preuve de certaines propriétés pourtant intuitivement simples (par exemple

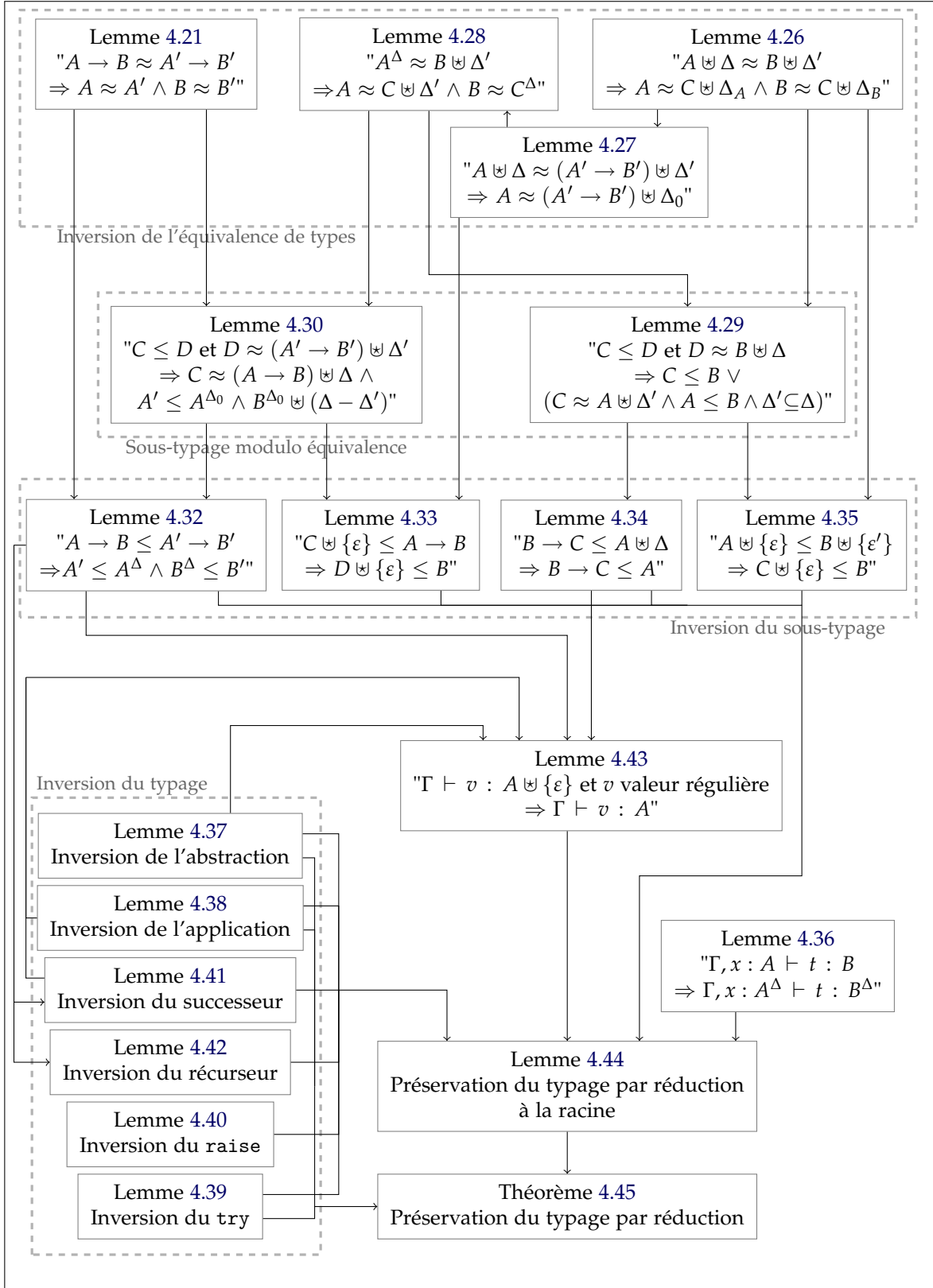


FIG. 4.3 – Schéma de dépendance pour la preuve de la préservation du typage par réduction

le fait que $A \rightarrow B \approx A' \rightarrow B'$ implique $A \approx A'$ et $B \approx B'$). On va donc utiliser l'astuce habituelle consistant à briser la symétrie en dirigeant les règles. Cette astuce appliquée à l'équivalence de type nous donne la *simplification de type*.

Définition 4.15 (Simplification de type)

On appelle simplification de type la relation de réduction $\rightsquigarrow$ définie comme la plus petite relation induite par les règles de la figure 4.4. On note $\rightsquigarrow^*$ la clôture réflexive et transitive de la simplification.

$$\begin{array}{ll}
 (A \rightarrow B)^\Delta \rightsquigarrow A^\Delta \rightarrow B^\Delta & (spl_1) \\
 (A \uplus \Delta) \uplus \Delta' \rightsquigarrow A \uplus (\Delta \cup \Delta') & (spl_2) \\
 (A^\Delta)^{\Delta'} \rightsquigarrow A^{\Delta \cup \Delta'} & (spl_3) \\
 (A \uplus \Delta)^{\Delta'} \rightsquigarrow A^{\Delta'} \uplus \Delta & (spl_4) \\
 A \uplus \emptyset \rightsquigarrow A & (spl_5) \\
 A^\emptyset \rightsquigarrow A & (spl_6) \\
 \\
 \frac{A \rightsquigarrow A'}{A \rightarrow B \rightsquigarrow A' \rightarrow B} (spl_{arr1}) & \frac{B \rightsquigarrow B'}{A \rightarrow B \rightsquigarrow A \rightarrow B'} (spl_{arr2}) \\
 \\
 \frac{A \rightsquigarrow A'}{A \uplus \Delta \rightsquigarrow A' \uplus \Delta} (spl_{uni}) & \frac{A \rightsquigarrow A'}{A^\Delta \rightsquigarrow A'^\Delta} (spl_{cor})
 \end{array}$$

FIG. 4.4 – La simplification de type

Pour pouvoir raisonner avec la simplification, on va d'abord montrer que c'est une relation confluente. Pour cela, on utilise la méthode de résolution des paires critiques. Rappelons en effet qu'une relation (du premier ordre) qui est fortement normalisable et dont toutes les paires critiques peuvent se résoudre est confluente (voir [BN98] pour une démonstration de ce résultat).

On montre donc d'une part que la simplification est fortement normalisable et d'autre part que toutes ses paires critiques peuvent se résoudre.

Lemme 4.16 (Normalisation forte de la simplification) *La simplification est fortement normalisable.*

Preuve :

On associe à chaque type A sa mesure $\lfloor A \rfloor$ qui est un entier naturel supérieur ou égal à 2 et qui est définie par :

$$\begin{array}{ll}
 \lfloor \text{nat} \rfloor & = 2 \\
 \lfloor A \rightarrow B \rfloor & = \lfloor A \rfloor + \lfloor B \rfloor \\
 \lfloor A \uplus \Delta \rfloor & = \lfloor A \rfloor + 1 \\
 \lfloor A^\Delta \rfloor & = \lfloor A \rfloor^2
 \end{array}$$

Remarquons tout d'abord que quel que soit le type A , on a bien $\lfloor A \rfloor \geq 2$. On montre alors que si $A \rightsquigarrow B$, alors $\lfloor A \rfloor > \lfloor B \rfloor$ et ce par cas sur la règle utilisée pour montrer $A \rightsquigarrow B$:

- (spl₁) $\lfloor (A \rightarrow B)^\Delta \rfloor = (\lfloor A \rfloor + \lfloor B \rfloor)^2 > \lfloor A \rfloor^2 + \lfloor B \rfloor^2 = \lfloor A^\Delta \rightarrow B^\Delta \rfloor$.
- (spl₂) $\lfloor A \uplus \Delta \uplus \Delta' \rfloor = \lfloor A \rfloor + 2 > \lfloor A \rfloor + 1 = \lfloor A \uplus (\Delta \cup \Delta') \rfloor$.
- (spl₃) $\lfloor (A^\Delta)^{\Delta'} \rfloor = \lfloor A \rfloor^4 > \lfloor A \rfloor^2 = \lfloor A^{\Delta \cup \Delta'} \rfloor$.
- (spl₄) $\lfloor A \uplus \Delta^{\Delta'} \rfloor = (\lfloor A \rfloor + 1)^2 > \lfloor A \rfloor^2 + 1 = \lfloor A^{\Delta'} \uplus \Delta \rfloor$.
- (spl₅) $\lfloor A \uplus \emptyset \rfloor = \lfloor A \rfloor + 1 > \lfloor A \rfloor$.
- (spl₆) $\lfloor A^\emptyset \rfloor = \lfloor A \rfloor^2 > \lfloor A \rfloor$.
- (spl_{arr1}) Si $\lfloor A \rfloor > \lfloor A' \rfloor$, alors on a bien $\lfloor A \rightarrow B \rfloor = \lfloor A \rfloor + \lfloor B \rfloor > \lfloor A' \rfloor + \lfloor B \rfloor = \lfloor A' \rightarrow B \rfloor$.
- (spl_{arr2}) Si $\lfloor B \rfloor > \lfloor B' \rfloor$, alors on a bien $\lfloor A \rightarrow B \rfloor = \lfloor A \rfloor + \lfloor B \rfloor > \lfloor A \rfloor + \lfloor B' \rfloor = \lfloor A \rightarrow B' \rfloor$.
- (spl_{uni}) Si $\lfloor A \rfloor > \lfloor A' \rfloor$, alors $\lfloor A \uplus \Delta \rfloor = \lfloor A \rfloor + 1 > \lfloor A' \rfloor + 1 = \lfloor A' \uplus \Delta \rfloor$.
- (spl_{cor}) Si $\lfloor A \rfloor > \lfloor A' \rfloor \geq 2$, alors $\lfloor A^\Delta \rfloor = \lfloor A \rfloor^2 > \lfloor A' \rfloor^2 = \lfloor A'^\Delta \rfloor$.

Lemme 4.17 (Résolution des paires critiques) *Toutes les paires critiques de la simplification se résolvent.*

Preuve :

On montre figure 4.5 et 4.6 comment résoudre toutes les paires critiques de la simplification.

Lemme 4.18 (Confluence de la simplification) *Si A, B et C sont des types tels que $A \rightsquigarrow^* B$ et $A \rightsquigarrow^* C$, alors il existe un type D tel que*

$$B \rightsquigarrow^* D \quad \text{et} \quad C \rightsquigarrow^* D$$

Preuve :

La confluence est une conséquence des lemmes 4.16 et 4.17 (voir [BN98]).

On montre alors que deux types sont équivalents si et seulement si ils ont un réduit commun par la simplification :

Lemme 4.19 *Si A et B sont des types, alors $A \approx B$ si et seulement si il existe un type C tel que $A \rightsquigarrow^* C$ et $B \rightsquigarrow^* C$.*

Preuve :

Les deux sens de l'équivalence sont faciles à montrer puisque à chaque règle de $\approx$ correspond une règle de $\rightsquigarrow^*$ et inversement.

En particulier, si $A \rightsquigarrow^* B$ alors $A \approx B$, propriété que l'on utilisera implicitement par la suite.

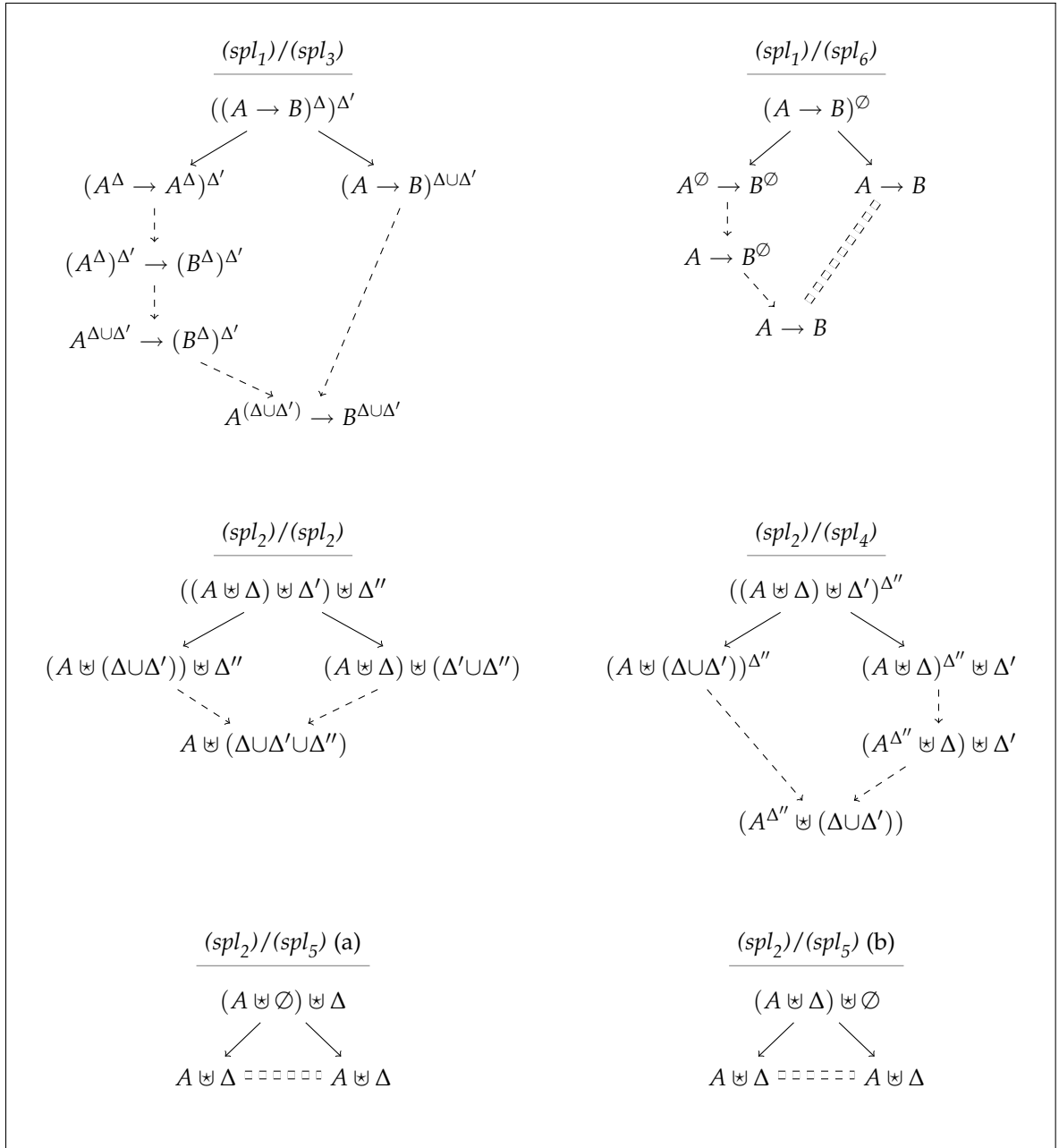


FIG. 4.5 – Paires critiques de la simplification de type

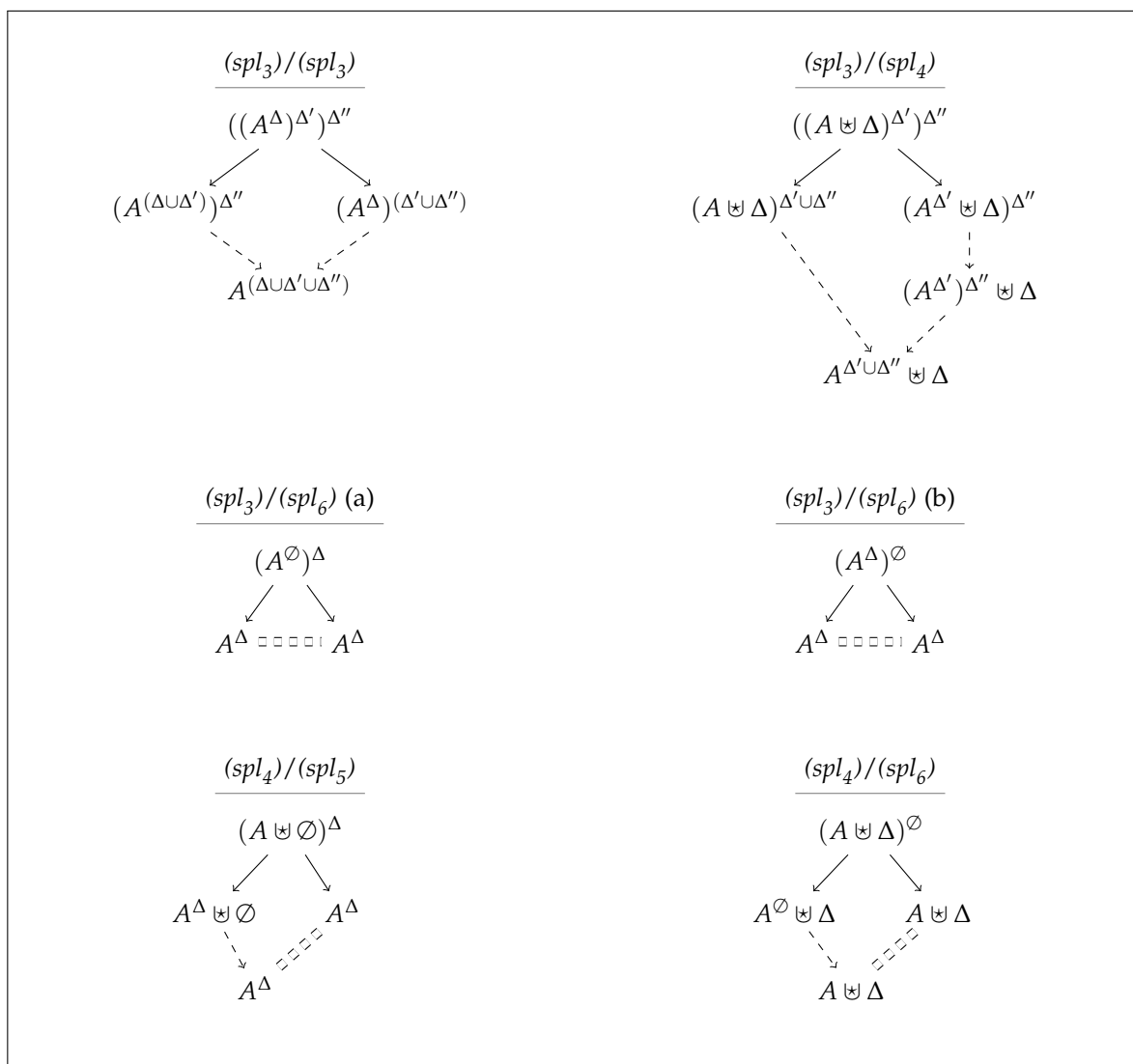


FIG. 4.6 – Paires critiques de la simplification de type (suite)

On montre également que la seule façon de simplifier une flèche est d'en simplifier les composantes, les simplifiés d'une flèche sont des flèches :

Lemme 4.20 Si A, B et C sont des types tels que $A \rightarrow B \rightsquigarrow^* C$, alors il existe A' et B' tel que

$$C = A' \rightarrow B' \quad \text{et} \quad A \rightsquigarrow^* A' \quad \text{et} \quad B \rightsquigarrow^* B'$$

Preuve :

Ce lemme est facilement prouvé par induction sur la longueur de la réduction $A \rightarrow B \rightsquigarrow^* C$. Toutes les règles de la simplification conservent en effet les flèches.

Une conséquence immédiate en est la propriété sur l'équivalence de types flèches suivantes :

Lemme 4.21 Si A, B, A' et B' sont des types tels que $A \rightarrow B \approx A' \rightarrow B'$, alors

$$A \approx A' \quad \text{et} \quad B \approx B'$$

Preuve :

En utilisant le lemme 4.19, on sait qu'il existe une type C tel que $A \rightarrow B \rightsquigarrow^* C$ et $A' \rightarrow B' \rightsquigarrow^* C$. L'application du lemme 4.20 sur les deux réductions nous permet alors de conclure.

En simplifiant suffisamment un type quelconque, on obtiendra forcément l'une des formes suivantes : nat , $\text{nat} \uplus \Delta$, $\text{nat}^{\Delta'}$, $\text{nat}^{\Delta'} \uplus \Delta$, $A \rightarrow B$ ou $(A \rightarrow B) \uplus \Delta$. Pour éviter d'avoir à écrire tous ces cas à chaque fois, on définit les méta-notations suivantes :

Notation

- Si $\Delta \neq \emptyset$, la notation $A \overset{\circ}{\uplus} \Delta$ désigne $A \uplus \Delta$.
Sinon, si $\Delta = \emptyset$, $A \overset{\circ}{\uplus} \Delta$ désigne ou bien $A \uplus \emptyset$, ou bien A .
- Si $\Delta \neq \emptyset$, la notation $A^{\circ\Delta}$ désigne A^Δ .
Sinon, si $\Delta = \emptyset$, $A^{\circ\Delta}$ désigne ou bien $A^\emptyset$, ou bien A .

On montre alors que tout type peut se simplifier en un type d'une des formes citées ci-dessus :

Lemme 4.22 Si A est un type, alors on a un des cas suivants :

- Il existe des ensembles d'exceptions Δ et Δ' tels que $A \rightsquigarrow^* \text{nat}^{\circ\Delta} \overset{\circ}{\uplus} \Delta'$.
- Il existe des types B et C et un ensemble d'exceptions Δ tels que $A \rightsquigarrow^* (B \rightarrow C) \overset{\circ}{\uplus} \Delta$.

Preuve :

On raisonne par induction sur la structure du type A . Si A est :

- nat , alors on conclut directement (on est dans le premier cas et $\Delta = \Delta' = \emptyset$).
- $B \rightarrow C$, alors on conclut directement (on est dans le second cas et $\Delta = \emptyset$).
- $X \uplus \Delta_x$, alors par hypothèse d'induction,
 - Soit $X \rightsquigarrow^* \text{nat}^{\circ\Delta} \uplus \Delta'$, mais alors $X \uplus \Delta_x \rightsquigarrow^* \text{nat}^{\circ\Delta} \uplus (\Delta' \cup \Delta_x)$.
 - Soit $X \rightsquigarrow^* (B \rightarrow C) \uplus \Delta$, mais alors $X \uplus \Delta_x \rightsquigarrow^* (B \rightarrow C) \uplus (\Delta \cup \Delta_x)$.
- X^{Δ_x} , alors par hypothèse d'induction,
 - Soit $X \rightsquigarrow^* \text{nat}^{\circ\Delta} \uplus \Delta'$, mais alors $X^{\Delta_x} \rightsquigarrow^* \text{nat}^{\circ(\Delta \cup \Delta_x)} \uplus \Delta'$.
 - Soit $X \rightsquigarrow^* (B \rightarrow C) \uplus \Delta$, mais alors $X^{\Delta_x} \rightsquigarrow^* (B^{\Delta_x} \rightarrow C^{\Delta_x}) \uplus \Delta$.

Notons que ce lemme implique trivialement que quel que soit A , ou bien $A \approx \text{nat}^{\Delta'} \uplus \Delta$, ou bien $A \approx (B \rightarrow C) \uplus \Delta$ (où on n'a plus besoin des notations introduites ci-dessus puisque on est passé à l'équivalence).

On montre la propriété suivante sur les réduits d'une union :

Lemme 4.23 Si A et C sont des types et Δ un ensemble de noms d'exceptions tels que $A \uplus \Delta \rightsquigarrow^* C$, alors il existe un type A' et un ensemble de noms d'exceptions Δ_A tel que

$$C = A' \uplus (\Delta_A \cup \Delta) \quad \text{et} \quad A \rightsquigarrow^* A' \uplus \Delta_A$$

Preuve :

On procède par induction sur la longueur de la réduction $A \uplus \Delta \rightsquigarrow^* C$. Si $C = A \uplus \Delta$, alors on conclut directement. Sinon, on a $A \uplus \Delta \rightsquigarrow D \rightsquigarrow^* C$. On raisonne alors pas cas sur la première réduction. Si cette première réduction est :

(*spl*₂) Alors $A = X \uplus \Delta_x$ et $D = X \uplus (\Delta_x \cup \Delta)$. Alors par hypothèse d'induction, on a $C = A' \uplus (\Delta_A \cup \Delta_x \cup \Delta)$ et $X \rightsquigarrow^* X \uplus \Delta_A$ et on a bien $X \uplus \Delta_x \rightsquigarrow^* X \uplus (\Delta_A \cup \Delta_x)$.

(*spl*₅) Alors $\Delta = \emptyset$ et $D = A$, mais alors $C = C$ et $A \rightsquigarrow^* C$.

(*spl*_{uni}) Alors on a $D = A' \uplus \Delta$ avec $A \rightsquigarrow A'$. Par hypothèse d'induction, $C = A'' \uplus (\Delta_A \cup \Delta)$ et $A' \rightsquigarrow^* A'' \uplus \Delta_A$. Mais on a bien alors $A \rightsquigarrow^* A'' \uplus \Delta_A$.

Aucun autre cas de réduction n'est possible puisque $A \uplus \Delta$ est une union.

L'équivalence de type n'ajoute, ni n'enlève d'exceptions. En conséquence, un type qui n'est pas une union n'est équivalent à une union que si l'ensemble d'exceptions de cette union est vide :

Lemme 4.24 Si A , B et C sont des types et Δ un ensemble de noms d'exceptions tels que $A \rightarrow B \approx C \uplus \Delta$, alors $\Delta = \emptyset$.

Preuve :

Avec le lemme 4.19, on sait qu'il existe un type D tel que $A \rightarrow B \rightsquigarrow^* D$ et $C \uplus \Delta \rightsquigarrow^* D$. Alors avec les lemmes 4.20 et 4.23, on sait que $D = A' \rightarrow B'$ et $D = C' \overset{\circ}{\uplus} (\Delta_C \cup \Delta)$, d'où on en conclut qu'on a forcément $\Delta_C \cup \Delta = \emptyset$ et donc $\Delta = \emptyset$.

Lemme 4.25 *Si C est un type et Δ et Δ' sont des ensembles de noms d'exceptions tels que $\text{nat}^{\Delta'} \approx C \uplus \Delta$, alors $\Delta = \emptyset$.*

Preuve :

Avec le lemme 4.19, on sait qu'il existe un type D tel que $\text{nat}^{\Delta'} \rightsquigarrow^* D$ et $C \uplus \Delta \rightsquigarrow^* D$. Alors avec le lemme 4.23, on sait que $D = C' \overset{\circ}{\uplus} (\Delta_C \cup \Delta)$ et, au vu de la définition de $\rightsquigarrow^*$, $D = \text{nat}^{\Delta'}$. On en conclut donc qu'on a forcément $\Delta_C \cup \Delta = \emptyset$ et donc $\Delta = \emptyset$.

On peut alors montrer la propriété suivante entre deux types unions équivalents :

Lemme 4.26 *Si A et B sont des types et Δ et Δ' des ensembles de noms d'exceptions tels que $A \uplus \Delta \approx B \uplus \Delta'$, alors il existe un type C et des ensembles de noms d'exceptions Δ_A et Δ_B tel que*

$$A \approx C \uplus \Delta_A \quad \text{et} \quad B \approx C \uplus \Delta_B \quad \text{et} \quad \Delta_A \cup \Delta = \Delta_B \cup \Delta'$$

Preuve :

En utilisant le lemme 4.22, on se réduit aux quatre cas suivants :

- $A \approx \text{nat}^{\Delta_X} \uplus \Delta_A$ et $B \approx \text{nat}^{\Delta'_X} \uplus \Delta_B$: On a alors $A \uplus \Delta \approx \text{nat}^{\Delta_X} \uplus (\Delta_A \cup \Delta)$ et $B \uplus \Delta' \approx \text{nat}^{\Delta'_X} \uplus (\Delta_B \cup \Delta')$. Avec le lemme 4.19, il existe un type C tel que $\text{nat}^{\Delta_X} \uplus \Delta_A \rightsquigarrow^* C$ et $\text{nat}^{\Delta'_X} \uplus \Delta_B \rightsquigarrow^* C$. Mais comme ni $\text{nat}^{\Delta_X} \uplus \Delta_A$, ni $\text{nat}^{\Delta'_X} \uplus \Delta_B$ ne sont réductibles par $\rightsquigarrow$, on a forcément $\text{nat}^{\Delta_X} \uplus \Delta_A = \text{nat}^{\Delta'_X} \uplus \Delta_B$ et donc $\Delta_X = \Delta'_X$ et $\Delta_A \cup \Delta = \Delta_B \cup \Delta'$.
- $A \approx (X \rightarrow Y) \uplus \Delta_A$ et $B \approx (X' \rightarrow Y') \uplus \Delta_B$: On a alors $A \uplus \Delta \approx (X \rightarrow Y) \uplus (\Delta_A \cup \Delta)$ et $B \uplus \Delta' \approx (X' \rightarrow Y') \uplus (\Delta_B \cup \Delta')$. Avec le lemme 4.19, il existe un type C tel que $(X \rightarrow Y) \uplus \Delta_A \rightsquigarrow^* C$ et $B \approx (X' \rightarrow Y') \uplus \Delta_B \rightsquigarrow^* C$. En utilisant conjointement les lemmes 4.23, 4.20 et 4.24, on obtient que l'on a d'une part $C = (X_0 \rightarrow Y_0) \uplus (\Delta_A \cup \Delta)$ avec $X \rightsquigarrow^* X_0$ et $Y \rightsquigarrow^* Y_0$ et d'autre part $C = (X'_0 \rightarrow Y'_0) \uplus (\Delta_B \cup \Delta')$ avec $X' \rightsquigarrow^* X'_0$ et $Y' \rightsquigarrow^* Y'_0$. Finalement, on a forcément $X_0 = X'_0$, $Y_0 = Y'_0$ et $\Delta_A \cup \Delta = \Delta_B \cup \Delta'$ et on peut conclure.
- $A \approx \text{nat}^{\Delta_X} \uplus \Delta_A$ et $B \approx (X' \rightarrow Y') \uplus \Delta_B$: On a alors $A \uplus \Delta \approx \text{nat}^{\Delta_X} \uplus (\Delta_A \cup \Delta)$ et $B \uplus \Delta' \approx (X' \rightarrow Y') \uplus \Delta_B \cup \Delta'$. Avec le lemme 4.19, il existe un type C tel que $\text{nat}^{\Delta_X} \uplus \Delta_A \rightsquigarrow^* C$ et $(X' \rightarrow Y') \uplus \Delta_B \rightsquigarrow^* C$. Puisque $\text{nat}^{\Delta_X} \uplus \Delta_A$ n'est pas réductible par $\rightsquigarrow$, on a forcément $(X' \rightarrow Y') \uplus \Delta_B \rightsquigarrow^* \text{nat}^{\Delta_X} \uplus \Delta_A$ et avec les lemmes 4.23, 4.20 et 4.24, on vérifie facilement que ce cas ne peut pas se produire.
- $A \approx (X \rightarrow Y) \uplus \Delta_A$ et $B \approx \text{nat}^{\Delta'_X} \uplus \Delta_B$: On est dans le cas symétrique du cas précédent et il ne peut pas non plus se produire.

Une conséquence simple du lemme précédent est :

Lemme 4.27 Si A , A' et B' sont des types et Δ et Δ' des ensembles de noms d'exceptions tels que $A \uplus \Delta \approx (A' \rightarrow B') \uplus \Delta'$, alors

$$A \approx (A' \rightarrow B') \uplus \Delta_0 \quad \text{et} \quad \Delta' = \Delta \cup \Delta_0$$

Preuve :

Puisque $A \uplus \Delta \approx (A' \rightarrow B') \uplus \Delta'$, avec le lemme 4.26, on sait qu'il existe C tel que $A \approx C \uplus \Delta_0$, $A' \rightarrow B' \approx C \uplus \Delta_B$ et $\Delta_0 \cup \Delta = \Delta_B \cup \Delta'$. Alors, avec le lemme 4.24 on a $\Delta_B = \emptyset$ et donc $C \approx A' \rightarrow B'$ et alors $A \approx (A' \rightarrow B') \uplus \Delta_0$ et $\Delta_0 \cup \Delta = \Delta'$.

Enfin, la dernière propriété de l'équivalence que l'on montre concerne l'équivalence entre un type corrompu et un type union :

Lemme 4.28 Si A et B sont des types et Δ et Δ' des ensembles de noms d'exceptions tels que $A^\Delta \approx B \uplus \Delta'$, alors il existe un type C tel

$$A \approx C \uplus \Delta' \quad \text{et} \quad B \approx C^\Delta$$

Preuve :

En utilisant le lemme 4.22, on se réduit aux cas suivants :

- Soit $A \approx \text{nat}^{\Delta_x} \uplus \Delta_A$: Alors $A^\Delta \approx \text{nat}^{(\Delta_x \cup \Delta)} \uplus \Delta_A \approx B \uplus \Delta'$ et avec le lemme 4.19, le fait que la simplification soit confluente (lemme 4.18) et le fait que $\text{nat}^{o(\Delta_x \cup \Delta)} \uplus \Delta_A$ n'est pas réductible par $\rightsquigarrow$, on a $B \uplus \Delta' \rightsquigarrow^* \text{nat}^{o(\Delta_x \cup \Delta)} \uplus \Delta_A$. Alors avec le lemme 4.23, on a $B \rightsquigarrow^* \text{nat}^{o(\Delta_x \cup \Delta)} \uplus \Delta_0$ et $\Delta_A = \Delta_0 \cup \Delta'$. En prenant $C = \text{nat}^{\Delta_x} \uplus \Delta_0$, on a bien que $A \approx C \uplus \Delta'$ et $B \approx C^\Delta$.
- Soit $A \approx (X \rightarrow Y) \uplus \Delta_A$: Alors $A^\Delta \approx (X^\Delta \rightarrow Y^\Delta) \uplus \Delta_A \approx B \uplus \Delta'$ et avec le lemme 4.27, $B \approx (X^\Delta \rightarrow Y^\Delta) \uplus \Delta_0$ avec $\Delta_A = \Delta' \cup \Delta_0$. Donc si on prend $C = (X \rightarrow Y) \uplus \Delta_0$, on a bien $A \approx C \uplus \Delta'$ et $B \approx C^\Delta$.

Propriétés du sous-typage modulo équivalence. On vient de montrer diverses propriétés de l'équivalence de type. On va maintenant les utiliser pour montrer des propriétés du sous-typage modulo équivalence de type. Ces propriétés nous permettront de démontrer les propriétés du sous-typage nécessaires à la preuve de préservation du typage par réduction.

Lemme 4.29 Si C , D et B sont des types et Δ un ensemble de noms d'exceptions tels que $C \leq D$ et $D \approx B \uplus \Delta$, alors on a forcément l'un des cas suivants :

- Soit $C \leq B$.
- Soit il existe A tel que $C \approx A \uplus \Delta'$ et $A \leq B$ et $\Delta' \subseteq \Delta$

Preuve :

On raisonne par induction sur la dérivation de $C \leq D$. Si la dernière règle utilisée est :

(*st-id*) On a $C \leq C$ et $C \approx B \uplus \Delta$. On est donc directement dans le second cas.

(*st-trans*) On a $C \leq D$ car $C \leq X$ et $X \leq D$ et on a $D \approx B \uplus \Delta$. Par hypothèse d'induction,

– Soit $X \leq B$ et on conclut car alors $C \leq B$.

– Soit $X \approx A_0 \uplus \Delta_0$ avec $A_0 \leq B$ et $\Delta_0 \subseteq \Delta$. On peut appliquer l'hypothèse d'induction avec $C \leq X$ et :

– Soit $C \leq A_0$, mais alors $C \leq B$.

– Soit $C \approx A \uplus \Delta'$ avec $A \leq A_0$ et $\Delta' \subseteq \Delta_0$. Mais alors $A \leq B$ et $\Delta' \subseteq \Delta$.

(*st-arrow*) On est dans le cas où $C_1 \rightarrow C_2 \leq D_1 \rightarrow D_2$ et $D_1 \rightarrow D_2 \approx B \uplus \Delta$. Mais alors avec le lemme 4.24 on a que $\Delta = \emptyset$ et donc avec le lemme 4.14, on a $D_1 \rightarrow D_2 \doteq B$ et donc $C_1 \rightarrow C_2 \leq B$.

(*st-cor-noex*) On a $C^\emptyset \leq C$ et $C \approx B \uplus \Delta$. Mais alors $C^\emptyset \approx B \uplus \Delta$ et on conclut directement.

(*st-uni-merge*) On est dans le cas d'une égalité de l'équivalence, on est donc directement dans le second cas.

(*st-cor-merge*) On est dans le cas d'une égalité de l'équivalence, on est donc directement dans le second cas.

(*st-uni-arrow*) On est dans le cas où $(C_1 \rightarrow C_2) \uplus \Delta' \leq C_1 \rightarrow C_2 \uplus \Delta'$ et $C_1 \rightarrow C_2 \uplus \Delta' \approx B \uplus \Delta$. Avec le lemme 4.24, on a $\Delta = \emptyset$ et donc avec le lemme 4.14, on a $C_1 \rightarrow C_2 \uplus \Delta' \doteq B$ et donc $(C_1 \rightarrow C_2) \uplus \Delta' \leq B$.

(*st-uni-ctx*) On est dans le cas où $C \uplus \Delta' \leq D \uplus \Delta'$ car $C \leq D$ et on a $D \uplus \Delta' \approx B \uplus \Delta$. Avec le lemme 4.26, on sait qu'il existe X, Δ_D et Δ_B tel que $D \approx X \uplus \Delta_D$, $B \approx X \uplus \Delta_B$ et $\Delta_D \cup \Delta' = \Delta_B \cup \Delta$. On applique l'hypothèse de récurrence et on a que

– Soit $C \leq X \leq X \uplus \Delta_B \doteq B$ (avec le lemme 4.14).

– Soit $C \approx Y \uplus \Delta''$ avec $Y \leq X$ et $\Delta'' \subseteq \Delta_D$. Soit $\Delta_X = (\Delta'' \cup \Delta') \cap \Delta_B$, $C \uplus \Delta' \approx Y \uplus (\Delta'' \cup \Delta') \approx (Y \uplus \Delta_X) \uplus ((\Delta'' \cup \Delta') - \Delta_X)$ et on peut conclure car d'une part, puisque $\Delta_X \subseteq \Delta_B$, $Y \uplus \Delta_X \leq X \uplus \Delta_B \doteq B$ (lemme 4.14). Et d'autre part, puisque $\Delta'' \cup \Delta' \subseteq \Delta_D \cup \Delta' = \Delta_B \cup \Delta$, $(\Delta'' \cup \Delta') - \Delta_X \subseteq \Delta$.

(*st-uni*) On est dans le cas où $C \leq C \uplus \Delta'$ et $C \uplus \Delta' \approx B \uplus \Delta$. Avec le lemme 4.26, il existe X, Δ_C et Δ_B tel que $C \approx X \uplus \Delta_C$, $B \approx X \uplus \Delta_B$ et $\Delta_C \cup \Delta' = \Delta_B \cup \Delta$. Mais donc $A \approx (X \uplus (\Delta_A \cap \Delta_B)) \uplus (\Delta_A - (\Delta_A \cap \Delta_B))$ et on conclut puisque $X \uplus (\Delta_A \cap \Delta_B) \leq X \uplus \Delta_B \doteq B$ (lemme 4.14) et comme $\Delta_C \subseteq \Delta_B \cup \Delta$, $\Delta_A - (\Delta_A \cap \Delta_B) \subseteq \Delta$.

(*st-cor*) On est dans le cas où $C \uplus \Delta' \leq C^{\Delta'}$ et $C^{\Delta'} \approx B \uplus \Delta$. Avec le lemme 4.28, il existe D tel que $C \approx D \uplus \Delta$ et $B \approx D^{\Delta'}$. Mais alors $C \uplus \Delta' \approx (D \uplus \Delta') \uplus \Delta$ et on conclut puisque $D \uplus \Delta' \leq D^{\Delta'} \doteq B$ (avec le lemme 4.14).

(*st-uni-cor*) On est dans le cas d'une égalité de l'équivalence, on est donc directement dans le second cas.

(*st-cor-arrow*) On est dans le cas d'une égalité de l'équivalence, on est donc directement dans le second cas.

Lemme 4.30 Si C, D, A' et B' sont des types et Δ' un ensemble de noms d'exceptions tels que $C \leq D$ et $D \approx (A' \rightarrow B') \uplus \Delta'$, alors il existe des types A et B et des ensembles d'exceptions Δ et Δ_0 tels que

- $C \approx (A \rightarrow B) \uplus \Delta$
- $A' \leq A^{\Delta_0}$
- $B^{\Delta_0} \uplus (\Delta - \Delta') \leq B'$

Preuve :

On raisonne par induction sur la dérivation de $C \leq D$. Si la dernière règle utilisée est :

(*st-id*) Alors c'est évident.

(*st-trans*) On est dans le cas où $C \leq D$ car il existe X tel que $C \leq X$ et $X \leq D$. Comme de plus

- $D \approx (A' \rightarrow B') \uplus \Delta'$, on applique l'hypothèse d'induction pour obtenir qu'il existe A_x, B_x, Δ_x et Δ'_x tel que
- $X \approx (A' \rightarrow B') \uplus \Delta_x$
 - $A' \leq A_x^{\Delta'_x}$
 - $B_x^{\Delta'_x} \uplus (\Delta' - \Delta_x) \leq B'$

Mais on peut donc également appliquer l'hypothèse d'induction à $C \leq X$, et on obtient qu'il existe A, B, Δ et Δ_0 tel que

- $C \approx (A \rightarrow B) \uplus \Delta$
- $A_x \leq A^{\Delta_0}$
- $B^{\Delta_0} \uplus (\Delta_x - \Delta) \leq B_x$

Mais on conclut alors puisque

- $A' \leq A_x^{\Delta'_x} \leq A^{\Delta_0 \cup \Delta'_x}$
- $B^{\Delta_0 \cup \Delta'_x} \uplus \Delta' - \Delta \doteq (B^{\Delta_0} \uplus (\Delta_x - \Delta))^{\Delta'_x} \uplus (\Delta' - \Delta_x) \leq B_x^{\Delta'_x} \uplus \Delta' - \Delta_x \leq B'$

(*st-arrow*) On est dans le cas où $C_1 \rightarrow C_2 \leq D_1 \rightarrow D_2$ car $D_1 \leq C_1$ et $C_2 \leq D_2$. Alors, avec le lemme 4.24, on a $\Delta' = \emptyset$ et donc avec le lemme 4.21, on a $D_1 \approx A'$ et $D_2 \approx B'$. De plus $C_1 \rightarrow C_2 \approx C_1 \rightarrow C_2$ et on a $A' \leq C_1$ et $C_2 \leq B'$ puisque avec le lemme 4.14, $A' \doteq D_1$ et $B' \doteq D_2$.

(*st-cor-noex*) On est dans le cas où $C^\emptyset \leq C$ et $C \approx (A' \rightarrow B') \uplus \Delta'$. Mais alors $C^\emptyset \approx ((A' \rightarrow B') \uplus \Delta')^\emptyset \approx (A' \rightarrow B') \uplus \Delta'$ et on conclut.

(*st-uni-merge*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

(*st-cor-merge*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

(*st-uni-arrow*) On est dans le cas où $(C_1 \rightarrow C_2) \uplus \Delta_c \leq C_1 \rightarrow C_2 \uplus \Delta_c$ et $C_1 \rightarrow C_2 \uplus \Delta_c \approx A' \rightarrow B' \uplus \Delta'$. Grâce aux lemmes 4.24 et 4.21, $\Delta' = \emptyset$, $C_1 \approx A'$ et $C_2 \uplus \Delta_c \approx B'$. Et comme $(C_1 \rightarrow C_2) \uplus \Delta_c \approx (C_1 \rightarrow C_2) \uplus \Delta_c$, on conclut en montrant avec le lemme 4.14 que $A' \leq C_1$ et $C_2 \uplus \Delta_c \leq B'$ (et $\Delta_c - \Delta' = \Delta_c$ puisque $\Delta' = \emptyset$).

(*st-uni-ctx*) On est dans le cas où $C \uplus \Delta_c \leq D \uplus \Delta_c$ car $C \leq D$ et avec $D \uplus \Delta_c \approx (A' \rightarrow B') \uplus \Delta'$. Avec le lemme 4.27, on a $D \approx (A' \rightarrow B') \uplus \Delta'_0$ avec $\Delta' = \Delta'_0 \cup \Delta_c$. Alors par hypothèse d'induction $C \approx (A \rightarrow B) \uplus \Delta_0$ avec $A' \leq A^{\Delta_1}$ et $B^{\Delta_1} \uplus (\Delta_0 - \Delta'_0) \leq B'$. Et finalement on conclut puisque $C \uplus \Delta_c \approx (A \rightarrow B) \uplus (\Delta_0 \cup \Delta_c)$ et $(\Delta_0 \cup \Delta_c) - \Delta' = \Delta_0 - \Delta'_0$.

(*st-uni*) On est dans le cas où $C \leq C \uplus \Delta_c$ et $C \uplus \Delta_c \approx (A' \rightarrow B') \uplus \Delta'$. Or avec le lemme 4.27, $C \approx (A' \rightarrow B') \uplus \Delta_0$ avec $\Delta' = \Delta_0 \cup \Delta_c$ et on conclut donc puisque $\Delta_0 - \Delta' = \emptyset$.

(*st-cor*) On est dans le cas où $C \uplus \Delta_c \leq C^{\Delta_c}$ et $C^{\Delta_c} \approx (A' \rightarrow B') \uplus \Delta'$. Alors, avec le lemme 4.28, on obtient que $C \approx (A \rightarrow B) \uplus \Delta'$ avec $A^{\Delta_c} \approx A'$ et $B^{\Delta_c} \approx B'$. On conclut donc avec le lemme 4.14.

(*st-uni-cor*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

(*st-cor-arrow*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

Dans l'énoncé du lemme précédent, les ensembles de noms d'exceptions Δ , Δ' et Δ_0 peuvent avoir des intersections non vides mais ne sont reliés par aucune relation particulière. Intuitivement, Δ rend compte d'une partie des exceptions de Δ' mais surtout d'exceptions qui peuvent être « rentrées » à droite de la flèche grâce à la règle de sous-typage (*st-uni-arrow*). L'ensemble Δ_0 lui, rend compte des exceptions qui ont pu se distri-

buer sur la flèche à l'aide de la règle (*st-cor-arrow*). On a par exemple

$$(A \rightarrow B)^{\{\varepsilon_0\}} \uplus \{\varepsilon_1\} \leq (A^{\{\varepsilon_0\}} \rightarrow (B^{\{\varepsilon_0\}} \uplus \{\varepsilon_1\})) \uplus \{\varepsilon_2\}$$

pour lequel le lemme précédent est vrai en prenant $\Delta_0 = \{\varepsilon_0\}$, $\Delta = \{\varepsilon_1\}$ et $\Delta' = \{\varepsilon_2\}$.

Le lemme qui suit caractérise les sous-types des entiers corrompus :

Lemme 4.31 *Si C et D sont des types et Δ et Δ' des ensembles de noms d'exceptions tels que $C \leq D$ et $D \approx \text{nat}^\Delta \uplus \Delta'$, alors il existe des ensembles de noms d'exceptions Δ_0 et Δ'_0 tels que*

- $C \approx \text{nat}^{\Delta_0} \uplus \Delta'_0$
- $\Delta_0 \cup \Delta'_0 \subseteq \Delta \cup \Delta'$

Preuve :

On raisonne par induction sur la dérivation de $C \leq D$. Si la dernière règle utilisée est :

(*st-id*) Alors c'est évident.

(*st-trans*) On est dans le cas où $C \leq D$ car il existe X tel que $C \leq X$ et $X \leq D$. Comme de plus $D \approx \text{nat}^\Delta \uplus \Delta'$, on applique l'hypothèse d'induction pour obtenir que $X \approx \text{nat}^{\Delta_x} \uplus \Delta'_x$ avec $\Delta_x \cup \Delta'_x \subseteq \Delta \cup \Delta'$. Mais alors on peut appliquer l'hypothèse d'induction sur $C \leq X$ qui nous permet alors de conclure facilement.

(*st-arrow*) On est dans le cas où $C_1 \rightarrow C_2 \leq D_1 \rightarrow D_2$ et $D_1 \rightarrow D_2 \approx \text{nat}^\Delta \uplus \Delta'$. Mais il est impossible d'avoir $D_1 \rightarrow D_2 \approx \text{nat}^\Delta \uplus \Delta'$.

(*st-cor-noex*) On est dans le cas où $C^\emptyset \leq C$ et $C \approx \text{nat}^\Delta \uplus \Delta'$. Mais $C^\emptyset \approx C$ donc on conclut.

(*st-uni-merge*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

(*st-cor-merge*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

(*st-uni-arrow*) On est dans le cas où $(C_1 \rightarrow C_2) \uplus \Delta_c \leq C_1 \rightarrow C_2 \uplus \Delta_c$ et $C_1 \rightarrow C_2 \uplus \Delta_c \approx \text{nat}^\Delta \uplus \Delta'$, ce cas est impossible.

(*st-uni-ctx*) On est dans le cas où $C \uplus \Delta_c \leq D \uplus \Delta_c$ car $C \leq D$ et $D \uplus \Delta_c \approx \text{nat}^\Delta \uplus \Delta'$. Donc (lemme 4.26), il existe B tel que $D \approx B \uplus \Delta_0$ et $\text{nat}^\Delta \approx B \uplus \Delta_1$ avec $\Delta_0 \cup \Delta_c = \Delta_1 \cup \Delta'$. En appliquant le lemme 4.25 on obtient que $\Delta_1 = \emptyset$ et donc $B \approx \text{nat}^\Delta$ et $D \approx \text{nat}^\Delta \uplus \Delta_0$. On applique donc l'hypothèse d'induction sur $C \leq D$ et on obtient que $C \approx \text{nat}^{\Delta_r} \uplus \Delta'_r$ avec $\Delta_r \cup \Delta'_r \subseteq \Delta \cup \Delta_0$. Donc $C \uplus \Delta_c \approx \text{nat}^{\Delta_r} \uplus (\Delta'_r \cup \Delta_c)$ et $\Delta_r \cup \Delta'_r \cup \Delta_c \subseteq \Delta \cup \Delta_0 \cup \Delta_c$.

(*st-uni*) On est dans le cas où $C \leq C \uplus \Delta_c$ et $C \uplus \Delta_c \approx \text{nat}^\Delta \uplus \Delta'$. Donc (lemme 4.26), il existe B tel que $C \approx B \uplus \Delta_0$ et $\text{nat}^\Delta \approx B \uplus \Delta_1$ avec $\Delta_0 \cup \Delta_c = \Delta_1 \cup \Delta'$. En appliquant le lemme 4.25 on obtient que $\Delta_1 = \emptyset$ et donc $B \approx \text{nat}^\Delta$ et $D \approx \text{nat}^\Delta \uplus \Delta_0$ avec $\Delta_0 \subseteq \Delta'$ et on conclut.

(*st-cor*) On est dans le cas où $C \uplus \Delta_c \leq C^{\Delta_c}$ et $C^{\Delta_c} \approx \text{nat}^\Delta \uplus \Delta'$. Donc (lemme 4.28), il existe B tel que $C \approx B \uplus \Delta'$ et $\text{nat}^\Delta \approx B \uplus \Delta_c$. On peut alors vérifier que l'on a forcément $B \approx \text{nat}^{\Delta_0}$ avec $\Delta_0 \cup \Delta_c = \Delta$. Donc $C \approx \text{nat}^{\Delta_0} \uplus \Delta'$ et donc $C \uplus \Delta_c \approx \text{nat}^{\Delta_0} \uplus (\Delta' \cup \Delta_c)$ et on a bien $\Delta_0 \cup \Delta_c \cup \Delta' \subseteq \Delta \cup \Delta'$.

(*st-uni-cor*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

(*st-cor-arrow*) On est dans le cas d'une des égalités de l'équivalence, on conclut donc directement.

Propriétés du sous-typage

Comme nous l'avons déjà précisé, on ne peut pas dans λ_ε déduire de $A \rightarrow B \leq A' \rightarrow B'$ le fait que $A' \leq A$ et $B \leq B'$. En revanche on peut montrer la propriété plus faible suivante :

Lemme 4.32 (Inversion du sous-typage des flèches) Si A, B, A' et B' sont des types tels que $A \rightarrow B \leq A' \rightarrow B'$, alors il existe un ensemble de noms d'exceptions Δ tel que

$$A' \leq A^\Delta \quad \text{et} \quad B^\Delta \leq B'$$

Preuve :

Puisque $A' \rightarrow B' \approx A' \rightarrow B'$, on applique le lemme 4.30 pour obtenir qu'il existe X, Y, Δ et Δ' tels que $A \rightarrow B \approx (X \rightarrow Y) \uplus \Delta'$, $A' \leq X^\Delta$ et $Y^\Delta \uplus \Delta' \leq B'$. Alors, on utilise les lemmes 4.24 (qui nous donne $\Delta' = \emptyset$) et 4.21 pour montrer que $A \approx X$ et $B \approx Y$ et avec le lemme 4.14, $A \doteq X$ et $B \doteq Y$ et donc finalement on a bien $A' \leq A^\Delta$ et $B^\Delta \leq B'$.

Le lemme suivant montre d'une certaine façon que si une union est promue en un type fonctionnel par sous-typage, alors le type à droite de cette flèche mentionnera nécessairement les exceptions de l'union.

Lemme 4.33 Si A, B et C sont des types et ε un nom d'exceptions tel que $C \uplus \{\varepsilon\} \leq A \rightarrow B$, alors il existe un type D tel que

$$D \uplus \{\varepsilon\} \leq B$$

Preuve :

Puisque $A \rightarrow B \approx A \rightarrow B$, le lemme 4.30 nous dit que $C \uplus \{\varepsilon\} \approx (A' \rightarrow B') \uplus \Delta'$ avec $B'^\Delta \uplus \Delta' \leq B$ (pour un certain Δ). Alors le lemme 4.27 nous dit que $C \approx (A' \rightarrow B') \uplus \Delta'_0$ avec $\Delta' = \Delta'_0 \cup \{\varepsilon\}$ et donc $B'^{\Delta_0} \uplus (\Delta'_0 \cup \{\varepsilon\}) \leq B$. Finalement en prenant $D = B'^{\Delta_0} \uplus (\Delta'_0)$ on a bien $D \uplus \{\varepsilon\} \leq B$.

Le lemme qui suit montre qu'une flèche (respectivement, le type des entiers) ne peut être le sous-type direct d'une union.

Lemme 4.34 Si A, B et C sont des types, Δ et Δ' sont des ensembles de noms d'exceptions, alors on a les implications suivantes :

$$\begin{aligned} B \rightarrow C \leq A \uplus \Delta &\Rightarrow B \rightarrow C \leq A \\ \text{nat}^{\Delta'} \leq A \uplus \Delta &\Rightarrow \text{nat}^{\Delta'} \leq A \end{aligned}$$

Preuve :

On a deux cas :

- Si $B \rightarrow C \leq A \uplus \Delta$, on applique le lemme 4.29 et on a que soit $B \rightarrow C \leq A$ et on conclut, soit $B \rightarrow C \approx D \uplus \Delta'$ avec $D \leq A$. Mais avec le lemme 4.24, on sait que $\Delta' = \emptyset$ et on conclut également.
- Si $\text{nat}^{\Delta'} \leq A \uplus \Delta$, on applique le lemme 4.29 et on a que soit $\text{nat}^{\Delta'} \leq A$ et on conclut, soit $\text{nat}^{\Delta'} \approx D \uplus \Delta''$ avec $D \leq A$. Mais avec le lemme 4.25, on sait que $\Delta' = \emptyset$ et on conclut également.

Enfin, la dernière propriété du sous-typage dont on a besoin est la suivante :

Lemme 4.35 *Si A et B sont des types et ε et ε' deux noms d'exceptions différents tels que $A \uplus \{\varepsilon\} \leq B \uplus \{\varepsilon'\}$, alors il existe un type C tel que*

$$C \uplus \{\varepsilon\} \leq B$$

Preuve :

Puisque $B \uplus \{\varepsilon'\} \approx B \uplus \{\varepsilon'\}$, on applique le lemme 4.29 qui nous dit que

- Soit $A \uplus \{\varepsilon\} \leq B$ et on conclut directement.
- Soit $A \uplus \{\varepsilon\} \approx C \uplus \Delta$ avec $C \leq B$ et $\Delta \subseteq \{\varepsilon'\}$. On applique le lemme 4.26 pour obtenir qu'il existe X , Δ_A et Δ_C tels que $A \approx X \uplus \Delta_A$, $C \approx X \uplus \Delta_C$ et $\Delta_A \cup \{\varepsilon\} = \Delta_C \cup \Delta$. Alors puisque $\Delta \subseteq \{\varepsilon'\}$, on a forcément $\varepsilon \in \Delta_C$ et donc $X \uplus \Delta_C \doteq (X \uplus \Delta'_C) \uplus \{\varepsilon\} \leq B$ (en utilisant le lemme 4.14).

Remarquons que la conclusion du lemme ci-dessus n'est pas que $A \uplus \{\varepsilon\} \leq B$. En effet, si A mentionne ε' , ceci n'est pas vrai (et C est alors un sous-type de A qui ne mentionne pas ε').

Propriétés du typage

On commence par montrer le lemme de corruption pour les jugements de typage :

Lemme 4.36 *Si t est un terme, A et B deux types, Γ un environnement de typage et x une variable de terme tel que $\Gamma, x : A \vdash t : B$, alors pour tout ensemble de noms d'exceptions Δ , on a*

$$\Gamma, x : A^\Delta \vdash t : B^\Delta$$

Preuve :

On va en fait montrer que si $\Gamma, x : A, \Omega \vdash t : B$ (où Ω est un environnement de typage quelconque), alors $\Gamma, x : A^\Delta, \Omega \vdash t : B^\Delta$, et ce par induction sur $\Gamma, x : A, \Omega \vdash t : B$:

(t-ax) : Donc $\Gamma, x : A, \Omega \vdash y : B$ avec $(y : B) \in \Gamma, x : A, \Omega$ et deux cas de figure peuvent se présenter : $x = y$ ou $x \neq y$. Si $x = y$, alors $B = A$ et on conclut directement. Si $x \neq y$, alors $\Gamma, x : A^\Delta, \Omega \vdash y : B$, mais on peut également conclure vu que $B \leq B^\Delta$ quel que soit Δ .

(t-abs) : Donc $\Gamma, x : A, \Omega, y : B \vdash t : C$ et on doit montrer que $\Gamma, x : A^\Delta, \Omega \vdash \lambda y. t : (B \rightarrow C)^\Delta$. Alors par hypothèse d'induction, on a que $\Gamma, x : A^\Delta, \Omega, y : B \vdash t : C^\Delta$ et donc que $\Gamma, x : A^\Delta, \Omega \vdash \lambda y. t : B \rightarrow C^\Delta$. Mais on peut alors conclure car on a que

$$B \rightarrow C^\Delta \leq (B \rightarrow C^\Delta)^\Delta \leq B^\Delta \rightarrow C^\Delta \leq (B \rightarrow C)^\Delta.$$

(t-app) : Donc $\Gamma, x : A, \Omega \vdash t : B \rightarrow C$ et $\Gamma, x : A, \Omega \vdash u : B$ et on doit montrer que $\Gamma, x : A^\Delta, \Omega \vdash t u : C^\Delta$. Par hypothèse d'induction, on a que $\Gamma, x : A^\Delta, \Omega \vdash t : (B \rightarrow C)^\Delta$ et $\Gamma, x : A^\Delta, \Omega \vdash u : B^\Delta$. Mais comme $(B \rightarrow C)^\Delta \leq B^\Delta \rightarrow C^\Delta$, finalement on a bien que $\Gamma, x : A^\Delta, \Omega \vdash t u : C^\Delta$.

(t-zero) : Puisque $\text{nat} \leq \text{nat}^\Delta$, il est facile de montrer que $\Gamma, x : A^\Delta, \Omega \vdash 0 : \text{nat}^\Delta$.

- (*t-succ*) : Comme $A \leq A^\Delta$ quelque soit A , il est facile de montrer que $\Gamma, x : A^\Delta, \Omega \vdash S : (\text{nat} \rightarrow \text{nat})^\Delta$.
- (*t-rec*) : Comme $A \leq A^\Delta$ quelque soit A , il est facile de montrer que $\Gamma, x : A^\Delta, \Omega \vdash \text{rec} : (B \uplus \Delta' \rightarrow (\text{nat}^{\Delta'} \rightarrow B \uplus \Delta' \rightarrow B \uplus \Delta') \rightarrow \text{nat}^{\Delta'} \uplus \Delta'' \rightarrow B \uplus (\Delta' \cup \Delta''))^\Delta$.
- (*t-raise*) : Comme $A \leq A^\Delta$ quelque soit A , il est facile de montrer que $\Gamma, x : A^\Delta, \Omega \vdash \text{raise } \varepsilon : (B \uplus \{\varepsilon\})^\Delta$.
- (*t-try*) : Donc $\Gamma, x : A, \Omega \vdash t : B \uplus \{\varepsilon\}$ et $\Gamma, x : A, \Omega \vdash u : B$ et on doit montrer que $\Gamma, x : A^\Delta, \Omega \vdash \text{try } t \text{ with } \varepsilon \mapsto u : B^\Delta$. Or, par hypothèse d'induction, on a que $\Gamma, x : A^\Delta, \Omega \vdash t : (B \uplus \{\varepsilon\})^\Delta$ et $\Gamma, x : A^\Delta, \Omega \vdash u : B^\Delta$. Mais comme $(B \uplus \{\varepsilon\})^\Delta \leq B^\Delta \uplus \{\varepsilon\}$, on peut conclure.
- (*t-subtyp*) : Donc $\Gamma, x : A, \Omega \vdash t : C$ avec $C \leq B$. Par hypothèse d'induction, on a que $\Gamma, x : A^\Delta, \Omega \vdash t : C^\Delta$. De plus, grâce au lemme 4.10, on a que $C^\Delta \leq B^\Delta$ et donc finalement, on a bien que $\Gamma, x : A^\Delta, \Omega \vdash t : B^\Delta$.

Remarquons que ce lemme est particulièrement important. Il contient en substance le principe de la sémantique des flèches que nous utiliserons pour les modèles de réalisabilité des chapitre 5 et 7.

Lemmes d'inversions On va maintenant démontrer des lemmes d'inversions pour toutes les constructions de termes de λ_ε .

Lemme 4.37 (Inversion de l'abstraction) Si t est un terme, C un type et Γ un environnement de typage tel que $\Gamma \vdash \lambda x. t : C$, alors il existe des types A et B tel que

$$\begin{aligned} \Gamma, x : A &\vdash t : B \\ A \rightarrow B &\leq C \end{aligned}$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash \lambda x. t : C$. Les seules règles qui ont pu être appliquées sont :

(*t-abs*) Mais alors, il existe des types A et B tel que $C \equiv A \rightarrow B$ et $\Gamma, x : A \vdash t : B$, et donc en particulier $C \leq A \rightarrow B$

(*t-subtyp*) On a donc que $\Gamma \vdash \lambda x. t : D$ et $D \leq C$. Par hypothèse d'induction, il existe des types A et B tel que $\Gamma, x : A \vdash t : B$ et $A \rightarrow B \leq D$. Mais alors on a bien $A \rightarrow B \leq C$.

Lemme 4.38 (Inversion de l'application) Si t et u sont deux termes, B un type et Γ un environnement de typage tel que $\Gamma \vdash t u : B$, alors il existe un type A tel que

$$\begin{aligned} \Gamma &\vdash t : A \rightarrow B \\ \Gamma &\vdash u : A \end{aligned}$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash t u : B$. Les seules règles qui ont pu être appliquées sont :

(*t-app*) On a directement le résultat.

(*t-subtyp*) On a donc que $\Gamma \vdash t u : C$ avec $C \leq B$. Alors, par hypothèse d'induction, il existe un type A tel que $\Gamma \vdash t : A \rightarrow C$ et $\Gamma \vdash u : A$. Mais comme $A \rightarrow C \leq A \rightarrow B$ (grâce à la règle (*st-arrow*)), on a bien aussi que $\Gamma \vdash t : A \rightarrow B$.

Lemme 4.39 (Inversion du try) Si t et u sont deux termes, B un type, Γ un environnement de typage et ε un nom d'exception tel que $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$, alors

$$\begin{aligned} \Gamma \vdash t : A \uplus \{\varepsilon\} \\ \Gamma \vdash u : A \end{aligned}$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$. Les seules règles qui ont pu être appliquées sont :

(*t-try*) On a directement le résultat.

(*t-subtyp*) On a donc que $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : B$ avec $B \leq A$. Alors, par hypothèse d'induction, $\Gamma \vdash t : B \uplus \{\varepsilon\}$ et $\Gamma \vdash u : B$. Et donc, $\Gamma \vdash u : A$ et puisque $B \uplus \{\varepsilon\} \leq A \uplus \{\varepsilon\}$ (grâce à la règle (*st-uni-ctx*)), $\Gamma \vdash t : A \uplus \{\varepsilon\}$.

Lemme 4.40 (Inversion du raise) Si A est un type, Γ un environnement de typage et ε un nom d'exception tel que $\Gamma \vdash \text{raise } \varepsilon : A$, alors il existe un type B tel que

$$B \uplus \{\varepsilon\} \leq A$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash \text{raise } \varepsilon : A$. Les seules règles qui ont pu être appliquées sont :

(*t-raise*) On a directement le résultat puisque alors $A = B \uplus \{\varepsilon\}$ pour un certain B .

(*t-subtyp*) On a également directement le résultat par transitivité du sous-typage.

Lemme 4.41 (Inversion du successeur) Si A et B sont deux types et Γ un environnement de typage tel que $\Gamma \vdash S : A \rightarrow B$, alors il existe un ensemble de noms d'exceptions Δ tel que

$$\begin{aligned} A &\leq \text{nat}^\Delta \\ \text{nat}^\Delta &\leq B \end{aligned}$$

Preuve :

On commence par montrer par induction sur la dérivation du jugement $S \leq A \rightarrow B$ que $\text{nat} \rightarrow \text{nat} \leq A \rightarrow B$. Puis on conclut par application du lemme d'inversion du sous-typage des flèches (lemme 4.32).

Lemme 4.42 (Inversion du récursur) *Si A, B, C et D sont des types et Γ un environnement de typage tel que $\Gamma \vdash \text{rec} : A \rightarrow B \rightarrow C \rightarrow D$, alors il existe un type X et des ensembles de noms d'exceptions Δ, Δ' et Δ'' tel que*

$$\begin{aligned} A &\leq X^\Delta \uplus \Delta' \\ B &\leq \text{nat}^{\Delta \cup \Delta'} \rightarrow X^\Delta \uplus \Delta' \rightarrow X^\Delta \uplus \Delta' \\ C &\leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta'' \\ X^\Delta \uplus (\Delta' \cup \Delta'') &\leq D \end{aligned}$$

Preuve :

On commence par montrer par une simple induction sur la dérivation du jugement $\Gamma \vdash \text{rec} : A \rightarrow B \rightarrow C \rightarrow D$ qu'il existe un type X et un ensemble de noms d'exceptions Δ' tel que

$$X \uplus \Delta' \rightarrow (\text{nat}^{\Delta'} \rightarrow X \uplus \Delta' \rightarrow X \uplus \Delta') \rightarrow \text{nat}^{\Delta'} \uplus \Delta'' \rightarrow X \uplus (\Delta' \cup \Delta'') \leq A \rightarrow B \rightarrow C \rightarrow D.$$

Mais alors, en appliquant plusieurs fois le lemme d'inversion du sous-typage des flèches (lemme 4.32), on montre qu'il existe des ensembles de noms d'exceptions $\Delta_1, \Delta_2, \Delta_3$ tel que :

$$\begin{aligned} A &\leq (X \uplus \Delta')^{\Delta_1} \\ B &\leq (\text{nat}^{\Delta'} \rightarrow X \uplus \Delta' \rightarrow X \uplus \Delta')^{\Delta_1 \cup \Delta_2} \\ C &\leq \text{nat}^{\Delta' \cup \Delta_1 \cup \Delta_2 \cup \Delta_3} \uplus \Delta'' \\ (X \uplus (\Delta' \cup \Delta''))^{\Delta_1 \cup \Delta_2 \cup \Delta_3} &\leq D \end{aligned}$$

On conclut alors facilement en prenant pour ensemble de noms d'exceptions $\Delta = \Delta_1 \cup \Delta_2 \cup \Delta_3$.

Enfin, on montre un dernier lemme qui exprime au niveau du typage que les valeurs régulières ne lèvent pas d'exceptions :

Lemme 4.43 *Si v est une valeur régulière, A un type, Γ un environnement de typage et ε un nom d'exception tel que $\Gamma \vdash v : A \uplus \{\varepsilon\}$, alors*

$$\Gamma \vdash v : A$$

Preuve :

On va raisonner par cas sur la valeur régulière v . Par définition, v est l'un des termes suivants :

- $\lambda x. t$: par inversion de l'abstraction (lemme 4.37), il existe des types B et C tel que $\Gamma \vdash \lambda x. t : B \rightarrow C$ et $B \rightarrow C \leq A \uplus \{\varepsilon\}$. Mais en appliquant le lemme 4.34, on a que $B \rightarrow C \leq A$ et donc bien que $\Gamma \vdash \lambda x. t : A$.

- 0 : Il est facile de montrer (par induction sur $\Gamma \vdash 0 : A \uplus \{\varepsilon\}$) qu'alors $\text{nat} \leq A \uplus \{\varepsilon\}$. Mais donc, en appliquant le lemme 4.34, on a $\text{nat} \leq A$ et finalement $\Gamma \vdash 0 : A$.
- S : Il est facile de montrer (par induction sur $\Gamma \vdash S : A \uplus \{\varepsilon\}$) qu'alors $\text{nat} \rightarrow \text{nat} \leq A \uplus \{\varepsilon\}$. Mais donc, en appliquant le lemme 4.34, on a $\text{nat} \rightarrow \text{nat} \leq A$ et finalement $\Gamma \vdash S : A$.
- S t : Grâce aux lemmes d'inversion de l'application et d'inversion du successeur (lemmes 4.38 et 4.41), on montre qu'il existe un ensemble de noms d'exceptions Δ tel que $\Gamma \vdash S t : \text{nat}^\Delta$ et $\text{nat}^\Delta \leq A \uplus \{\varepsilon\}$. Mais alors, en appliquant le lemme 4.34, on a $\text{nat}^\Delta \leq A$ et donc $\Gamma \vdash S t : A$.
- rec : Il est facile de montrer (par induction sur $\Gamma \vdash \text{rec} : A \uplus \{\varepsilon\}$) qu'alors il existe un type B et des ensembles de noms d'exceptions Δ et Δ' tel que $\Gamma \vdash \text{rec} : B \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow B \uplus \Delta \rightarrow B \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow B \uplus (\Delta \cup \Delta')$ et $B \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow B \uplus \Delta \rightarrow B \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow B \uplus (\Delta \cup \Delta') \leq A \uplus \{\varepsilon\}$. Mais alors, en appliquant le lemme 4.34, on a forcément que $B \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow B \uplus \Delta \rightarrow B \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow B \uplus \Delta \cup \Delta' \leq A$ et donc $\Gamma \vdash \text{rec} : A$.
- rec t : Le lemme 4.38 d'inversion de l'application nous dit qu'il existe un type B tel que $\Gamma \vdash \text{rec} : B \rightarrow A \uplus \{\varepsilon\}$ et $\Gamma \vdash t : B$. Mais il est alors facile de montrer qu'il existe un type C et des ensembles de noms d'exceptions Δ_1 et Δ_2 tel que $\Gamma \vdash \text{rec} : C \uplus \Delta_1 \rightarrow (\text{nat}^{\Delta_1} \rightarrow C \uplus \Delta_1 \rightarrow C \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow C \uplus \Delta_1 \cup \Delta_2$ et $C \uplus \Delta_1 \rightarrow (\text{nat}^{\Delta_1} \rightarrow C \uplus \Delta_1 \rightarrow C \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow C \uplus \Delta_1 \cup \Delta_2 \leq B \rightarrow A \uplus \{\varepsilon\}$. On peut alors appliquer le lemme d'inversion du sous-typage des flèches (lemme 4.32) pour obtenir qu'il existe un ensemble de noms d'exceptions Δ tel que $((\text{nat}^{\Delta_1} \rightarrow C \uplus \Delta_1 \rightarrow C \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow C \uplus \Delta_1 \cup \Delta_2)^\Delta \leq A \uplus \{\varepsilon\}$ et $B \leq (C \uplus \Delta_1)^\Delta$. On peut alors utiliser le lemme 4.34 pour montrer que $((\text{nat}^{\Delta_1} \rightarrow C \uplus \Delta_1 \rightarrow C \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow C \uplus \Delta_1 \cup \Delta_2)^\Delta \leq A$. Mais donc $\Gamma \vdash t : (C \uplus \Delta_1)^\Delta$ et $\Gamma \vdash \text{rec} t : ((\text{nat}^{\Delta_1} \rightarrow C \uplus \Delta_1 \rightarrow C \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow C \uplus \Delta_1 \cup \Delta_2)^\Delta$ et on peut bien conclure que $\Gamma \vdash \text{rec} t : A$.
- rec t t' : Le lemme d'inversion de l'application (lemme 4.38) nous dit qu'il existe des type B et C tel que $\Gamma \vdash \text{rec} : B \rightarrow C \rightarrow A \uplus \{\varepsilon\}$, $\Gamma \vdash t : B$ et $\Gamma \vdash t' : C$. Mais il est alors facile de montrer qu'il existe un type D et des ensembles de noms d'exceptions Δ_1 et Δ_2 tel que $\Gamma \vdash \text{rec} : D \uplus \Delta_1 \rightarrow (\text{nat}^{\Delta_1} \rightarrow D \uplus \Delta_1 \rightarrow D \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow D \uplus (\Delta_1 \cup \Delta_2)$ et $D \uplus \Delta_1 \rightarrow (\text{nat}^{\Delta_1} \rightarrow D \uplus \Delta_1 \rightarrow D \uplus \Delta_1) \rightarrow \text{nat}^{\Delta_1} \uplus \Delta_2 \rightarrow D \uplus (\Delta_1 \cup \Delta_2) \leq B \rightarrow C \rightarrow A \uplus \{\varepsilon\}$. On peut alors appliquer le lemme d'inversion du sous-typage des flèches (lemme 4.32) pour obtenir qu'il existe des ensembles de noms d'exceptions Δ et Δ' tel que $\text{nat}^{\Delta_1 \cup \Delta \cup \Delta'} \uplus \Delta_2 \rightarrow (D \uplus \Delta_1)^{\Delta \cup \Delta'} \leq A \uplus \{\varepsilon\}$, $C \leq (\text{nat}^{\Delta_1} \rightarrow D \uplus \Delta_1 \rightarrow D \uplus \Delta_1)^{\Delta \cup \Delta'}$ et $B \leq D^{\Delta_1 \cup \Delta}$. On peut alors utiliser le lemme 4.34 pour montrer que $\text{nat}^{\Delta_1 \cup \Delta \cup \Delta'} \uplus \Delta_2 \rightarrow (D \uplus (\Delta_1 \cup \Delta_2))^{\Delta \cup \Delta'} \leq A$. Mais donc $\Gamma \vdash t : (D \uplus \Delta_1)^{\Delta \cup \Delta'}$ et $\Gamma \vdash t' : (\text{nat}^{\Delta_1} \rightarrow D \uplus \Delta_1 \rightarrow D \uplus \Delta_1)^{\Delta \cup \Delta'}$ et donc $\Gamma \vdash \text{rec} t t' : \text{nat}^{\Delta_1 \cup \Delta \cup \Delta'} \uplus \Delta_2 \rightarrow (C \uplus (\Delta_1 \cup \Delta_2))^{\Delta \cup \Delta'}$ et on peut bien conclure que $\Gamma \vdash \text{rec} t t' : A$.

4.4.2 Préservation du typage par réduction

On commence par montrer que la notion de réduction vérifie la propriété de préservation du typage :

Lemme 4.44 (Préservation du typage par réduction à la racine) *Si t et t' sont deux termes, A un type et Γ un environnement de typage tel que $\Gamma \vdash t : A$ et $t \rightarrow t'$, alors*

$$\Gamma \vdash t' : A$$

Preuve :

On va procéder par cas sur la réduction $t \rightarrow t'$:

- (β) On a $\Gamma \vdash (\lambda x. t) u : B$ et on doit montrer que $\Gamma \vdash t\{x := u\} : B$. Avec le lemme d'inversion de l'application (lemme 4.38), il existe un type A tel que $\Gamma \vdash \lambda x. t : A \rightarrow B$ et $\Gamma \vdash u : A$. Alors, avec le lemme d'inversion de l'abstraction (lemme 4.37), il existe deux types A' et B' tel que $\Gamma, x : A' \vdash t : B'$ et $A' \rightarrow B' \leq A \rightarrow B$. Mais alors, on utilise le lemme 4.32 afin d'obtenir $A \leq A'^\Delta$ et $B'^\Delta \leq B$ pour un certain ensemble de noms d'exceptions Δ . Donc d'une part, $\Gamma \vdash u : A'^\Delta$ et d'autre part, avec le lemme 4.36, $\Gamma, x : A'^\Delta \vdash t : B'^\Delta$. On utilise alors le lemme 4.6 pour montrer $\Gamma \vdash t\{x := u\} : B'^\Delta$, d'où on conclut que $\Gamma \vdash t\{x := u\} : B$.
- (*rapp*) On a $\Gamma \vdash (\text{raise } \varepsilon) u : B$ et on doit montrer que $\Gamma \vdash \text{raise } \varepsilon : B$. Avec le lemme d'inversion de l'application (lemme 4.38), il existe un type A tel que $\Gamma \vdash \text{raise } \varepsilon : A \rightarrow B$ et $\Gamma \vdash u : A$. Et avec le lemme d'inversion du *raise* (lemme 4.40), il existe un type C tel que $C \uplus \{\varepsilon\} \leq A \rightarrow B$. Alors, grâce au lemme 4.33, on sait qu'il existe un type D tel que $D \uplus \{\varepsilon\} \leq B$. Mais alors quel que soit le type D , on a toujours $\Gamma \vdash \text{raise } \varepsilon : D \uplus \{\varepsilon\}$.
- (*catch*) On a $\Gamma \vdash \text{try raise } \varepsilon \text{ with } \varepsilon \mapsto u : A$ et on doit montrer que $\Gamma \vdash u : A$, mais ceci est une application directe du lemme d'inversion du *try* (lemme 4.39).
- (*trai*) On a $\Gamma \vdash \text{try raise } \varepsilon' \text{ with } \varepsilon \mapsto u : A$ et on doit montrer que $\Gamma \vdash \text{raise } \varepsilon' : A$. Avec le lemme d'inversion du *try* (lemme 4.39), on a $\Gamma \vdash \text{raise } \varepsilon' : A \uplus \{\varepsilon\}$. Alors, avec le lemme d'inversion du *raise* (lemme 4.40), il existe un type C tel que $C \uplus \varepsilon' \leq A \uplus \{\varepsilon\}$. Mais alors, grâce au lemme 4.35 on a $C \uplus \varepsilon' \leq A$ et donc on a aussi $\Gamma \vdash \text{raise } \varepsilon' : A$.
- (*tval*) On a $\Gamma \vdash \text{try } v \text{ with } \varepsilon \mapsto u : A$ où v est une valeur régulière, et on doit montrer que $\Gamma \vdash v : A$. Or avec le lemme d'inversion du *try* (lemme 4.39), on a $\Gamma \vdash v : A \uplus \{\varepsilon\}$ et on conclut alors avec le lemme 4.43.
- (*recz*) On a $\Gamma \vdash \text{rec } z f 0 : A$ et on doit montrer que $\Gamma \vdash z : A$. Avec plusieurs applications du lemme d'inversion de l'application (lemme 4.38), on montre qu'il existe des type B, C et D tel que $\Gamma \vdash \text{rec} : D \rightarrow C \rightarrow B \rightarrow A$ et (entre autre) $\Gamma \vdash z : D$. Mais alors, avec le lemme d'inversion du récursur (lemme 4.42), on obtient $D \leq A$ et on a donc $\Gamma \vdash z : A$.
- (*recs*) On a $\Gamma \vdash \text{rec } z f (S n) : A$ et on doit montrer que $\Gamma \vdash f n (\text{rec } z f n) : A$. Avec plusieurs applications du lemme d'inversion de l'application (lemme 4.38), on montre qu'il existe des type B, C et D tel que $\Gamma \vdash \text{rec} : D \rightarrow C \rightarrow B \rightarrow A$, $\Gamma \vdash z : D$, $\Gamma \vdash f : C$ et $\Gamma \vdash (S n) : B$. Mais alors, avec le lemme d'inversion du récursur (lemme 4.42), il existe un type X et des ensembles de noms d'exceptions Δ, Δ' et Δ'' tel que

$$\begin{aligned} D &\leq X^\Delta \uplus \Delta' \\ C &\leq \text{nat}^{\Delta \cup \Delta'} \rightarrow X^\Delta \uplus \Delta' \rightarrow X^\Delta \uplus \Delta' \\ B &\leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta'' \\ X^\Delta \uplus (\Delta' \cup \Delta'') &\leq A \end{aligned}$$

Donc $\Gamma \vdash S n : \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$ et en utilisant le lemme d'inversion de l'application (lemme 4.38) couplé au lemme d'inversion du successeur (lemme 4.41), il existe un type E et ensemble de noms d'exceptions Δ_1 tel que $\Gamma \vdash n : E$, $E \leq \text{nat}^{\Delta_1}$ et $\text{nat}^{\Delta_1} \leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$. Alors, en utilisant le lemme 4.34 sur cette dernière inégalité de sous-typage, on obtient $\text{nat}^{\Delta_1} \leq \text{nat}^{\Delta \cup \Delta'}$ et donc finalement $\Gamma \vdash n : \text{nat}^{\Delta \cup \Delta'}$. Par ailleurs, par (*t-rec*),

$$\Gamma \vdash \text{rec} : X \uplus \Delta' \rightarrow (\text{nat}^{\Delta'} \rightarrow X \uplus \Delta' \rightarrow X \uplus \Delta') \rightarrow \text{nat}^{\Delta'} \rightarrow X \uplus \Delta'$$

et donc par corruption, on a

$$\Gamma \vdash \text{rec} : X^\Delta \uplus \Delta' \rightarrow (\text{nat}^{(\Delta' \cup \Delta)} \rightarrow X^\Delta \uplus \Delta' \rightarrow X^\Delta \uplus \Delta') \rightarrow \text{nat}^{(\Delta' \cup \Delta)} \rightarrow X^\Delta \uplus \Delta'.$$

Il s'ensuit que $\Gamma \vdash \text{rec } z f n : X^\Delta \uplus \Delta'$ et donc $\Gamma \vdash f n (\text{rec } z f n) : X^\Delta \uplus \Delta'$. Et puisque $X^\Delta \uplus (\Delta' \cup \Delta'') \leq A$, on a finalement $\Gamma \vdash f n (\text{rec } z f n) : A$.

(*recr*) On a $\Gamma \vdash \text{rec } z \ f \ (\text{raise } \varepsilon) : A$ et on doit montrer que $\Gamma \vdash \text{raise } \varepsilon : A$. Avec plusieurs applications du lemme d'inversion de l'application (lemme 4.38) puis du lemme d'inversion du récursur (lemme 4.42), il existe un type X et des ensembles de noms d'exceptions Δ , Δ' et Δ'' tel que $\Gamma \vdash \text{raise } \varepsilon : \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$ et $X^\Delta \uplus (\Delta' \cup \Delta'') \leq A$. Alors avec le lemme d'inversion du *raise* il existe un type C tel que $C \uplus \{\varepsilon\} \leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$ et il est alors simple de montrer qu'alors $\varepsilon \in \Delta \cup \Delta' \cup \Delta''$ d'où on a $\Gamma \vdash \text{raise } \varepsilon : X^\Delta \uplus (\Delta' \cup \Delta'')$. Finalement, on obtient $\Gamma \vdash \text{raise } \varepsilon : A$.

On a maintenant tous les éléments pour montrer que le langage λ_ε vérifie la propriété de préservation du typage par réduction :

Théorème 4.45 (Préservation du typage par réduction) *Si t et t' sont deux termes, A un type et Γ un environnement de typage tel que $\Gamma \vdash t : A$ et $t \rightarrow t'$, alors*

$$\Gamma \vdash t' : A$$

Preuve :

On va procéder par induction sur la dérivation de $t \rightarrow t'$:

(*red*) On est exactement dans le cadre du lemme 4.44.

(*red_λ*) On a $\Gamma \vdash \lambda x. t : C$ et on doit montrer que $\Gamma \vdash \lambda x. t' : C$. Avec le lemme d'inversion de l'abstraction (lemme 4.37) il existe des types A et B tel que $\Gamma, x : A \vdash t : B$ et $A \rightarrow B \leq C$. Mais alors, l'hypothèse d'induction nous donne $\Gamma, x : A \vdash t' : B$, d'où on déduit $\Gamma \vdash \lambda x. t' : A \rightarrow B$ et conclut.

(*red_{app1}*) On a $\Gamma \vdash t u : B$ et on doit montrer que $\Gamma \vdash t' u : B$. Avec le lemme d'inversion de l'application (lemme 4.38), il existe un terme A tel que $\Gamma \vdash t : A \rightarrow B$ et $\Gamma \vdash u : A$. Donc par hypothèse d'induction, $\Gamma \vdash t' : A \rightarrow B$ et on conclut.

(*red_{app2}*) On a $\Gamma \vdash t u : B$ et on doit montrer que $\Gamma \vdash t u' : B$. Avec le lemme d'inversion de l'application (lemme 4.38), il existe un terme A tel que $\Gamma \vdash t : A \rightarrow B$ et $\Gamma \vdash u : A$. Donc par hypothèse d'induction, $\Gamma \vdash u' : A$ et on conclut.

(*red_{try1}*) On a $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$ et on doit montrer que $\Gamma \vdash \text{try } t' \text{ with } u \mapsto : A$. Avec le lemme d'inversion du *try* (lemme 4.39), on a $\Gamma \vdash t : A \uplus \{\varepsilon\}$ et $\Gamma \vdash u : A$. Donc par hypothèse d'induction, $\Gamma \vdash t' : A \uplus \{\varepsilon\}$ et on conclut.

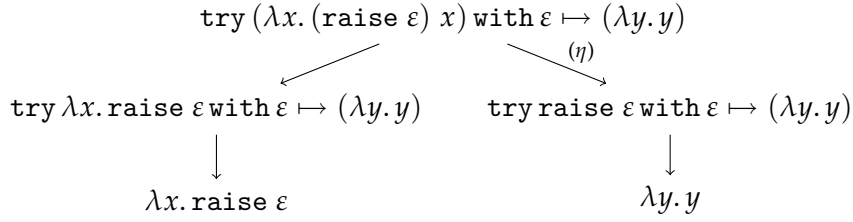
(*red_{try2}*) On a $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$ et on doit montrer que $\Gamma \vdash \text{try } t \text{ with } u; \mapsto : A$. Avec le lemme d'inversion du *try* (lemme 4.39), on a $\Gamma \vdash t : A \uplus \{\varepsilon\}$ et $\Gamma \vdash u : A$. Donc par hypothèse d'induction, $\Gamma \vdash u' : A$ et on conclut.

4.4.3 Préservation du typage par η -réduction

Le calcul λ_ε , et en particulier son mécanisme d'exception, ne nécessite pas la présence de la règle de η -réduction. D'ailleurs, l'introduction de cette règle invalide la propriété de confluence du calcul. En effet, si on ajoute la règle de réduction suivante :

$$\lambda x. t \ x \rightarrow_\eta t \quad (x \notin \text{FV}(t)) \quad (\eta)$$

alors la paire critique suivante n'est pas résoluble :



Néanmoins, les types de λ_ε sont préservés par η -réduction :

Théorème 4.46 (Préservation du typage par η -réduction) *Si t est un terme, A un type, Γ un environnement de typage et x une variable de terme tels que $\Gamma \vdash \lambda x. t x : A$ et $x \notin \text{FV}(t)$, alors*

$$\Gamma \vdash t : A$$

Preuve :

Supposons que $\Gamma \vdash \lambda x. t x : A$ avec $x \notin \text{FV}(t)$. Par inversion de l'abstraction (lemme 4.37), il existe des types B et C tels que $B \rightarrow C \leq A$ et $\Gamma, x : B \vdash t x : C$. Alors, par inversion de l'application (lemme 4.38), il existe un type D tel que $\Gamma, x : B \vdash t : D \rightarrow C$ et $\Gamma, x : B \vdash x : D$. Par une simple induction sur la dérivation de $\Gamma, x : B \vdash x : D$, on montre alors qu'on a $B \leq D$. Par conséquent et en utilisant la règle de sous-typage (*st-arrow*), on a $D \rightarrow C \leq B \rightarrow C$ et donc $\Gamma, x : B \vdash t : B \rightarrow C$. Enfin, puisque $B \rightarrow C \leq A$, on a $\Gamma, x : B \vdash t : A$ et puisque $x \notin \text{FV}(t)$, on a $\Gamma \vdash t : A$ en utilisant le lemme 4.7.

4.4.4 Progression du calcul

On montre que l'exécution d'un programme bien typé de λ_ε n'est jamais bloquée, c'est à dire que tout terme bien typé dans l'environnement vide (et donc clos) qui n'est pas une valeur est réductible.

Théorème 4.47 (Progression du calcul) *Si t est un terme et A un type tels que $\vdash t : A$, alors*

- Soit t est une valeur ;
- Soit il existe un terme t' tel que $t \rightarrow_{wh} t'$.

Preuve :

On raisonne par induction sur la dérivation de $\vdash t : A$. Si la dernière règle utilisée est :

(t-ax) Cette règle ne peut avoir été appliquée puisque l'environnement de typage est vide.

(t-abs) Alors $t = \lambda x. t'$ et est donc une valeur.

(t-app) On est dans le cas où $t = u v$ avec $\vdash u : A \rightarrow B$ et $\vdash v : A$. On raisonne alors par cas sur la forme du terme u :

- Si $u = \lambda x. u'$, alors $u v \rightarrow u' \{x := v\}$ par (β) .
- Si $u = \text{raise } \varepsilon$, alors $u v \rightarrow \text{raise } \varepsilon$ par (rapp) .
- Si u est S , rec , $\text{rec } z$, alors $u v$ est une valeur.

- Si $u = S n$, alors $\vdash S n : A \rightarrow B$ et par inversion de l'application (lemme 4.38) puis du successeur (lemme 4.41), on obtient qu'il existe Δ tel que $\text{nat}^\Delta \leq A \rightarrow B$. Alors le lemme 4.30 nous dit qu'il existe A' , B' et Δ' tel que $\text{nat}^\Delta \approx (A' \rightarrow B') \uplus \Delta'$. Mais on montre alors facilement que ce cas est impossible (en effet, ces types ne peuvent avoir un réduit commun puisque nat^Δ n'est pas une flèche).
- Si $u = \text{rec } z f$, alors $\vdash \text{rec } z f : A \rightarrow B$. En appliquant l'inversion de l'application (lemme 4.38) et l'inversion du récursur (lemme 4.42), on obtient qu'il existe Δ et Δ' tel que $A \leq \text{nat}^\Delta \uplus \Delta'$. Alors, $\vdash v : \text{nat}^\Delta \uplus \Delta'$ et
 - Soit v n'est pas une valeur, mais alors par hypothèse d'induction, il existe v' tel que $v \rightarrow v'$ et donc $u v \rightarrow u v'$ par (*hred_{rec}*).
 - Soit v est une valeur et avec le lemme 4.48, v est soit 0, soit $S n$, soit $\text{raise } \varepsilon$ et dans tous les cas $u v$ se réduit (par (*recz*), (*recs*) ou (*recr*)).
- Sinon, u n'est pas une valeur et donc par hypothèse d'induction appliquée à $\vdash u : A \rightarrow B$, on obtient qu'il existe u' tel que $u \rightarrow u'$. Donc $u v \rightarrow u' v$ par (*hred_{app}*).

(*t-zero*) Alors $t = 0$ et est donc une valeur.

(*t-succ*) Alors $t = S$ et est donc une valeur.

(*t-rec*) Alors $t = \text{rec}$ et est donc une valeur.

(*t-raise*) Alors $t = \text{raise } \varepsilon$ et est donc une valeur.

(*t-try*) Alors $t = \text{try } u \text{ with } \varepsilon \mapsto v$ et on a $\vdash u : A \uplus \{\varepsilon\}$ et $\vdash v : A$. On applique alors l'hypothèse d'induction sur $\vdash u : A \uplus \{\varepsilon\}$ et on obtient que

- Soit u est une valeur, mais alors on a toujours une réduction associée (*tval*) si u est une valeur régulière, (*catch*) ou (*trai*) si c'est un raise .
- Soit il existe u' tel que $u \rightarrow u'$. Mais alors $\text{try } u \text{ with } \varepsilon \mapsto v \rightarrow \text{try } u' \text{ with } \varepsilon \mapsto v$ par (*hred_{try}*).

(*t-subtyp*) On utilise alors l'hypothèse d'induction qui nous permet de conclure directement.

Notons que la réduction de tête faible étant incluse dans la réduction, ce théorème implique que si $\vdash t : A$, alors soit t est une valeur, soit il existe t' tel que $t \rightarrow t'$.

Lemme 4.48 Si n est une valeur et Δ et Δ' sont des ensembles d'exceptions tels que $\vdash n : \text{nat}^\Delta \uplus \Delta'$, alors

$$n = 0 \quad \text{ou} \quad n = S t \text{ (pour un terme } t \text{ quelconque)} \quad \text{ou} \quad n = \text{raise } \varepsilon \text{ (pour } \varepsilon \text{ quelconque)}$$

Preuve :

On montre que n ne peut pas être une valeur différente de 0, $S x$ ou $\text{raise } \varepsilon$. En effet, une tel valeur serait $\lambda x. n'$, S , rec , $\text{rec } z$ ou $\text{rec } z f$. Il est facile de vérifier que dans tous ces cas on aura forcément des types A et B tels que $A \rightarrow B \leq \text{nat}^\Delta \uplus \Delta'$. Le lemme 4.31 nous dit alors que l'on aurait $A \rightarrow B \approx \text{nat}^{\Delta_0} \uplus \Delta'_0$, ce qui est impossible.

Notons que dans l'énoncé du lemme précédent, si $n = \text{raise } \varepsilon$, nous ne supposons rien de particulier sur ε car nous n'en avons pas besoin (le but de ce lemme étant seulement de montrer qu'une valeur de type $\text{nat}^\Delta \uplus \Delta'$ est nécessairement un terme pour lequel le récursur se réduit). En pratique, on aura toujours pour ce cas $\varepsilon \in (\Delta \cup \Delta')$.

4.4.5 Corruption des termes

On veut prouver que, dans λ_ε , on peut toujours remplacer n'importe quel sous-terme d'un terme par une exception. Autrement dit, on peut toujours corrompre un terme, ce qui a pour effet de corrompre son type.

Pour cela, on introduit la relation de corruption $\sqsubseteq_\Delta$ entre termes.

Définition 4.49

Si t et u sont deux termes, la relation $t \sqsubseteq_\Delta u$ qui se lit « t est davantage corrompu par les exceptions de Δ que u » est définie comme la plus petite relation dérivable avec les règles d'inférence de la figure 4.7.

$$\begin{array}{c}
 \frac{}{t \sqsubseteq_\Delta t} \text{ (c-id)} \quad \frac{\varepsilon \in \Delta}{t \sqsubseteq_\Delta \text{raise } \varepsilon} \text{ (c-rai)} \quad \frac{t \sqsubseteq_\Delta t'}{\lambda x. t \sqsubseteq_\Delta \lambda x. t'} \text{ (c-lam)} \\
 \\
 \frac{t \sqsubseteq_\Delta t' \quad u \sqsubseteq_\Delta u'}{t u \sqsubseteq_\Delta t' u'} \text{ (c-app)} \\
 \\
 \frac{t \sqsubseteq_\Delta t' \quad u \sqsubseteq_\Delta u'}{\text{try } t \text{ with } \varepsilon \mapsto u \sqsubseteq_\Delta \text{try } t' \text{ with } \varepsilon \mapsto u'} \text{ (c-try)}
 \end{array}$$

FIG. 4.7 – La relation de corruption

Si t est un terme, l'ensemble des termes u tel que $t \sqsubseteq_\Delta u$ est l'ensemble des termes obtenus en remplaçant certains sous-termes dans t .

On montre alors le théorème, dit de corruption, qui exprime le fait que si on sait typer un terme (avec le type A), toutes ses corruptions sont également typables, quitte à changer le type par A^Δ . Ce théorème montre donc formellement que notre calcul respecte le premier principe que nous avons énoncé dans la section 4.1.1.

Théorème 4.50 (Corruption) Si t et u sont deux termes, A un type et Δ un ensemble de noms d'exceptions tel que $\Gamma \vdash t : A$ et $t \sqsubseteq_\Delta u$, alors

$$\Gamma \vdash u : A^\Delta$$

Preuve :

On montre cette propriété par induction sur la dérivation du jugement $t \sqsubseteq_\Delta u$:

(c-id) C'est direct puisque quelque soit Δ , $A \leq A^\Delta$.

(c-rai) On veut montrer que $\Gamma \vdash \text{raise } \varepsilon : A^\Delta$. Or, on a $\Gamma \vdash \text{raise } \varepsilon : A^{\{\varepsilon\}}$ et puisque $\varepsilon \in \Delta$, $A^{\{\varepsilon\}} \leq A^\Delta$. Donc finalement, on a bien que $\Gamma \vdash \text{raise } \varepsilon : A^\Delta$.

(*c-lam*) On sait que $\Gamma \vdash \lambda x. t : A$ et $t \sqsubseteq_{\Delta} t'$. En utilisant le lemme d'inversion de l'abstraction (lemme 4.37), il existe B et C tel que $B \rightarrow C \leq A$ et $\Gamma, x : B \vdash t : C$. Donc, par hypothèse d'induction, $\Gamma, x : B \vdash t' : C^{\Delta}$ et alors $\Gamma \vdash \lambda x. t' : B \rightarrow C^{\Delta}$. Mais comme

$$B \rightarrow C^{\Delta} \leq (B \rightarrow C^{\Delta})^{\Delta} \doteq B^{\Delta} \rightarrow C^{\Delta} \doteq (B \rightarrow C)^{\Delta},$$

on conclut bien finalement que $\Gamma \vdash \lambda x. t' : A^{\Delta}$ (en utilisant le lemme 4.10 avec le jugement $B \rightarrow C \leq A$).

(*c-app*) On sait que $\Gamma \vdash t u : A$, donc avec le lemme d'inversion de l'application (lemme 4.38), il existe un type B tel que $\Gamma \vdash t : B \rightarrow A$ et $\Gamma \vdash u : B$. Donc par hypothèse d'induction, $\Gamma \vdash t' : (B \rightarrow A)^{\Delta}$ et $\Gamma \vdash u' : B^{\Delta}$, d'où avec (*c-app*), $\Gamma \vdash t' u' : A^{\Delta}$.

(*c-try*) On sait que $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$, donc avec le lemme d'inversion du *try* (lemme 4.39), on a $\Gamma \vdash t : A \uplus \{\varepsilon\}$ et $\Gamma \vdash u : A$. Par hypothèse d'induction, $\Gamma \vdash t' : (A \uplus \{\varepsilon\})^{\Delta}$ et $\Gamma \vdash u' : A^{\Delta}$. Donc, avec (*c-try*), $\Gamma \vdash \text{try } t' \text{ with } \varepsilon \mapsto u' : A^{\Delta}$.

4.4.6 Sûreté vis-à-vis des exceptions

On montre que le typage des exceptions dans λ_{ε} est sûr au sens où, si le type d'un terme ne mentionne pas une certaine exception, ce terme ne lèvera pas l'exception en question lors de son exécution.

Définition 4.51

A chaque type A , on associe l'ensemble $\text{Exn}(A)$ des exceptions mentionnées par le type A défini par :

$$\begin{aligned} \text{Exn}(\text{nat}) &= \emptyset \\ \text{Exn}(A \rightarrow B) &= \text{Exn}(A) \cup \text{Exn}(B) \\ \text{Exn}(A \uplus \Delta) &= \text{Exn}(A) \cup \Delta \\ \text{Exn}(A^{\Delta}) &= \text{Exn}(A) \cup \Delta \end{aligned}$$

Théorème 4.52 (Sûreté du typage vis-à-vis des exceptions) Si t est un terme, A un type, Γ un environnement de typage et ε un nom d'exceptions tels que $\varepsilon \notin \text{Exn}(A)$ et $\Gamma \vdash t : A$, alors

$$t \not\rightarrow^* \text{raise } \varepsilon$$

Preuve :

Supposons que $t \rightarrow^* \text{raise } \varepsilon$ et montrons que cette hypothèse est absurde. Si $t \rightarrow^* \text{raise } \varepsilon$, puisque $\Gamma \vdash t : A$, le théorème de préservation du typage par réduction (théorème 4.45) nous dit que $\Gamma \vdash \text{raise } \varepsilon : A$. Donc, en utilisant le lemme d'inversion du *raise* (lemme 4.40), il existe un type B tel que $B \uplus \{\varepsilon\} \leq A$. Mais $\varepsilon \in \text{Exn}(B \uplus \{\varepsilon\})$ et il est facile de vérifier que si $C \leq D$ alors $\text{Exn}(C) \leq \text{Exn}(D)$ (par une simple induction sur la dérivation d'une telle relation) et donc $\varepsilon \in \text{Exn}(A)$, ce qui est incompatible avec les hypothèses du théorème.

De plus, pour les entiers naturels, on montre un résultat proche du précédent qui est qu'un programme de type *nat* qui termine se réduit toujours sur un entier naturel (et donc en particulier ne lèvera pas d'exceptions) :

Lemme 4.53 *Si t est un terme tel que $\vdash t : \text{nat}$ et t est faiblement normalisant, alors il existe un entier $n \geq 0$ tel que*

$$t \rightarrow^* S^n 0$$

Preuve :

Puisque t est faiblement normalisant, il existe t' tel que $t \rightarrow^* t'$ et t' est irréductible. Puisque $\vdash t : \text{nat}$, le théorème de préservation du typage par réduction nous dit que $\vdash t' : \text{nat}$. On peut alors montrer que si t' est irréductible et $\vdash t' : \text{nat}$, alors $t' = S^n 0$ (la preuve est une adaptation simple de celle du lemme 4.48).

On montrera au chapitre 5 que la condition de normalisation du lemme précédent est toujours satisfaite.

5

Sémantique

DANS ce chapitre nous allons présenter la sémantique du système de types de λ_ε . En particulier, nous commencerons par montrer que la corruption modifie le sens du type flèche au sens où celui-ci ne peut plus être interprété comme l'est celui du λ -calcul simplement typé (section 5.1). Puis nous présenterons un modèle de réalisabilité du calcul λ_ε qui nous permettra en particulier de montrer que les programmes se réduisent toujours sur une valeur (section 5.2).

5.1 Corruption et flèche

En réalisabilité, on interprète la flèche du λ -calcul simplement typé par l'opération qui à deux ensembles de termes A et B associe l'ensemble $A \multimap B$ défini par :

$$A \multimap B = \{ t \mid \forall u \in A, t u \in B \}$$

Cette interprétation a entre autre choses l'avantage de valider le sous-typage des flèches, c'est à dire leur covariance à droite et contravariance à gauche.

Dans λ_ε , la flèche est également covariante à droite et contravariante à gauche. Mais en plus, quels que soient les types A et B et l'ensemble de noms d'exceptions Δ , on a le sous-typage suivant :

$$A \rightarrow B \leq A^\Delta \rightarrow B^\Delta$$

Ce sous-typage nous interdit d'utiliser l'interprétation habituelle de la réalisabilité $\multimap$ pour la flèche de λ_ε . Pour s'en convaincre, considérons le terme t suivant

$$t \equiv \lambda x. \text{try } x \text{ with } \varepsilon \mapsto (\lambda y. y).$$

et supposons que l'on conserve l'interprétation usuelle des entiers naturels (ce qui sera le cas dans le modèle de la section 5.2), c'est à dire qu'on les interprète par l'ensemble de termes nat défini par :

$$\text{nat} = \{ t \mid t \rightarrow^* S^n 0 \}$$

Alors la fonction t réalise l'ensemble $\text{nat} \multimap \text{nat}$. Mais il est en revanche facile de vérifier que t ne réalise pas l'ensemble $\text{nat}^{\{\varepsilon\}} \multimap \text{nat}^{\{\varepsilon\}}$. En effet, en reprenant l'intuition de la

corruption, le terme $\text{raise } \varepsilon$ doit nécessairement appartenir à l'interprétation de $\text{nat}^{\{\varepsilon\}}$, mais le terme t ($\text{raise } \varepsilon$) se réduit sur $\lambda y. y$ qui n'appartient pas à l'interprétation de $\text{nat}^{\{\varepsilon\}}$.

Autrement dit, l'interprétation $\rightarrow$ pour la flèche ne valide pas le sous-typage $A \rightarrow B \leq A^\Delta \rightarrow B^\Delta$.

Pour valider le sous-typage $A \rightarrow B \leq A^\Delta \rightarrow B^\Delta$, on va interpréter la flèche de λ_ε de la façon suivante : si pour un type A donné, on note $\llbracket A \rrbracket$ son interprétation, on va prendre :

$$\llbracket A \rightarrow B \rrbracket = \bigcap_{\Delta \subseteq \mathcal{E}} \llbracket A^\Delta \rrbracket \rightarrow \llbracket B^\Delta \rrbracket$$

Autrement dit un terme t réalise $\llbracket A \rightarrow B \rrbracket$ si et seulement si pour tout ensemble d'exceptions Δ (éventuellement vide) et pour tout terme u réalisant $\llbracket A^\Delta \rrbracket$, alors le terme $t u$ réalise $\llbracket B^\Delta \rrbracket$. Remarquons que cette interprétation valide de façon précise le lemme 4.36 de la preuve de préservation du typage par réduction qui montre que dans λ_ε , si

$$\Gamma, x : A \vdash t : B$$

alors quel que soit Δ ,

$$\Gamma, x : A^\Delta \vdash t : B^\Delta.$$

La flèche de λ_ε est donc incluse dans la flèche usuelle de la réalisabilité, elle est plus précise que cette dernière.

Nous concluons cette section en remarquant une analogie¹ entre le comportement de la corruption vis-à-vis de la flèche de λ_ε et la règle *higher-order frame* [BY05] que l'on trouve en logique de séparation [Rey02]. Une même idée semble partagée puisque les règles de la corruption permettent d'utiliser une fonction avec une exception qu'elle ne connaît pas (son comportement est alors neutre vis-à-vis de cette exception), alors que la règle *higher-order frame* permet d'utiliser une fonction en présence de mémoire qu'elle n'utilise pas (elle est alors neutre vis-à-vis de cet espace mémoire, n'y accédant pas). Nous n'avons pour l'instant pas poussé plus loin cette analogie.

5.2 Un premier modèle de réalisabilité

Nous allons présenter un modèle de réalisabilité de λ_ε . Autrement dit, un modèle où les termes seront interprétés par eux-mêmes et les types seront interprétés par des ensembles de termes. Après avoir défini cette interprétation des types, nous en montrerons les propriétés, dont la correction vis-à-vis du typage de λ_ε et en déduirons un théorème de faible normalisation en forme normale de tête faible des termes du calcul.

5.2.1 Définition du modèle

Principe de l'interprétation

On associe à chaque type A de λ_ε un ensemble de termes noté $\llbracket A \rrbracket$.

¹Je remercie François Pottier et Didier Rémy pour m'avoir indiqué cette analogie.

Les flèches. Le type des flèches va être interprété par (voir section 5.1) :

$$\llbracket A \rightarrow B \rrbracket = \bigcap_{\Delta \subseteq \mathcal{E}} \{ t \mid \forall u \in \llbracket A \rrbracket^\Delta, t u \in \llbracket B \rrbracket^\Delta \}$$

L'union. Selon la définition que nous en avons donnée en section 4.1.2, un terme est de type $A \uplus \Delta$ s'il est de type A ou s'il se réduit sur `raise` ε pour $\varepsilon \in \Delta$. On interprétera l'union exactement selon cette définition.

La corruption. L'interprétation de la corruption, du type A^Δ , est plus problématique. Les termes de type A^Δ sont les termes de type A où on a éventuellement remplacé un nombre quelconque de sous-termes par `raise` ε pour $\varepsilon \in \Delta$. Pour interpréter directement la corruption, on doit être capable de définir l'ensemble des termes constitué de toute les corruptions possibles des termes d'un ensemble initial. Une telle opération serait probablement fastidieuse à manipuler et on lui préfère l'approche suivante : on paramètre l'interprétation des types par un ensemble de noms d'exceptions. Ainsi, au lieu de définir directement l'interprétation d'un type, on définit l'interprétation de ce type corrompu par l'ensemble d'exceptions en paramètre. Ainsi, l'interprétation du type A^Δ se contente de pousser l'ensemble Δ en paramètre de l'interprétation de A . Autrement dit, on repousse l'interprétation réelle de la corruption au niveau du type de base, les entiers naturels.

Définition de l'interprétation

Définition 5.1 (Interprétation des types de λ_ε)

À chaque type A de λ_ε et pour tout ensemble de noms d'exceptions Δ , on associe l'ensemble de termes clos $\llbracket A \rrbracket^\Delta$ nommé interprétation de A . Cet ensemble est défini inductivement par :

$$\begin{aligned} \llbracket \text{nat} \rrbracket^\Delta &= \{ t \mid t \rightarrow^* S^n 0, n \in \mathbb{N} \} \\ &\quad \cup \{ t \mid t \rightarrow^* S^n \text{raise } \varepsilon, n \in \mathbb{N}, \varepsilon \in \Delta \} \\ \llbracket A \rightarrow B \rrbracket^\Delta &= \bigcap_{\Delta \subseteq \Delta'} \{ t \mid \forall u \in \llbracket A \rrbracket^{\Delta'}, t u \in \llbracket B \rrbracket^{\Delta'} \} \\ \llbracket A^{\Delta'} \rrbracket^\Delta &= \llbracket A \rrbracket^{\Delta \cup \Delta'} \\ \llbracket A \uplus \Delta' \rrbracket^\Delta &= \llbracket A \rrbracket^\Delta \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta' \} \end{aligned}$$

L'idée de paramétrer l'interprétation avec un ensemble de noms d'exceptions peut être vu de la façon suivante : l'interprétation du type A commence par calculer sa forme normale par la simplification de type (définition 4.15) que l'on a défini pour la preuve de préservation du typage par réduction. Cette simplification pousse la corruption vers les feuilles du type. Le type obtenu est sémantiquement équivalent au type initial (un type et son simplifié sont égaux par sous-typage) mais ne mentionne la corruption que pour le type des entiers. C'est alors cette forme simplifiée qui est interprétée.

Notation

Si A est un type, on nomme *interprétation simple* de A et on note $\llbracket A \rrbracket$ l'ensemble $\llbracket A \rrbracket^\emptyset$.

On va montrer que l'interprétation que nous venons de définir valide le typage de λ_ε . On commence par donner figure 5.1 les dépendances des différents lemmes qui vont nous permettre d'arriver à ce résultat.

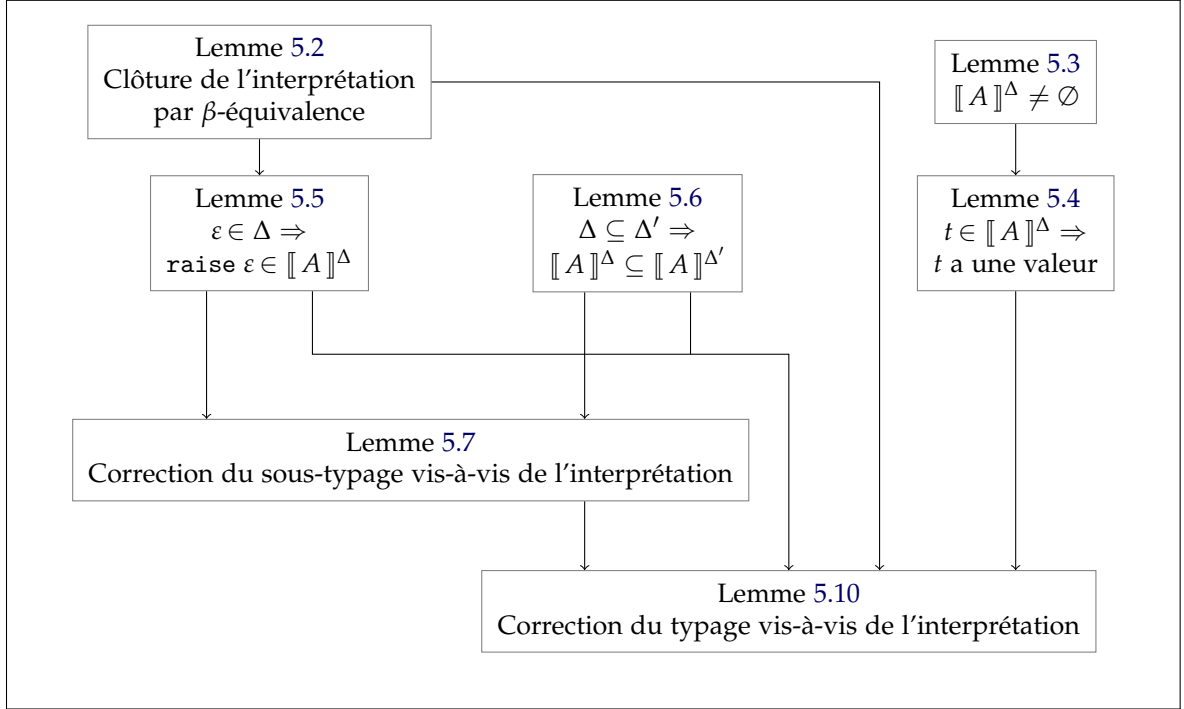


FIG. 5.1 – Schéma de dépendance pour la preuve de correction du typage vis-à-vis de l'interprétation

5.2.2 Propriétés de l'interprétation

L'interprétation que nous venons de définir est close par β -équivalence :

Lemme 5.2 (Clôture par β -équivalence) Si t et u sont deux termes, A un type et Δ un ensemble de noms d'exceptions tel que $t \in \llbracket A \rrbracket^\Delta$ et $t =_\beta u$, alors

$$u \in \llbracket A \rrbracket^\Delta$$

Preuve :

On prouve ce lemme par induction sur la structure du type A . Le cas de nat est une conséquence de la confluence de λ_ε (théorème 3.7). Le cas de $B^{\Delta'}$ est une simple application de l'hypothèse d'induction et le cas de $B \uplus \Delta'$ est une conséquence de la confluence et de l'hypothèse d'induction. Pour le cas $B \rightarrow C$, on doit montrer que pour tout ensemble de noms d'exceptions $\Delta' \supseteq \Delta$, si $u' \in \llbracket B \rrbracket^{\Delta'}$, alors $u u' \in \llbracket C \rrbracket^{\Delta'}$. Mais puisque $t =_\beta u$, on a $t u' =_\beta u u'$. Comme $t \in \llbracket B \rightarrow C \rrbracket^\Delta$, par définition $t u' \in \llbracket C \rrbracket^{\Delta'}$. Alors, par hypothèse d'induction, $\llbracket C \rrbracket^{\Delta'}$ est clos par β -équivalence et on a $u u' \in \llbracket C \rrbracket^{\Delta'}$.

En particulier, l'interprétation est donc close par réduction et par anti-réduction.

De plus, l'interprétation d'un type est toujours habitée :

Lemme 5.3 *Si A est un type et Δ un ensemble de noms d'exceptions, alors*

$$\llbracket A \rrbracket^\Delta \neq \emptyset$$

Preuve :

On raisonne par induction sur la structure du terme A :

- Si A est nat , alors $\llbracket A \rrbracket^\Delta$ contient au moins 0.
- Si A est $B \rightarrow C$, on doit montrer que quel que soit $\Delta' \supseteq \Delta$, il existe un terme t tel que quel que soit $u \in \llbracket A \rrbracket^{\Delta'}$, $t u \in \llbracket B \rrbracket^{\Delta'}$. Soit $\Delta' \supseteq \Delta$, par hypothèse d'induction $\llbracket B \rrbracket^{\Delta'} \neq \emptyset$ et donc il existe $b \in \llbracket B \rrbracket^{\Delta'}$. Mais alors quel que soit $u \in \llbracket A \rrbracket^{\Delta'}$, $(\lambda x. b) u \rightarrow b\{x := u\}$. Mais b étant clos, x n'est pas une variable libre de b (à α -renommage près) et donc $b\{x := u\} = b \in \llbracket B \rrbracket^{\Delta'}$.
- Si A est $B \uplus \Delta'$, alors par hypothèse d'induction, $\llbracket B \rrbracket^\Delta$ n'est pas vide et donc $\llbracket B \uplus \Delta' \rrbracket^\Delta$ ne l'est pas non plus.
- Si A est $B^{\Delta'}$, alors on conclut directement par hypothèse d'induction.

On montre également que les types sont interprétés uniquement par des termes se réduisant sur une valeur :

Lemme 5.4 *Si t est un terme, A un type et Δ un ensemble de noms d'exceptions tel que $t \in \llbracket A \rrbracket^\Delta$, alors t a une valeur.*

Preuve :

On raisonne par induction sur la définition de $\llbracket A \rrbracket^\Delta$. Le cas de nat est trivial et les cas de $A \uplus \Delta'$ et $A^{\Delta'}$ sont de simples applications de l'hypothèse d'induction. Pour le cas de la flèche, supposons que $t \in \llbracket B \rightarrow C \rrbracket^\Delta$. Alors par définition, pour tout ensemble de noms d'exceptions $\Delta' \supseteq \Delta$ et pour tout $u \in \llbracket B \rrbracket^{\Delta'}$, $t u \in \llbracket C \rrbracket^{\Delta'}$. Mais par ailleurs, on sait que $\llbracket B \rrbracket^{\Delta'}$ n'est pas vide (lemme 5.3). Donc il existe un terme $u \in \llbracket B \rrbracket^{\Delta'}$ et alors $t u \in \llbracket C \rrbracket^{\Delta'}$. L'hypothèse d'induction nous dit alors que $t u$ a une valeur. On conclut avec le lemme 3.22.

Le terme $\text{raise } \varepsilon$ habite l'interprétation de n'importe quel type pour peu que l'ensemble d'exceptions paramétrant l'interprétation en question est non vide :

Lemme 5.5 *Si A est un type, Δ un ensemble de noms d'exceptions et ε un nom d'exception tels que $\varepsilon \in \Delta$, alors*

$$\{ t \mid t \rightarrow^* \text{raise } \varepsilon \} \subseteq \llbracket A \rrbracket^\Delta$$

Preuve :

On raisonne par induction sur la définition de $\llbracket A \rrbracket^\Delta$ pour montrer que $\text{raise } \varepsilon \in \llbracket A \rrbracket^\Delta$, puis on conclut par clôture de l'interprétation par anti-réduction (lemme 5.2). Les cas de nat et $A \uplus \Delta$ sont

des conséquences directes de leur définition. Le cas A^Δ est une application directe de l'hypothèse d'induction et pour le cas $A \rightarrow B$, si $u \in \llbracket A \rrbracket^{\Delta'}$, alors $(\text{raise } \varepsilon) u \rightarrow \text{raise } \varepsilon$ et on conclut donc par hypothèse d'induction et en utilisant la clôture par anti-réduction de $\llbracket B \rrbracket^{\Delta'}$ (lemme 5.2).

Enfin, on montre le lemme suivant :

Lemme 5.6 *Si A est un type et Δ et Δ' des ensembles de noms d'exceptions tels que $\Delta \subseteq \Delta'$, alors*

$$\llbracket A \rrbracket^\Delta \subseteq \llbracket A \rrbracket^{\Delta'}$$

Preuve :

On procède par induction sur la structure du type A . Le cas de nat est trivial. Les cas de $A \uplus \Delta'$ et $A^{\Delta'}$ sont de simples applications de l'hypothèse d'induction. Pour le cas de $B \rightarrow C$, on a

$$\begin{aligned} \llbracket B \rightarrow C \rrbracket^\Delta &= \bigcap_{\Delta \subseteq \Delta_0} \{ t \mid \forall u \in \llbracket B \rrbracket^{\Delta_0}, t u \in \llbracket C \rrbracket^{\Delta_0} \} \\ \llbracket B \rightarrow C \rrbracket^{\Delta'} &= \bigcap_{\Delta' \subseteq \Delta'_0} \{ t \mid \forall u \in \llbracket B \rrbracket^{\Delta'_0}, t u \in \llbracket C \rrbracket^{\Delta'_0} \} \end{aligned}$$

Supposons $t \in \llbracket B \rightarrow C \rrbracket^\Delta$, on veut montrer que $t \in \llbracket B \rightarrow C \rrbracket^{\Delta'}$. Mais si $\Delta'_0 \supseteq \Delta'$, alors puisque $\Delta' \supseteq \Delta$, on a $\Delta'_0 \supseteq \Delta$. Alors, on a bien que $\forall u \in \llbracket B \rrbracket^{\Delta'_0}, t u \in \llbracket C \rrbracket^{\Delta'_0}$.

Un corollaire immédiat est que le modèle valide la règle $A \leq A^\Delta$:

Corollaire *Si A est un type et Δ un ensemble de noms d'exceptions, alors*

$$\llbracket A \rrbracket \subseteq \llbracket A \rrbracket^\Delta$$

5.2.3 Correction

On commence par montrer que l'interprétation valide les règles du sous-typage de λ_ε :

Lemme 5.7 (Correction des règles de sous-typage vis-à-vis de l'interprétation) *Si A et B sont deux types tel que $A \leq B$, alors pour tout ensemble de noms d'exceptions Δ ,*

$$\llbracket A \rrbracket^\Delta \subseteq \llbracket B \rrbracket^\Delta$$

Preuve :

On procède par induction sur la dérivation de $A \leq B$:

(*st-id*) On conclut directement par réflexivité de l'inclusion d'ensemble.

(*st-trans*) On conclut directement par transitivité de l'inclusion d'ensemble.

- (*st-arrow*) On sait par hypothèse d'induction que quel que soit Δ , $\llbracket A' \rrbracket^\Delta \subseteq \llbracket A \rrbracket^\Delta$ et $\llbracket B \rrbracket^\Delta \subseteq \llbracket B' \rrbracket^\Delta$ et on veut montrer que quel que soit Δ , $\llbracket A \rightarrow B \rrbracket^\Delta \subseteq \llbracket A' \rightarrow B' \rrbracket^\Delta$. Soit Δ un ensemble de nom d'exceptions, soit $t \in \llbracket A \rightarrow B \rrbracket^\Delta$, soit $\Delta' \supseteq \Delta$ et soit $u \in \llbracket A' \rrbracket^{\Delta'}$. L'hypothèse d'induction nous dit que $u \in \llbracket A \rrbracket^{\Delta'}$ et, puisque $\Delta' \supseteq \Delta$, par définition $t u \in \llbracket B \rrbracket^{\Delta'}$. Alors par hypothèse d'induction $t u \in \llbracket B' \rrbracket^{\Delta'}$ et on a donc bien $t \in \llbracket A' \rightarrow B' \rrbracket^\Delta$.
- (*st-uni-arrow*) Soit $t \in \llbracket (A \rightarrow B) \uplus \Delta' \rrbracket^\Delta$, on doit montrer que $t \in \llbracket A \rightarrow (B \uplus \Delta') \rrbracket^\Delta$. Par définition, ou bien $t \rightarrow^* \text{raise } \varepsilon$ pour $\varepsilon \in \Delta'$, ou bien $t \in \llbracket (A \rightarrow B) \rrbracket^\Delta$. Soit $\Delta'' \supseteq \Delta$ et soit $u \in \llbracket A \rrbracket^{\Delta'}$.
- Si $t \rightarrow^* \text{raise } \varepsilon$, alors $t u \rightarrow^* \text{raise } \varepsilon$ et par définition on a bien $t u \in \llbracket B \uplus \Delta' \rrbracket^{\Delta''}$ (puisque $\varepsilon \in \Delta'$).
 - Si $t \in \llbracket (A \rightarrow B) \rrbracket^\Delta$, alors comme $\Delta'' \supseteq \Delta$, $t u \in \llbracket B \rrbracket^{\Delta''}$ et donc par définition on a bien $t u \in \llbracket B \uplus \Delta' \rrbracket^{\Delta''}$.
- (*st-uni-ctx*) L'union de λ_ε étant interprétée comme une union ensembliste, il est trivial que si $\llbracket A \rrbracket^\Delta \subseteq \llbracket B \rrbracket^\Delta$ alors $\llbracket A \uplus \Delta' \rrbracket^\Delta \subseteq \llbracket B \uplus \Delta' \rrbracket^\Delta$.
- (*st-cor*) On doit montrer que $\llbracket A \uplus \Delta' \rrbracket^\Delta \subseteq \llbracket A^{\Delta'} \rrbracket^\Delta = \llbracket A \rrbracket^{\Delta \cup \Delta'}$. Si $t \in \llbracket A \uplus \Delta' \rrbracket^\Delta$, alors :
- Soit $t \rightarrow^* \text{raise } \varepsilon$ où $\varepsilon \in \Delta'$ et alors on conclut avec le lemme 5.5.
 - Soit $t \in \llbracket A \rrbracket^\Delta$ et on conclut avec le lemme 5.6.
- (*st-cor-arrow*) On doit montrer que $\llbracket (A \rightarrow B)^{\Delta'} \rrbracket^\Delta = \llbracket A^{\Delta'} \rightarrow B^{\Delta'} \rrbracket^\Delta$. On va montrer les deux inclusions séparément :
- Si $t \in \llbracket (A \rightarrow B)^{\Delta'} \rrbracket^\Delta = \llbracket A \rightarrow B \rrbracket^{\Delta \cup \Delta'}$, alors soit $\Delta_1 \supseteq \Delta$ et soit $u \in \llbracket A^{\Delta'} \rrbracket^{\Delta_1} = \llbracket A \rrbracket^{\Delta_1 \cup \Delta'}$. Puisque $\Delta_1 \supseteq \Delta$, $\Delta_1 \cup \Delta' \supseteq \Delta \cup \Delta'$ et donc on a bien $t u \in \llbracket B \rrbracket^{\Delta_1 \cup \Delta'}$.
 - Si $t \in \llbracket A^{\Delta'} \rightarrow B^{\Delta'} \rrbracket^\Delta$, alors soit $\Delta_2 \supseteq \Delta \cup \Delta'$ et soit $u \in \llbracket A \rrbracket^{\Delta_2}$. En particulier on a $\Delta_2 \supseteq \Delta$ et $\Delta_2 \cup \Delta' = \Delta_2$ et donc $t u \in \llbracket B \rrbracket^{\Delta_2}$.

Tous les autres cas sont des conséquences directes des définitions.

L'interprétation des types s'étend naturellement aux environnements de typage (qui sont de simples types enregistrements) en posant :

Définition 5.8 (Interprétation des environnement de typage)

Si Γ est un environnement de typage et Δ un ensemble de noms d'exceptions,

$$\llbracket \Gamma \rrbracket^\Delta = \{ \sigma \mid \forall (x : A) \in \Gamma, \sigma(x) \in \llbracket A \rrbracket^\Delta \}.$$

Dans la suite, on notera plus simplement $\llbracket \Gamma \rrbracket$ l'ensemble $\llbracket \Gamma \rrbracket^\emptyset$. Remarquons que si $\Delta \subseteq \Delta'$, puisque $\llbracket A \rrbracket^\Delta \subseteq \llbracket A \rrbracket^{\Delta'}$ (lemme 5.6), il est facile de vérifier que $\llbracket \Gamma \rrbracket^\Delta \subseteq \llbracket \Gamma \rrbracket^{\Delta'}$ et donc en particulier que $\llbracket \Gamma \rrbracket \subseteq \llbracket \Gamma \rrbracket^\Delta$.

Une substitution de variables de termes est une fonction partielle des variables de termes dans les termes. Si σ est une substitution, on définit l'ensemble des variables libres d'une substitution $\text{FV}(\sigma)$ par :

$$\text{FV}(\sigma) = \bigcup_{x \in \text{dom}(\sigma)} \text{FV}(\sigma(x))$$

Définition 5.9 (Substitution parallèle)

Si σ est une substitution de variables de termes et t est un terme, on définit la substitution parallèle $t[\sigma]$ de t par σ , opération définie inductivement par :

$$\begin{aligned}
x[\sigma] &= \sigma(x) \\
(\lambda x. t)[\sigma] &= \lambda x. t[\sigma] \quad (\text{si } x \notin \text{FV}(\sigma) \text{ et } x \notin \text{dom}(\sigma)) \\
(t u)[\sigma] &= t[\sigma] u[\sigma] \\
(\text{raise } \varepsilon)[\sigma] &= \text{raise } \varepsilon \\
(\text{try } t \text{ with } \varepsilon \mapsto u)[\sigma] &= \text{try } t[\sigma] \text{ with } \varepsilon \mapsto u[\sigma] \\
0[\sigma] &= 0 \\
S[\sigma] &= S \\
\text{rec}[\sigma] &= \text{rec}
\end{aligned}$$

Remarquons que à α -renommage près pour le cas de l'abstraction, la substitution parallèle est toujours définie.

On peut alors montrer que l'interprétation des types valide les règles de typage :

Théorème 5.10 (Correction du typage vis-à-vis du modèle) Si t est un terme, A un type et Γ un environnement de typage tel que $\Gamma \vdash t : A$, alors pour tout ensemble de noms d'exceptions Δ et toute substitution $\sigma \in \llbracket \Gamma \rrbracket^\Delta$, on a

$$t[\sigma] \in \llbracket A \rrbracket^\Delta$$

Preuve :

On raisonne par induction sur la dérivation de $\Gamma \vdash t : A$:

(t-ax) Si $\sigma \in \llbracket \Gamma \rrbracket^\Delta$ et $(x : A) \in \Gamma$, alors par définition $t[\sigma] \in \llbracket A \rrbracket^\Delta$.

(t-abs) On a $\Gamma \vdash \lambda x. t : A \rightarrow B$ car $\Gamma, x : A \vdash t : B$. Soit une substitution $\sigma \in \llbracket \Gamma \rrbracket^\Delta$, un ensemble de noms d'exception $\Delta' \supseteq \Delta$ et un terme $u \in \llbracket A \rrbracket^{\Delta'}$. À α -renommage près, on peut toujours considérer que $x \notin \text{FV}(\sigma)$ et $x \notin \text{dom}(\sigma)$. Considérons la substitution $\gamma = \sigma + \{x \mapsto u\}$, on a $\gamma \in \llbracket \Gamma, x : A \rrbracket^{\Delta'}$. Alors par hypothèse d'induction, $t[\gamma] \in \llbracket B \rrbracket^{\Delta'}$. Mais puisque $(\lambda x. t)[\sigma] u = (\lambda x. t[\sigma]) u \rightarrow t[\gamma]$ et comme l'ensemble $\llbracket B \rrbracket^{\Delta'}$ est clos par anti-réduction (lemme 5.2), on a $(\lambda x. t)[\sigma] \in \llbracket A \rightarrow B \rrbracket^\Delta$.

(t-app) On a que $\Gamma \vdash t u : B$ car $\Gamma \vdash t : A \rightarrow B$ et $\Gamma \vdash u : A$. Soit une substitution $\sigma \in \llbracket \Gamma \rrbracket^\Delta$, par hypothèse d'induction, $t[\sigma] \in \llbracket A \rightarrow B \rrbracket^\Delta$ et $u[\sigma] \in \llbracket A \rrbracket^\Delta$. Il s'ensuit par définition de $\llbracket A \rightarrow B \rrbracket^\Delta$ que $t[\sigma] u[\sigma] \equiv (t u)[\sigma] \in \llbracket B \rrbracket^\Delta$.

(t-zero) On doit montrer que $0 \in \llbracket \text{nat} \rrbracket^\Delta$, ce qui est direct par définition de $\llbracket \text{nat} \rrbracket^\Delta$.

(t-succ) On doit montrer que pour tout ensemble de noms d'exceptions $\Delta' \supseteq \Delta$ et tout terme $n \in \llbracket \text{nat} \rrbracket^{\Delta'}$, $S n \in \llbracket \text{nat} \rrbracket^{\Delta'}$. Ceci est direct par définition de $\llbracket \text{nat} \rrbracket^{\Delta'}$.

(t-rec) On doit montrer

$$\text{rec} \in \llbracket A \uplus \Delta' \rightarrow (\text{nat}^{\Delta'} \rightarrow A \uplus \Delta' \rightarrow A \uplus \Delta') \rightarrow \text{nat}^{\Delta'} \uplus \Delta'' \rightarrow A \uplus (\Delta' \cup \Delta'') \rrbracket^\Delta,$$

c'est à dire, par définition de l'interprétation des flèches, que quels que soient les ensembles de noms d'exceptions Δ_1, Δ_2 et Δ_3 tels que $\Delta \subseteq \Delta_1 \subseteq \Delta_2 \subseteq \Delta_3$, si

$$- z \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_1},$$

- $f \in \llbracket (\text{nat}^{\Delta'} \rightarrow A \uplus \Delta' \rightarrow A \uplus \Delta') \rrbracket^{\Delta_2}$,
- $n \in \llbracket \text{nat}^{\Delta'} \uplus \Delta'' \rrbracket^{\Delta_3} = \llbracket \text{nat} \rrbracket^{\Delta_3 \cup \Delta'} \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta'' \}$,
alors $\text{rec } z f n \in \llbracket A \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$. Si $n \in \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta'' \}$, alors on a $\text{rec } z f n \rightarrow \text{rec } z f \text{raise } \varepsilon \rightarrow \text{raise } \varepsilon$ et puisque $\text{raise } \varepsilon \in \llbracket A \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$ par définition et que l'interprétation est close par anti-réduction (lemme 5.2), on a bien $\text{rec } z f n \in \llbracket A \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$. Sinon, $n \in \llbracket \text{nat} \rrbracket^{\Delta_3 \cup \Delta'}$. En utilisant Θ pour représenter soit 0, soit $\text{raise } \varepsilon$ (où $\varepsilon \in \Delta_3 \cup \Delta'$), alors par définition $n \rightarrow^* S^k \Theta$ pour $k \in \mathbb{N}$. Donc $\text{rec } z f n \rightarrow^* \text{rec } z f (S^k \Theta)$ et on va prouver que $\text{rec } z f (S^k \Theta) \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_3}$ par induction sur k (ce qui nous permettra de conclure puisque $\llbracket A \uplus \Delta' \rrbracket^{\Delta_3} \subseteq \llbracket A \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$ par définition de ces interprétations) :
- Si $k = 0$, alors :
 - Soit $n \rightarrow^* 0$, alors $\text{rec } z f 0 \rightarrow z$ et puisque $z \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_1}$ et $\Delta_1 \subseteq \Delta_3$, on a $z \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_3}$ grâce au lemme 5.6. On conclut alors par clôture de l'interprétation par anti-réduction (lemme 5.2).
 - Soit $n \rightarrow^* \text{raise } \varepsilon$ où $\varepsilon \in \Delta_3 \cup \Delta'$, alors $\text{rec } z f \text{raise } \varepsilon \rightarrow \text{raise } \varepsilon$. Or si $\varepsilon \in \Delta_3$, $\text{raise } \varepsilon \in \llbracket A \rrbracket^{\Delta_3}$ (lemme 5.5) et donc par définition de l'interprétation de l'union, si $\varepsilon \in \Delta_3 \cup \Delta'$, alors $\text{raise } \varepsilon \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_3}$. On conclut alors par clôture de l'interprétation par anti-réduction (lemme 5.2).
- Si $k = k' + 1$, c'est donc que $n \rightarrow^* S (S^{k'} \Theta)$. On a alors la réduction $\text{rec } z f (S (S^{k'} \Theta)) \rightarrow f (S^{k'} \Theta) (\text{rec } z f (S^{k'} \Theta))$. Par hypothèse d'induction $\text{rec } z f (S^{k'} \Theta) \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_3}$ et par définition de l'interprétation, $S^{k'} \Theta \in \llbracket \text{nat}^{\Delta'} \rrbracket^{\Delta_3}$. Comme de plus $f \in \llbracket (\text{nat}^{\Delta'} \rightarrow A \uplus \Delta' \rightarrow A \uplus \Delta') \rrbracket^{\Delta_2}$ et $\Delta_2 \subseteq \Delta_3$, on a finalement $f (S^{k'} \Theta) (\text{rec } z f (S^{k'} \Theta)) \in \llbracket A \uplus \Delta' \rrbracket^{\Delta_3}$ et on conclut alors par clôture de l'interprétation par anti-réduction (lemme 5.2).

(t-raise) On doit montrer que $\text{raise } \varepsilon \in \llbracket A \uplus \{\varepsilon\} \rrbracket^{\Delta}$, ce qui est évident au vu de la définition de $\llbracket A \uplus \{\varepsilon\} \rrbracket^{\Delta}$.

(t-try) On a $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$ car $\Gamma \vdash t : A \uplus \{\varepsilon\}$ et $\Gamma \vdash u : A$. Soit une substitution $\sigma \in \llbracket \Gamma \rrbracket^{\Delta}$, par hypothèse d'induction $t[\sigma] \in \llbracket A \uplus \{\varepsilon\} \rrbracket^{\Delta}$ et $u[\sigma] \in \llbracket A \rrbracket^{\Delta}$. Donc par définition de $\llbracket A \uplus \{\varepsilon\} \rrbracket^{\Delta}$,

- Soit $t[\sigma] \rightarrow^* \text{raise } \varepsilon$, et alors $(\text{try } t \text{ with } \varepsilon \mapsto u)[\sigma] = \text{try } t[\sigma] \text{ with } \varepsilon \mapsto u[\sigma] \rightarrow^* u[\sigma]$ et puisque $u[\sigma] \in \llbracket A \rrbracket^{\Delta}$, on a $(\text{try } t \text{ with } \varepsilon \mapsto u)[\sigma] \in \llbracket A \rrbracket^{\Delta}$ (l'interprétation étant close par anti-réduction).
- Soit $t[\sigma] \in \llbracket A \rrbracket^{\Delta}$, mais alors le lemme 5.4 nous dit que $t[\sigma]$ a une valeur et donc on a $(\text{try } t \text{ with } \varepsilon \mapsto u)[\sigma] = \text{try } t[\sigma] \text{ with } \varepsilon \mapsto u[\sigma]$ qui se réduit soit sur $t[\sigma]$, soit sur $u[\sigma]$. Mais ces deux termes appartiennent à l'ensemble $\llbracket A \rrbracket^{\Delta}$ et on conclut par clôture par anti-réduction de l'interprétation.

(t-subtyp) Ce cas est une application directe du lemme de correction du sous-typage vis-à-vis de l'interprétation (lemme 5.7) et de l'hypothèse d'induction.

5.2.4 Propriétés dérivées

La principale propriété du calcul que le modèle nous permet de déduire est une propriété de normalisation faible. Les termes bien typés de λ_ε ont une valeur :

Théorème 5.11 (Réduction sur une valeur) *Si Γ est un environnement de typage, t un terme et A un type tel que $\Gamma \vdash t : A$, alors t a une valeur.*

Preuve :

Ce théorème est une conséquence direct du théorème de correction du typage vis-à-vis de l'interprétation (théorème 5.10) et du fait que les termes habitant l'interprétation des types ont une valeur (lemme 5.4).

Le modèle nous permet également de montrer qu'un programme de type nat se réduit toujours sur un entier naturel :

Lemme 5.12 *Si t est un terme tel que $\vdash t : \text{nat}$, alors il existe un entier $n \geq 0$ tel que*

$$t \rightarrow^* S^n 0$$

Preuve :

Si $\vdash t : \text{nat}$, alors par correction de l'interprétation vis-à-vis du typage (théorème 5.10), $t \in \llbracket \text{nat} \rrbracket$. Or par définition les habitants de l'ensemble $\llbracket \text{nat} \rrbracket$ sont exactement les termes se réduisant sur $S^n 0$.

Remarquons que le théorème de préservation du typage par réduction (théorème 4.45) nous a déjà permis d'établir un résultat similaire, le lemme 4.53. Une différence avec ce précédent résultat est que l'hypothèse de faible normalisation n'est ici pas nécessaire.

Par ailleurs, ce résultat est obtenu ici par des arguments uniquement sémantiques, alors que le lemme 4.53 l'était par des arguments syntaxiques. Ce résultat présente ainsi une assurance supplémentaire de la sûreté de la détection des exceptions effectuée par le système de type.

6

Extensions

DANS ce chapitre, nous allons présenter un certain nombre d’extensions au calcul λ_ε , extensions principalement destinées à rendre λ_ε plus utilisable en tant que langage de programmation. Nous commencerons par ajouter un point fixe au calcul (section 6.1). Nous ajouterons ensuite à λ_ε de nouveaux types de données primitifs (section 6.2). Les entiers naturels de λ_ε sont un type de données paresseux, mais nous montrerons en section 6.3 que des entiers naturels stricts peuvent être codés. Enfin, nous discuterons la possibilité d’étendre le mécanisme d’exceptions du calcul afin que les exceptions puissent transporter des arguments en section 6.4.

6.1 Point fixe et « match »

6.1.1 Présentation

Pour permettre les définitions par récurrence, nous avons présenté les entiers naturels de λ_ε munis d’un récursur proche de celui du système T de Gödel. Ce choix, par opposition à l’utilisation d’un simple point fixe, permet d’assurer à λ_ε des propriétés de normalisation. En particulier, le modèle de réalisabilité du chapitre 5 nous a permis de montrer que les termes de λ_ε se réduisent sur une valeur (théorème 5.11).

Nous allons montrer dans cette section que si on accepte la perte de ces propriétés, l’opérateur de récursion `rec` peut être remplacé par un opérateur de point fixe et un opérateur non récursif de « matching ». L’un des avantages d’une telle extension étant alors que la programmation de certaines fonctions y est plus simple (voir l’exemple de la division euclidienne, section 6.1.2). Dans la suite, nous nommerons cette extension $\lambda_\varepsilon^{\text{fix}}$.

Termes et réductions

Dans le calcul $\lambda_\varepsilon^{\text{fix}}$, l’opérateur de récursion `rec` est redondant (voir les exemples ci-dessous) et on va donc le supprimer. On ajoute à la syntaxe les deux nouvelles constructions `fix` et `match`. La syntaxe des termes de $\lambda_\varepsilon^{\text{fix}}$ est donc

$$t ::= x \mid \lambda x. t \mid t t \mid \text{raise } \varepsilon \mid \text{try } t \text{ with } \varepsilon \mapsto t \mid \text{fix} \mid 0 \mid S \mid \text{match}$$

Les valeurs régulières sont modifiées en conséquence. Une valeur régulière de $\lambda_\epsilon^{\text{fix}}$ est un terme dans une des formes suivantes :

$$v ::= \lambda x. t \mid 0 \mid S \mid S t \mid \text{fix} \mid \text{match} \mid \text{match } t \mid \text{match } t t$$

De plus, on ajoute aux réductions de λ_ϵ les règles suivantes :

$$\begin{array}{lll} \text{fix } f & \rightarrow & f (\text{fix } f) & (\text{fix}) \\ \text{match } z f 0 & \rightarrow & z & (\text{matchz}) \\ \text{match } z f (S n) & \rightarrow & f n & (\text{matchs}) \\ \text{match } z f (\text{raise } \epsilon) & \rightarrow & \text{raise } \epsilon & (\text{matchr}) \end{array}$$

Dans la suite nous utiliserons le sucre syntaxique suivant :

Notation

– On notera

$$\begin{array}{l} \text{match } n \text{ with} \\ | 0 \rightarrow z \\ | S m \rightarrow f \end{array}$$

pour le terme

$$\text{match } z (\lambda m. f) n$$

– On notera

$$\text{let rec } f a_1 \dots a_n = e \text{ in } t$$

pour le terme

$$(\lambda f. t) (\text{fix } (\lambda f. \lambda a_1. \dots \lambda a_n. e))$$

Typage et sous-typage

La syntaxe des types de λ_ϵ n'est pas modifiée et en particulier le sous-typage de $\lambda_\epsilon^{\text{fix}}$ est aussi conservé. Aux règles de typage, on ajoute les deux règles suivantes pour le typage des deux nouvelles constructions :

$$\frac{}{\Gamma \vdash \text{fix} : (A \rightarrow A) \rightarrow A} (t\text{-fix})$$

$$\frac{}{\Gamma \vdash \text{match} : A \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow A \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow A \uplus (\Delta \cup \Delta')} (t\text{-match})$$

Notons que la règle $(t\text{-fix})$ n'est pas compatible avec le modèle de réalisabilité du chapitre 5. En effet, on peut montrer que fix n'est pas dans l'interprétation du type $(A \rightarrow A) \rightarrow A$ puisque par exemple $\text{fix } (\lambda x. x) \notin \llbracket \text{nat} \rrbracket$ (ce terme se réduit indéfiniment sur lui-même). Les termes de $\lambda_\epsilon^{\text{fix}}$ ne vérifient en particulier aucune propriété de normalisation. On peut imaginer récupérer ces propriétés en introduisant un point fixe gardé [Gim94].

6.1.2 Exemples

Dans $\lambda_\epsilon^{\text{fix}}$, l'opérateur de récursion rec de λ_ϵ est codable par la fonction myrec suivante :

$$\begin{aligned} \text{myrec} & : B \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow B \uplus \Delta \rightarrow B \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow B \uplus (\Delta \cup \Delta') \\ & \equiv \lambda z. \lambda f. \text{let rec aux } n = \\ & \quad \text{match } n \text{ with} \\ & \quad | 0 \rightarrow z \\ & \quad | S x \rightarrow f x (\text{aux } x) \\ & \text{in aux} \end{aligned}$$

On peut vérifier que les réductions de myrec sont celles de l'opérateur rec de λ_ϵ . Dans le typage de ce terme, les types des variables sont :

$$\begin{aligned} z & : B \uplus \Delta \\ f & : \text{nat}^\Delta \rightarrow B \uplus \Delta \rightarrow B \uplus \Delta \\ \text{aux} & : \text{nat}^\Delta \uplus \Delta' \rightarrow B \uplus (\Delta \cup \Delta') \\ n & : \text{nat}^\Delta \uplus \Delta' \end{aligned}$$

On peut définir une opération cbind à partir de myrec comme dans la section 4.3 ou plus directement par :

$$\begin{aligned} \text{cbind} & : \text{nat}^\Delta \rightarrow (\text{nat} \rightarrow A) \rightarrow A \uplus \Delta \\ & \equiv \lambda n. \lambda f. \\ & \quad \text{let rec aux } n' \text{ acc} = \\ & \quad \quad \text{match } n' \text{ with} \\ & \quad \quad | 0 \rightarrow f \text{ acc} \\ & \quad \quad | S x \rightarrow \text{aux } x (S \text{ acc}) \\ & \text{in aux } n 0 \end{aligned}$$

De même, l'opération bind introduite dans la section 4.3 peut être codée dans $\lambda_\epsilon^{\text{fix}}$ directement à l'aide de fix et match par :

$$\begin{aligned} \text{bind} & : \text{nat} \uplus \Delta \rightarrow (\text{nat} \rightarrow A) \rightarrow A \uplus \Delta \\ & \equiv \lambda n. \lambda f. \\ & \quad \text{match } n \text{ with} \\ & \quad | 0 \rightarrow f 0 \\ & \quad | S x \rightarrow f x \end{aligned}$$

Dans la suite on va utiliser la fonction cbind à l'aide de la notation clet que nous avons définie pour λ_ϵ et que l'on rappelle ci-dessous :

Notation

Si n et t sont deux termes et a une variable de terme, on notera

$$\text{clet } a = n \text{ in } t$$

le terme

$$\text{cbind } n (\lambda a. t).$$

Nous allons maintenant donner un codage possible de la division euclidienne dans $\lambda_\epsilon^{\text{fix}}$, codage qui s'avère plus naturel que celui de λ_ϵ donné en section 4.3. Mais avant cela, on montre le codage de l'opération de soustraction dans $\lambda_\epsilon^{\text{fix}}$:

```

pred  :  nat → nat ⊔ {neg}
      ≡  λn. match n with
          | 0 -> raise neg
          | S x -> x

minus :  nat → nat → nat{neg}
      ≡  fix (λf. λa. λb.
          match b with
          | 0 -> a
          | S x -> f (pred a) x)

```

La division euclidienne s'écrit alors :

```

div   :  nat → nat → nat ⊔ {divby0}
      ≡  λa. λb.
          let rec aux n =
              try
                  clet x = minus n b in
                  match x with
                  | 0 -> 1
                  | S _ -> S (aux x)
              with neg ↦ 0
          in match b with
              | 0 -> raise divby0
              | S _ -> aux a

```

où, pour le typage, les variables mentionnées dans ce terme ont les types suivants :

```

aux : nat → nat
n   : nat
x   : nat

```

et donc en particulier, $\text{minus } n \ b$ a le type nat^{neg} , en conséquence de quoi,

```
clet x = minus n b in ...
```

a le type $\text{nat} \uplus \{\text{neg}\}$.

6.1.3 Propriétés

Les seules propriétés de λ_ϵ perdues par $\lambda_\epsilon^{\text{fix}}$ sont les propriétés de normalisation. En particulier, on va montrer que toutes les propriétés de λ_ϵ données dans la section 4.4 sont aussi des propriétés de $\lambda_\epsilon^{\text{fix}}$. On se contentera dans la suite de donner les différences avec les preuves pour le cas de λ_ϵ .

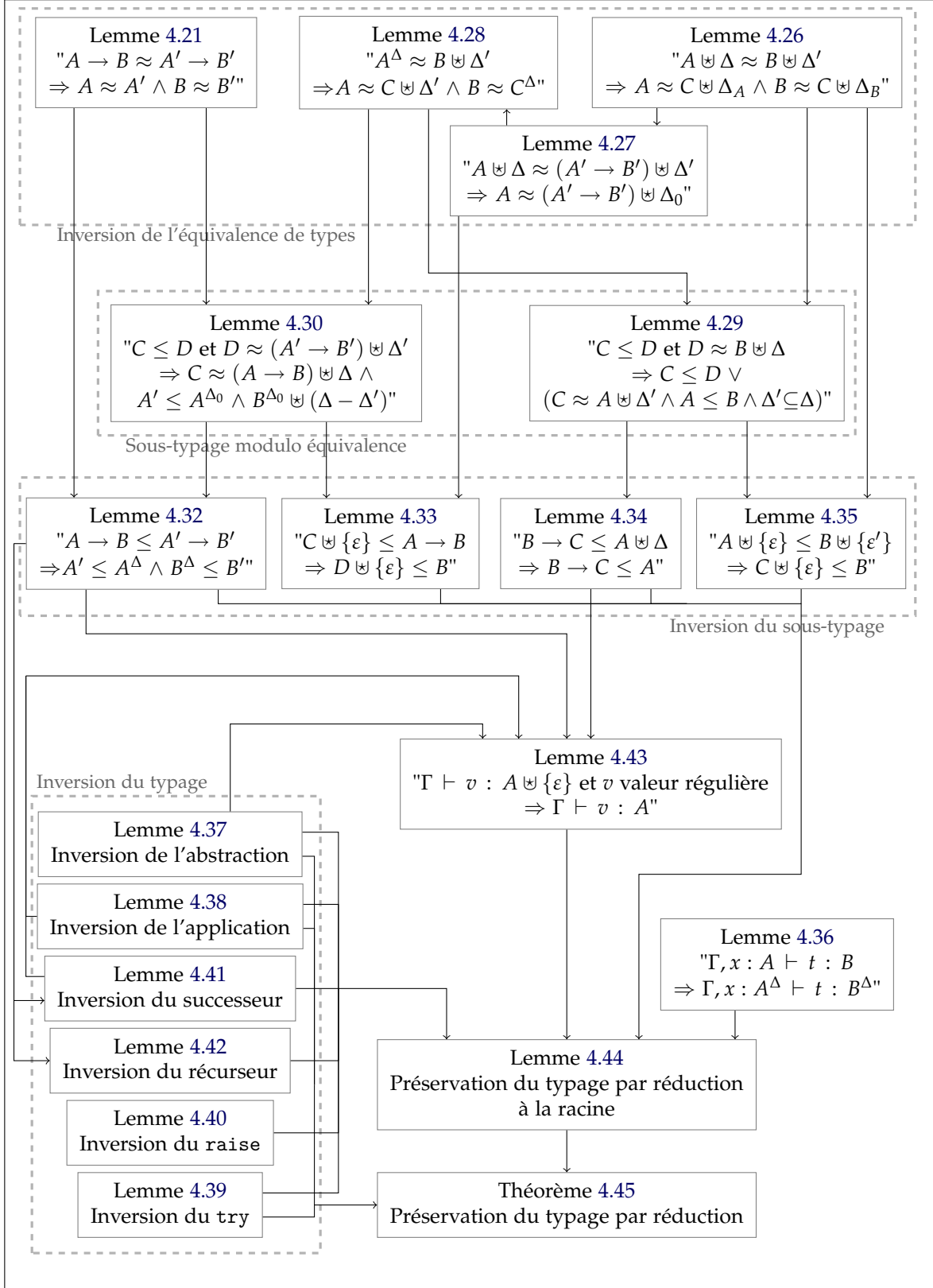


FIG. 6.1 – Rappel des dépendances pour la preuve de la préservation du typage par réduction de λ_ϵ .

Préservation du typage par réduction. On redonne figure 6.1 les principaux lemmes de la preuve pour λ_ε . Dans $\lambda_\varepsilon^{\text{fix}}$, la syntaxe des types et le sous-typage est inchangé et ainsi les lemmes d'inversion de l'équivalence, de sous-typage modulo équivalence et d'inversion du sous-typage ne nécessitent aucun changement.

L'adaptation de la preuve du lemme 4.36 pour les deux nouvelles règles de typage (*t-fix*) et (*t-match*) est trivial puisque *fix* et *match* ne mentionnent pas de variables.

Les lemmes d'inversions du typage pour les constructions déjà présentes dans le λ_ε sont également inchangés et on ajoute les deux nouveaux lemmes d'inversion suivant :

Lemme 6.1 (Inversion du point fixe) *Si A est un type et Γ un environnement de typage tel que $\Gamma \vdash \text{fix} : A$, alors il existe un type C tel que*

$$(C \rightarrow C) \rightarrow C \leq A$$

Preuve :

On raisonne par induction sur la dérivation de $\Gamma \vdash \text{fix} : A$. Les deux règles qui ont pu être appliquées sont (*t-fix*) et (*t-subtyp*). Le premier cas est direct, le second est une simple application de l'hypothèse d'induction.

Lemme 6.2 (Inversion du match) *Si A, B, C et D sont des types et Γ un environnement de typage tel que $\Gamma \vdash \text{match} : A \rightarrow B \rightarrow C \rightarrow D$, alors il existe un type X et des ensembles de noms d'exceptions Δ, Δ' et Δ'' tel que*

$$\begin{aligned} A &\leq X^\Delta \uplus \Delta' \\ B &\leq \text{nat}^{\Delta \cup \Delta'} \rightarrow X^\Delta \uplus \Delta' \\ C &\leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta'' \\ X^\Delta \uplus (\Delta' \cup \Delta'') &\leq D \end{aligned}$$

Preuve :

On commence par montrer par induction sur la dérivation du jugement $\Gamma \vdash \text{rec} : A \rightarrow B \rightarrow C \rightarrow D$ qu'il existe un type X et un ensemble de noms d'exceptions Δ' tel que

$$X \uplus \Delta' \rightarrow (\text{nat}^{\Delta'} \rightarrow X \uplus \Delta') \rightarrow \text{nat}^{\Delta'} \uplus \Delta'' \rightarrow X \uplus (\Delta' \cup \Delta'') \leq A \rightarrow B \rightarrow C \rightarrow D.$$

Alors, en appliquant plusieurs fois successivement le lemme d'inversion du sous-typage des flèches (lemme 4.32), il existe des ensembles de noms d'exceptions $\Delta_1, \Delta_2, \Delta_3$ tel que :

$$\begin{aligned} A &\leq (X \uplus \Delta')^{\Delta_1} \\ B &\leq (\text{nat}^{\Delta'} \rightarrow X \uplus \Delta')^{\Delta_1 \cup \Delta_2} \\ C &\leq \text{nat}^{\Delta' \cup \Delta_1 \cup \Delta_2 \cup \Delta_3} \uplus \Delta'' \\ (X \uplus (\Delta' \cup \Delta''))^{\Delta_1 \cup \Delta_2 \cup \Delta_3} &\leq D \end{aligned}$$

On conclut alors facilement en prenant pour ensemble de noms d'exceptions $\Delta = \Delta_1 \cup \Delta_2 \cup \Delta_3$.

En ajoutant `fix` et `match`, on a ajouté de nouvelles valeurs régulières au calcul. On doit donc adapter la preuve du lemme 4.43 :

Lemme 6.3 *Si v est une valeur régulière de $\lambda_\varepsilon^{\text{fix}}$, A un type, Γ un environnement de typage et ε un nom d'exception tel que $\Gamma \vdash v : A \uplus \{\varepsilon\}$, alors*

$$\Gamma \vdash v : A$$

Preuve :

Comme pour le lemme 4.43, on raisonne par cas sur v et tous les cas pour les valeurs déjà présentes dans λ_ε sont repris sans modification. Pour les nouvelles valeurs régulières de $\lambda_\varepsilon^{\text{fix}}$, on a :

- `fix` : On a $\Gamma \vdash \text{fix} : A \uplus \{\varepsilon\}$ donc par inversion du point fixe 6.1, il existe un type C tel que

$$(C \rightarrow C) \rightarrow C \leq A \uplus \{\varepsilon\}.$$

Alors, avec le lemme 4.34, $(C \rightarrow C) \rightarrow C \leq A$ et donc $\Gamma \vdash \text{fix} : A$.

- `match`, `match z` et `match z f` : les types des constructions `rec` de λ_ε et `match` de $\lambda_\varepsilon^{\text{fix}}$ sont très proches. La seule différence est que le type du second argument de `rec` est $\text{nat}^\Delta \rightarrow A \uplus \Delta \rightarrow A \uplus \Delta$, alors que le type de celui de `match` est simplement $\text{nat}^\Delta \rightarrow A \uplus \Delta$. Ainsi, il est facile d'adapter les cas de `rec`, `rec z` et `rec z f` de la preuve du lemme 4.43 et nous le laissons en exercice au lecteur.

On a maintenant tous les éléments pour montrer que la réduction de $\lambda_\varepsilon^{\text{fix}}$ préserve les types :

Théorème 6.4 (Préservation du typage par la réduction pour $\lambda_\varepsilon^{\text{fix}}$) *Si t et t' sont deux termes de $\lambda_\varepsilon^{\text{fix}}$, A un type et Γ un environnement de typage tel que $\Gamma \vdash t : A$ et $t \rightarrow t'$, alors*

$$\Gamma \vdash t' : A$$

Preuve :

Pour toutes les réductions de λ_ε , on reprend la preuve du théorème 4.45 puisque tous les lemmes ont été repris sans modification ou adapté ci-dessus. Pour les nouvelles règles de réductions :

- (*fix*) On a $\Gamma \vdash \text{fix } f : A$ et on doit montrer que $\Gamma \vdash f (\text{fix } f) : A$. Par inversion de l'application (lemme 4.38) sur $\Gamma \vdash \text{fix } f : A$, il existe un type B tel que $\Gamma \vdash \text{fix} : B \rightarrow A$ et $\Gamma \vdash f : B$. Alors, par inversion du point fixe (lemme 6.1), il existe C tel que

$$(C \rightarrow C) \rightarrow C \leq B \rightarrow A.$$

Alors, par inversion du sous-typage des flèches (lemme 4.32), il existe Δ tel que $B \leq C^\Delta \rightarrow C^\Delta$ et $C^\Delta \leq A$. Donc $\Gamma \vdash f : C^\Delta \rightarrow C^\Delta$ et par (*t-fix*), $\Gamma \vdash \text{fix } f : C^\Delta$. Finalement, $\Gamma \vdash f (\text{fix } f) : C^\Delta$ et on conclut.

- (*matchz*) On a $\Gamma \vdash \text{match } z f 0 : A$ et on doit montrer que $\Gamma \vdash z : A$. Avec plusieurs applications du lemme d'inversion de l'application (lemme 4.38), il existe des types B , C et D tel que $\Gamma \vdash \text{match} : D \rightarrow C \rightarrow B \rightarrow A$ et (entre autre) $\Gamma \vdash z : D$. Mais alors, avec le lemme d'inversion du `match` (lemme 6.2), on obtient $D \leq A$ et donc $\Gamma \vdash z : A$.

(*matchs*) On a $\Gamma \vdash \text{match } z \text{ } f \text{ } (S \text{ } n) : A$ et on doit montrer que $\Gamma \vdash f \text{ } n : A$. Avec plusieurs applications du lemme d'inversion de l'application (lemme 4.38), on montre qu'il existe des types B, C et D tel que $\Gamma \vdash \text{match} : D \rightarrow C \rightarrow B \rightarrow A$, $\Gamma \vdash z : D$, $\Gamma \vdash f : C$ et $\Gamma \vdash (S \text{ } n) : B$. Avec le lemme d'inversion du match (lemme 6.2), il existe un type X et des ensembles de noms d'exceptions Δ, Δ' et Δ'' tel que

$$\begin{aligned} D &\leq X^\Delta \uplus \Delta' \\ C &\leq \text{nat}^{\Delta \cup \Delta'} \rightarrow X^\Delta \uplus \Delta' \\ B &\leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta'' \\ X^\Delta \uplus (\Delta' \cup \Delta'') &\leq A \end{aligned}$$

Donc $\Gamma \vdash S \text{ } n : \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$ et en utilisant le lemme d'inversion de l'application (lemme 4.38) couplé au lemme d'inversion du successeur (lemme 4.41), il existe un type A et ensemble de noms d'exceptions Δ_1 tel que $\Gamma \vdash n : A$, $A \leq \text{nat}^{\Delta_1}$ et $\text{nat}^{\Delta_1} \leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$. En utilisant le lemme 4.34 sur cette dernière inégalité de sous-typage, on obtient $\text{nat}^{\Delta_1} \leq \text{nat}^{\Delta \cup \Delta'}$ et donc finalement $\Gamma \vdash n : \text{nat}^{\Delta \cup \Delta'}$. On a alors $\Gamma \vdash f \text{ } n : X^\Delta \uplus \Delta'$ et puisque $X^\Delta \uplus \Delta' \cup \Delta'' \leq A$, on a finalement $\Gamma \vdash f \text{ } n : A$.

(*matchr*) On a $\Gamma \vdash \text{match } z \text{ } f \text{ } (\text{raise } \varepsilon) : A$ et on doit montrer que $\Gamma \vdash \text{raise } \varepsilon : A$. Avec plusieurs applications du lemme d'inversion de l'application (lemme 4.38) puis du lemme d'inversion du match (lemme 6.2), il existe un type X et des ensembles de noms d'exceptions Δ, Δ' et Δ'' tel que $\Gamma \vdash \text{raise } \varepsilon : \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$ et $X^\Delta \uplus (\Delta' \cup \Delta'') \leq A$. Alors avec le lemme d'inversion du raise il existe un type C tel que $C \uplus \{\varepsilon\} \leq \text{nat}^{\Delta \cup \Delta'} \uplus \Delta''$ et il est alors simple de montrer qu'alors $\varepsilon \in \Delta \cup \Delta' \cup \Delta''$ d'où on a $\Gamma \vdash \text{raise } \varepsilon : X^\Delta \uplus (\Delta' \cup \Delta'')$. Finalement, $\Gamma \vdash \text{raise } \varepsilon : A$.

Préservation du typage par η -réduction. La preuve est reprise sans aucune modification puisqu'elle n'utilise que des lemmes qui ne sont pas modifiés dans $\lambda_\varepsilon^{\text{fix}}$.

Progression du calcul. Remarquons d'abord que le lemme 4.48 n'est pas invalidé par l'ajout de `fix` et `match` puisque toutes les nouvelles valeurs ont un type flèche (et ne peuvent par conséquent pas être des valeurs de type $\text{nat}^\Delta \uplus \Delta'$). On montre alors que $\lambda_\varepsilon^{\text{fix}}$ vérifie le théorème de progression du calcul :

Théorème 6.5 (Progression du calcul pour $\lambda_\varepsilon^{\text{fix}}$) Si t est un terme de $\lambda_\varepsilon^{\text{fix}}$ et A un type tels que $\vdash t : A$, alors soit t est une valeur, soit il existe un terme t' tel que

$$t \rightarrow t'$$

Preuve :

On reprend la preuve du théorème 4.47. Les nouvelles règles de typage (*t-fix*) et (*t-match*) ne pose pas de problèmes puisque `fix` et `match` sont des valeurs. Dans le cas de l'application (règle (*t-app*)), on doit prendre en compte les nouveaux cas suivant : `fix` u , `match` u , (`match` z) u et (`match` z f) u . Pour le cas `fix` u , on a toujours réduction (vers u (`fix` u)). Les cas `match` u et (`match` z) u sont des valeurs. Le cas de (`match` z f) u est une adaptation simple du cas de (`rec` z f) u que l'on laisse en exercice au lecteur.

Corruption des termes. L'ajout de `fix` et `match` ne modifie ni la définition de la relation de corruption (définition 4.49), ni le théorème de corruption (théorème 4.50) qui sont repris inchangés.

Sureté vis-à-vis des exceptions. Les preuves du théorème 4.52 et du lemmes 4.53 sont reprises de λ_ε sans modification, si ce n'est qu'elles utilisent la préservation du typage par réduction dont la preuve a été adaptée ci-dessus.

6.2 Types de données

Afin de garder la présentation de λ_ε concise, nous avons présenté le calcul muni uniquement du type de données des entiers naturels. Il n'est toutefois pas difficile d'y ajouter d'autres type de données comme nous allons le montrer dans cette section.

6.2.1 Les booléens

Pour les booléens, les syntaxes des termes, des types et des valeurs régulières sont enrichies de la façon suivante :

$$\begin{aligned} t &::= \dots \mid \text{true} \mid \text{false} \mid \text{ifthenelse} \\ A &::= \dots \mid \text{bool} \\ v &::= \dots \mid \text{true} \mid \text{false} \mid \text{ifthenelse} \mid \text{ifthenelse } t \end{aligned}$$

La notion de réduction est étendue avec les trois nouvelles règles suivantes :

$$\begin{aligned} \text{ifthenelse } t \ t' \ \text{true} &\rightarrow t \\ \text{ifthenelse } t \ t' \ \text{false} &\rightarrow t' \\ \text{ifthenelse } t \ t' \ (\text{raise } \varepsilon) &\rightarrow \text{raise } \varepsilon \end{aligned}$$

De plus, lorsque la construction `ifthenelse` est complètement appliquée, on utilisera la notation suivante pour simplifier la lecture :

Notation

Si b , t et t' sont des termes, alors on utilisera la notation

$$\text{if } b \text{ then } t \text{ else } t'$$

pour le terme

$$\text{ifthenelse } t \ t' \ b$$

.

Le type des booléens est un simple type énuméré possédant seulement deux formes canoniques : `true` et `false`. La seule façon de corrompre un tel terme est donc de le corrompre à la racine. Autrement dit, sémantiquement, les types bool^Δ et $\text{bool} \uplus \Delta$ sont égaux. Au niveau du typage, on ajoute donc la règle de sous-typage

$$\frac{}{\text{bool}^\Delta \leq \text{bool} \uplus \Delta}$$

et les règles de typage suivantes :

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$\frac{}{\Gamma \vdash \text{ifthenelse} : A \rightarrow A \rightarrow \text{bool} \uplus \Delta \rightarrow A \uplus \Delta}$$

On peut toutefois, si on le souhaite, éviter l'ajout de la nouvelle règle de sous-typage $\text{bool}^\Delta \leq \text{bool} \uplus \Delta$ si on change la règle de typage de `ifthenelse` par

$$\frac{}{\Gamma \vdash \text{ifthenelse} : A \rightarrow A \rightarrow \text{bool}^\Delta \rightarrow A \uplus \Delta}$$

et que l'on utilise, lorsque l'on veut passer de la corruption à l'union pour un booléen, le terme

$$\frac{\vdots}{\vdash \lambda b. \text{if } b \text{ then true else false} : \text{bool}^\Delta \rightarrow \text{bool} \uplus \Delta}$$

6.2.2 Le type `unit`

De façon standard, le type `unit` permet de coder des fonctions sans argument. L'ajout de ce type `unit` consiste en l'ajout aux syntaxes des termes, types et valeurs régulières de :

$$\begin{array}{l} t ::= \dots \mid () \\ A ::= \dots \mid \text{unit} \\ v ::= \dots \mid () \end{array}$$

On ajoute également la règle de typage suivante :

$$\frac{}{\Gamma \vdash () : \text{unit}}$$

Comme pour les booléens, les types $\text{unit} \uplus \Delta$ et unit^Δ coïncident. On ajoute donc la règle de sous-typage suivante :

$$\frac{}{\text{unit}^\Delta \leq \text{unit} \uplus \Delta}$$

6.2.3 Le type produit

Pour ajouter le type produit, on commence par ajouter à la syntaxe des termes, des types et des valeurs régulières les productions suivantes :

$$\begin{array}{l} t ::= \dots \mid \langle t, t \rangle \mid \text{case}_\times \\ A ::= \dots \mid A \times A \\ v ::= \dots \mid \langle t, t \rangle \mid \text{case}_\times \mid \text{case}_\times t \end{array}$$

Par ailleurs, on ajoute à la notion de réduction les règles suivantes :

$$\begin{array}{ll} \text{case}_\times f \langle t, u \rangle & \rightarrow f t u \\ \text{case}_\times f (\text{raise } \varepsilon) & \rightarrow \text{raise } \varepsilon \end{array}$$

Pour typer les couples, on commence par ajouter les deux règles de typage suivantes :

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B}$$

$$\overline{\Gamma \vdash \text{case}_\times : (A^\Delta \rightarrow B^\Delta \rightarrow C \uplus \Delta) \rightarrow (A \times B)^\Delta \uplus \Delta' \rightarrow C \uplus (\Delta \cup \Delta')}$$

Remarquons que les couples présentent une caractéristique inédite aux autres types de données que nous avons utilisés jusqu'à présent. Le type des couples dépend d'autres types. En conséquence, lorsque l'on considère un couple corrompu, c'est à dire un terme de type $(A \times B)^\Delta$, on peut distinguer deux endroits distincts où la corruption peut se « loger » :

- La structure du couple en elle-même peut ne pas être parfaitement définie, c'est alors le couple en lui-même qui est remplacé par $\text{raise } \varepsilon$;
- Les composantes du couple peuvent également être corrompues. En particulier, le couple $\langle t, \text{raise } \varepsilon \rangle$ est un couple corrompu. Ou plus généralement, $\langle t, u \rangle$ est un couple corrompu si t et/ou u sont corrompus.

Donc un couple bien formé mais qui contient une composante corrompue est lui-même un couple corrompu et par conséquent, on ajoute la règle de sous-typage suivante :

$$\overline{A^\Delta \times B^{\Delta'} \leq (A \times B)^{\Delta \cup \Delta'}}$$

Enfin, on ajoute la règle de sous-typage qui suit et qui exprime la monotonie du sous-typage par formation de couples :

$$\frac{A \leq A' \quad B \leq B'}{A \times B \leq A' \times B'}$$

Ensuite, on peut définir les projections habituelles (avec leurs types habituels) :

$$\begin{aligned} \text{fst} & : A \times B \rightarrow A \\ & = \lambda c. \text{case}_\times (\lambda a. \lambda_. a) c \\ \text{snd} & : A \times B \rightarrow B \\ & = \lambda c. \text{case}_\times (\lambda_. \lambda b. b) c \end{aligned}$$

Mais on peut également typer ces fonctions avec un type plus précis. On a :

$$\begin{aligned} \text{fst} & : (A \times B)^\Delta \uplus \Delta' \rightarrow A^\Delta \uplus \Delta' \\ \text{snd} & : (A \times B)^\Delta \uplus \Delta' \rightarrow B^\Delta \uplus \Delta' \end{aligned}$$

6.2.4 Le type somme

Les sommes sont ajoutées au calcul en complétant la syntaxe des termes, des types et des valeurs régulières par les productions suivantes :

$$\begin{aligned} t & ::= \dots \mid \text{injl} \mid \text{injrl} \mid \text{case}_+ \\ A & ::= \dots \mid A + A \\ v & ::= \dots \mid \text{injl} \mid \text{injrl} \mid \text{injl } t \mid \text{injrl } t \mid \text{case}_+ \mid \text{case}_+ t \mid \text{case}_+ t t \end{aligned}$$

On ajoute également à la notion de réduction les règles suivantes :

$$\begin{aligned} \text{case}_+ f g (\text{injl } t) & \rightarrow f t \\ \text{case}_+ f g (\text{injrl } t) & \rightarrow g t \\ \text{case}_+ f g (\text{raise } \varepsilon) & \rightarrow \text{raise } \varepsilon \end{aligned}$$

Le typage des sommes est pris en compte grâce à l'ajout des règles suivantes :

$$\overline{\Gamma \vdash \text{injl} : A \rightarrow A + B} \quad \overline{\Gamma \vdash \text{injrl} : B \rightarrow A + B}$$

$$\frac{}{\Gamma \vdash \text{case}_+ : (A^\Delta \rightarrow C \uplus \Delta) \rightarrow (B^\Delta \rightarrow C \uplus \Delta) \rightarrow (A + B)^\Delta \uplus \Delta' \rightarrow C \uplus (\Delta \cup \Delta')}$$

Enfin, et pour des raisons similaires au cas des produits, on ajoute les deux règles de sous-typage suivantes :

$$\frac{}{A^\Delta + B^{\Delta'} \leq (A + B)^{\Delta \cup \Delta'}} \quad \frac{A \leq A' \quad B \leq B'}{A + B \leq A' + B'}$$

6.2.5 Les listes

Présentation

Les listes sont un type de données où les exceptions apparaissent naturellement. Par ailleurs les listes encapsulent des valeurs d'un autre type et nous verrons que l'on peut écrire une fonction de « décorruption » d'une liste dès que l'on sait « décorrompre » les éléments qu'elle contient.

L'ajout des listes à λ_ε s'obtient par l'ajout aux syntaxes de termes, des type et des valeurs régulières des productions suivantes :

$$\begin{aligned} t &::= \dots \mid [] \mid \text{cons} \mid \text{fold} \\ A &::= \dots \mid A \text{ list} \\ v &::= \dots \mid [] \mid \text{cons} \mid \text{cons } t \mid \text{cons } t \ u \mid \text{fold} \mid \text{fold } t \mid \text{fold } t \ t' \end{aligned}$$

Pour simplifier la lecture, on introduit les notations suivantes :

Notation

- Le type $(A \text{ list})^\Delta$ sera parfois notée $A \text{ list}^\Delta$.
- Comme il est d'usage dans les langages de programmation fonctionnels avec les listes, on utilisera la notation entre crochets. Par exemple, le terme

$$\text{cons } a \ (\text{cons } b \ (\text{cons } c \ []))$$

sera plus simplement noté

$$[a; b; c].$$

On enrichit alors la notion de réduction avec les règles suivantes :

$$\begin{aligned} \text{fold } n \ c \ [] &\rightarrow n && (\text{foldn}) \\ \text{fold } n \ c \ (\text{cons } x \ l) &\rightarrow c \ x \ l \ (\text{fold } n \ c \ l) && (\text{foldc}) \\ \text{fold } n \ c \ (\text{raise } \varepsilon) &\rightarrow \text{raise } \varepsilon && (\text{foldr}) \end{aligned}$$

Les règles de typage pour les listes sont les suivantes :

$$\begin{aligned} \frac{}{\Gamma \vdash [] : A \text{ list}} \quad (t\text{-nil}) \quad & \frac{}{\Gamma \vdash \text{cons} : A \rightarrow A \text{ list} \rightarrow A \text{ list}} \quad (t\text{-cons}) \\ \frac{}{\Gamma \vdash \text{fold} : B \uplus \Delta \rightarrow (A^\Delta \rightarrow A \text{ list}^\Delta \rightarrow B \uplus \Delta \rightarrow B \uplus \Delta) \rightarrow (A \text{ list}^\Delta) \uplus \Delta' \rightarrow B \uplus (\Delta \cup \Delta')} \quad (t\text{-fold}) \end{aligned}$$

Enfin, et semblablement au cas des couples et des sommes, on ajoute les deux règles de sous-typage suivantes :

$$\frac{}{A^\Delta \text{ list} \leq A \text{ list}^\Delta} \text{ (st-list)} \qquad \frac{A \leq B}{A \text{ list} \leq B \text{ list}} \text{ (st-list-ctx)}$$

Remarquons que pour les listes corrompues, on peut distinguer divers cas de corruption :

- Une liste peut avoir des éléments corrompus. La liste en elle-même est parfaitement bien formée, mais certains de ses éléments sont, eux, potentiellement corrompus. Un exemple d’une telle liste est

$$\text{cons } a \text{ (cons (raise } \varepsilon) \text{ cons } c \text{ []) ;}$$

- Une liste elle-même peut être corrompue, sa structure peut ne pas être complètement définie. Un exemple d’une telle liste est

$$\text{cons } a \text{ (cons } b \text{ (raise } \varepsilon) \text{) ;}$$

- On peut enfin avoir un mélange des deux cas précédents, une liste corrompue d’éléments corrompus. C’est le cas par exemple de

$$\text{cons } a \text{ (cons (raise } \varepsilon) \text{ (cons } c \text{ (raise } \varepsilon) \text{))).}$$

Notons que pour le premier point où seul les éléments de la liste sont corrompus, la liste a le type $A \text{ list}^\Delta$ mais a aussi le type plus précis $A^\Delta \text{ list}$. En revanche, pour les deux autres cas, on ne peut leur trouver de type plus fin permettant de les distinguer. En particulier, il n’existe pas de type exprimant que seule la structure de la liste est corrompue, ses éléments ne l’étant pas.

Manipulation des listes corrompues

Dans la section 4.3, nous avons montré comment l’on pouvait écrire une fonction de « décorruption » d’un entier naturel, dont la fonction bind_{nat} qui y est présentée est une généralisation. On va montrer que des fonctions équivalentes peuvent être écrites pour les listes.

Une liste corrompue l’est soit parce que ses éléments le sont, soit parce que sa structure est corrompue, soit les deux. On ne peut donc pas dans le cas général décorrompre complètement une liste d’objets de type A^Δ si on ne sait pas décorrompre les objets de type A^Δ . En revanche, si l’on dispose d’une fonction de décorruption sur le type A^Δ , c’est à dire une fonction de type $A^\Delta \rightarrow (A \rightarrow B) \rightarrow B \uplus \Delta$, alors on peut écrire une fonction $\text{cbind}_{[A]}$ sur les listes d’objets de type A , à savoir :

$$\begin{aligned} \text{cbind}_{[A]} &: (A^\Delta \rightarrow (A \rightarrow B) \rightarrow B \uplus \Delta) \rightarrow A \text{ list}^\Delta \rightarrow (A \text{ list} \rightarrow B) \rightarrow B \uplus \Delta \\ &\equiv \lambda \text{bind}. \lambda l. \lambda f. \\ &\quad (\text{fold } (\lambda a. f a) (\lambda e. \lambda _ . \lambda r. \lambda a. \text{bind } e (\lambda x. r (\text{cons } x a))) l) [] \end{aligned}$$

Comme dans le cas des entiers (voir section 4.3), ce terme utilise un accumulateur et pour le typer on utilise fold avec pour type résultat (le type B dans la règle de typage de fold) le type :

$$(A \text{ list} \rightarrow B \uplus \Delta) \uplus \Delta$$

Et comme pour les entiers, on peut en écrire une version plus efficace si on sait que la corruption est à la racine :

$$\begin{aligned} \text{bind}_{[A]} & : A \text{ list} \uplus \Delta \rightarrow (A \text{ list} \rightarrow B) \rightarrow B \uplus \Delta \\ & \equiv \lambda l. \lambda f. \\ & \quad \text{fold } (f \text{ []}) (\lambda e. \lambda l'. \lambda _ . f (\text{cons } e \text{ } l')) l \end{aligned}$$

Notons que puisque l'exception est à la racine, cette fonction n'a pas besoin de prendre en argument une fonction pour décorrrompre les arguments.

On peut alors décorrrompre une liste d'entiers naturels avec la fonction suivante :

$$\begin{aligned} \text{eval}_{[\text{nat}]} & : \text{nat list}^\Delta \rightarrow \text{nat list} \uplus \Delta \\ & \equiv \lambda l. \text{cbind}_{[A]} (\text{cbind}_{\text{nat}}) l (\lambda x. x) \end{aligned}$$

6.2.6 Exemples d'utilisation

Nous allons donner quelques exemples d'utilisation des exceptions avec les types de données que nous venons d'introduire. Nous allons présenter certaines fonctions sur les listes utilisant naturellement des exceptions : la fonction récupérant le $n^{\text{ième}}$ élément d'une liste et celle permettant de trouver l'élément associé à une clef dans une liste d'associations.

Mais commençons par définir le test d'égalité sur les entiers, et pour cela, le test à zéro d'un entier :

$$\begin{aligned} \text{is0} & : \text{nat} \rightarrow \text{bool} \\ & \equiv \text{rec true } (\lambda _ . \lambda _ . \text{false}) \\ \text{eq}_{\text{nat}} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{bool} \\ & \equiv \lambda x. \lambda y. \text{try} \\ & \quad (\text{rec } (\lambda a. \text{is0 } a) (\lambda _ . \lambda r. \lambda a. r (\text{pred } a)) x) y \\ & \quad \text{with is0} \mapsto \text{false} \end{aligned}$$

On peut remarquer que dans eq_{nat} , le type de $(\text{rec } (\lambda a. \text{is0 } a) (\lambda _ . \lambda r. \lambda a. r (\text{pred } a)) x) y$ est $\text{bool}^{\{\text{is0}\}}$. Mais comme $\text{bool}^{\{\text{is0}\}} \leq \text{bool} \uplus \{\text{is0}\}$, le système de type détecte bien que l'exception est capturée par le `try` englobant et la fonction eq_{nat} est bien de type $\text{nat} \rightarrow \text{nat} \rightarrow \text{bool}$.

On peut également définir les traditionnelles fonctions renvoyant la tête et le reste d'une liste :

$$\begin{aligned} \text{hd} & : A \text{ list} \rightarrow A \uplus \{\text{hd_fail}\} \\ & \equiv \text{fold } (\text{raise hd_fail}) (\lambda e. \lambda l. \lambda _ . e) \\ \text{tl} & : A \text{ list} \rightarrow A \uplus \{\text{tl_fail}\} \\ & \equiv \text{fold } (\text{raise tl_fail}) (\lambda e. \lambda l. \lambda _ . l) \end{aligned}$$

On définit ensuite la fonction retournant le $n^{\text{ième}}$ élément d'une liste d'entiers :

$$\begin{aligned} \text{nth} & : \text{nat list} \rightarrow \text{nat} \rightarrow \text{nat} \uplus \{\text{too_short}\} \\ & \equiv \lambda l. \lambda n. (\text{fold } (\lambda _ . \text{raise too_short}) \\ & \quad (\lambda e. \lambda _ . \lambda r. \lambda a. \text{if } (\text{eq}_{\text{nat}} a n) \text{ then } e \text{ else } r (S a)) l) 0 \end{aligned}$$

Remarquons que définir une fonction retournant le $n^{\text{ième}}$ d'une liste quelconque nécessiterait une fonction testant l'égalité de deux termes d'un type quelconque, ce que nous n'avons pas défini.

La fonction itérant une fonction sur une liste est définie par :

$$\begin{aligned} \text{map} & : (A \rightarrow B) \rightarrow A \text{ list} \rightarrow B \text{ list} \\ & \equiv \lambda f. \text{fold } [] (\lambda e. \lambda_. \lambda r. \text{cons } (f e) r) \end{aligned}$$

Composer une fonction qui n'est pas écrite pour gérer les exceptions avec un argument potentiellement exceptionnel est transparent pour le programmeur dans λ_ϵ . Ainsi, on peut écrire la fonction prenant une liste de listes d'entiers en argument et renvoyant la liste des $n^{\text{ième}}$ éléments simplement avec :

$$\begin{aligned} \text{nths} & : (\text{nat list}) \text{ list} \rightarrow \text{nat} \rightarrow \text{nat list}^{\{\text{too_short}\}} \\ & \equiv \lambda l. \lambda n. \text{map } (\lambda e. \text{nth } e n) l \end{aligned}$$

Alors, étant donnée une liste d'entiers l quelconque et un entier n , on peut par exemple écrire :

$$\text{hd } (\text{nths } l n) : (\text{nat} \uplus \{\text{hd_fail}\})^{\{\text{too_short}\}}$$

Si maintenant on désire capturer les exceptions qui corrompent ce résultat, on peut écrire

$$\text{try } (\text{clet } x = \text{hd } (\text{nths } l n) \text{ in try } x \text{ with } \text{hd_fail} \mapsto 0) \text{ with } \text{too_short} \mapsto 0 : \text{nat}$$

où la construction `cletin` est celle définie en section 4.3.

Finissons ces quelques exemples en donnant une autre fonction classique sur les listes, la fonction *assoc* qui étant donné un entier, renvoie le premier élément associé à cet entier dans une liste d'association. Si aucun élément n'est associé à cet entier dans la liste, la fonction lève l'exception *not_found* :

$$\begin{aligned} \text{assoc} & : (\text{nat} \times B) \text{ list} \rightarrow \text{nat} \rightarrow B \uplus \{\text{not_found}\} \\ & \equiv \lambda l. \lambda a. \\ & \quad \text{fold } (\text{raise not_found}) (\lambda e. \lambda_. \lambda r. \text{if } (\text{eq}_{\text{nat}} (\text{fst } e) a) \text{ then } \text{snd } e \text{ else } r) l \end{aligned}$$

6.2.7 Propriétés

Nous n'allons nous intéresser qu'au cas des listes introduites section 6.2.5, les cas des autres types de données que l'on a définis (booléen, unit, produits et sommes) étant plus simples.

De plus, même si nous pensons que l'ajout des listes ne remet pas en cause la préservation du typage par réduction, nous n'en donnons pas la preuve ici. La majorité des lemmes doivent en effet être modifiés par rapport au cas de λ_ϵ , car la syntaxe des types et le sous-typage sont modifiés.

On va en revanche montrer que le modèle de réalisabilité du chapitre 5 peut être étendu au cas des listes. Pour cela, on ajoute à l'interprétation définie section 5.2.1 (définition 5.1, page 79) l'interprétation du type des listes suivante :

$$\begin{aligned} \llbracket A \text{ list} \rrbracket^\Delta & = \{ t \mid t \rightarrow^* \text{cons } a_0 (\dots (\text{cons } a_n []) \dots), \\ & \quad n \in \mathbb{N}, \forall 0 \leq i \leq n, a_i \in \llbracket A \rrbracket^\Delta \} \\ & \cup \{ t \mid t \rightarrow^* \text{cons } a_0 (\dots (\text{cons } a_n \text{ raise } \epsilon) \dots), \\ & \quad n \in \mathbb{N}, \forall 0 \leq i \leq n, a_i \in \llbracket A \rrbracket^\Delta, \epsilon \in \Delta \} \end{aligned}$$

On va alors montrer que la preuve de correction du typage vis-à-vis de l'interprétation peut être adaptée pour prendre en compte cette nouvelle interprétation. On commence par rappeler figure 6.2 les lemmes utilisés pour la preuve de ce résultat dans λ_ε et on va donner les ajouts à y effectuer pour le type des listes.

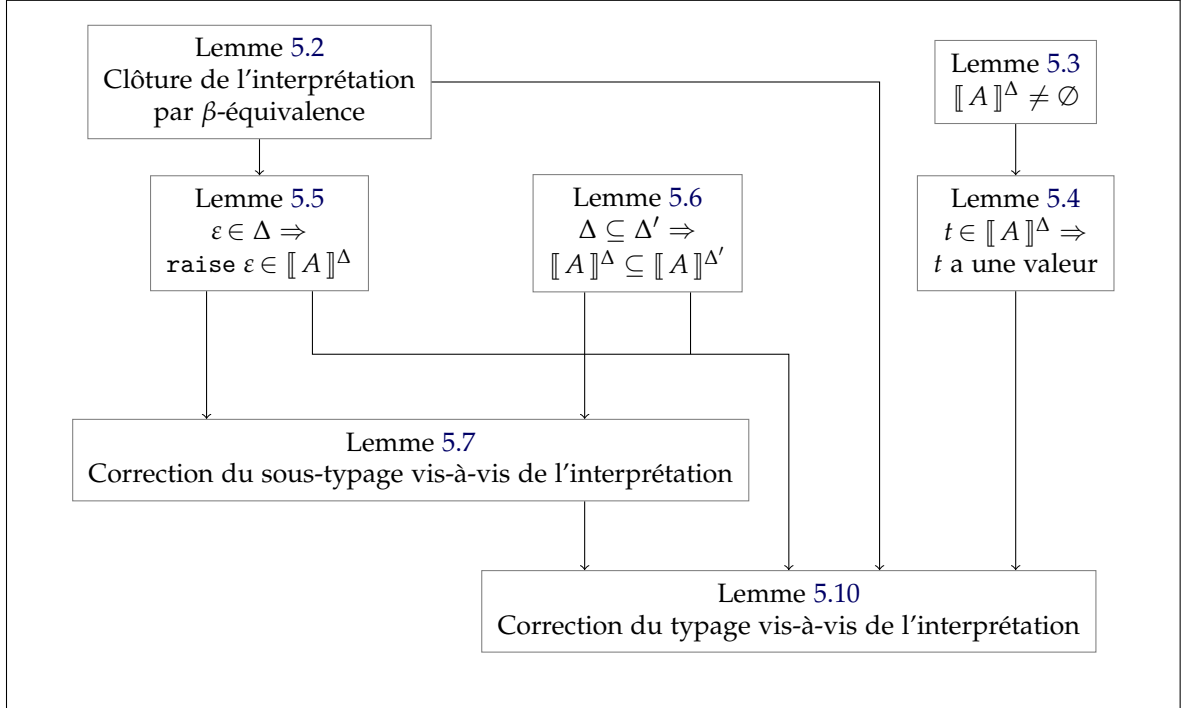


FIG. 6.2 – Rappel des dépendances pour la preuve de correction du typage de λ_ε vis-à-vis de l'interprétation

Notons d'abord que les preuves des lemmes « préliminaires » (lemmes 5.2, 5.3, 5.4, 5.5 et 5.6) s'effectuent par induction sur la définition de l'interprétation. On doit donc à chaque fois y ajouter le cas de l'interprétation des listes. Ces cas n'y posent cependant pas de difficultés :

- Pour le lemme 5.2, l'interprétation des listes est closes par β -équivalence car la réduction est confluyente et par hypothèse d'induction ;
- Pour le lemme 5.3, l'ensemble $[[A \text{ list}]]^\Delta$ contient toujours au moins la liste vide ;
- Pour les lemmes 5.4 et 5.5, le cas des listes est trivial par définition de l'interprétation. ;
- Pour le lemme 5.6, le cas des listes est une simple application de l'hypothèse d'induction.

La preuve du lemme 5.7 est non modifiée pour toutes les règles de sous-typage de λ_ε . Pour les deux nouvelles règles de sous-typage des listes ($(st\text{-}list)$ et $(st\text{-}list\text{-}ctx)$), on montre les deux résultats suivants :

Lemme 6.6 (Validation de la règle $(st\text{-}list)$ par l'interprétation) Si A est un type et Δ et Δ' sont deux ensembles de noms d'exceptions, alors

$$[[A^{\Delta'} \text{ list}]]^\Delta \subseteq [[A \text{ list}^{\Delta'}]]^\Delta$$

Preuve :

Si $l \in \llbracket A^{\Delta'} \text{ list} \rrbracket^{\Delta}$, alors par définition, l est de la forme $\text{cons } a_0 (\dots (\text{cons } a_n X) \dots)$ (où X est soit $[]$, soit $\text{raise } \varepsilon$, soit $\text{raise } \varepsilon$ pour $\varepsilon \in \Delta$) avec, quel que soit n et quel que soit i tel que $0 \leq i \leq n$, $a_i \in \llbracket A^{\Delta'} \rrbracket^{\Delta} = \llbracket A \rrbracket^{\Delta \cup \Delta'}$. Mais puisque $\Delta \subseteq \Delta \cup \Delta'$, il est facile de vérifier que par définition $l \in \llbracket (A \text{ list})^{\Delta'} \rrbracket^{\Delta}$.

Lemme 6.7 (Validation de la règle (*st-list-ctx*) par l'interprétation) Si A et B sont deux types et Δ est un ensemble de noms d'exceptions tel que $\llbracket A \rrbracket^{\Delta} \subseteq \llbracket B \rrbracket^{\Delta}$, alors

$$\llbracket A \text{ list} \rrbracket^{\Delta} \subseteq \llbracket B \text{ list} \rrbracket^{\Delta}$$

Preuve :

Si $l \in \llbracket A \text{ list} \rrbracket^{\Delta}$, alors l est de la forme $\text{cons } a_0 (\dots (\text{cons } a_n X) \dots)$ (où X est soit $[]$, soit $\text{raise } \varepsilon$, soit $\text{raise } \varepsilon$ pour $\varepsilon \in \Delta$) avec, quel que soit n et quel que soit i tel que $0 \leq i \leq n$, $a_i \in \llbracket A \rrbracket^{\Delta}$. Donc par hypothèse, pour tout i , $a_i \in \llbracket B \rrbracket^{\Delta}$ et on a $l \in \llbracket B \text{ list} \rrbracket^{\Delta}$.

Remarquons que la définition de substitution parallèle (définition 5.9) s'étend trivialement au cas des listes puisque les termes ajoutés au calcul sont de simple constantes.

Enfin, pour montrer que l'interprétation augmentée du cas des listes valide le typage de λ_{ε} auquel on a ajouté les listes, on montre les trois résultats suivants exprimant la validation des règles de typage des listes :

Lemme 6.8 (Validation de la règle (*t-nil*) par l'interprétation) Si A est un type et Δ un ensemble de noms d'exceptions, alors

$$[] \in \llbracket A \text{ list} \rrbracket^{\Delta}$$

Preuve :

Par définition de $\llbracket A \text{ list} \rrbracket^{\Delta}$.

Lemme 6.9 (Validation de la règle (*t-cons*) par l'interprétation) Si A est un type et Δ un ensemble de noms d'exceptions, alors

$$\text{cons} \in \llbracket A \rightarrow A \text{ list} \rightarrow A \text{ list} \rrbracket^{\Delta}$$

Preuve :

On doit montrer que pour tous ensembles de noms d'exceptions $\Delta_2 \supseteq \Delta_1 \supseteq \Delta$ et tous termes $x \in \llbracket A \rrbracket^{\Delta_1}$ et $l \in \llbracket A \text{ list} \rrbracket^{\Delta_2}$, on a $\text{cons } x \ l \in \llbracket A \text{ list} \rrbracket^{\Delta_2}$. Mais puisque $\llbracket A \rrbracket^{\Delta_1} \subseteq \llbracket A \rrbracket^{\Delta_2}$ (lemme 5.6), on conclut par définition de $\llbracket A \text{ list} \rrbracket^{\Delta_2}$.

Lemme 6.10 (Validation de la règle (t-fold) par l'interprétation) Si A et B sont deux types Δ, Δ' et Δ'' des ensembles de noms d'exceptions, alors

$$\text{fold} \in \llbracket B \uplus \Delta' \rightarrow (A^{\Delta'} \rightarrow A \text{ list}^{\Delta'} \rightarrow B \uplus \Delta' \rightarrow B \uplus \Delta') \rightarrow (A \text{ list}^{\Delta'}) \uplus \Delta'' \rightarrow B \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta}$$

Preuve :

Par définition de l'interprétation des flèches, on doit montrer que quel que soit les ensembles de noms d'exceptions Δ_1, Δ_2 et Δ_3 tels que $\Delta \subseteq \Delta_1 \subseteq \Delta_2 \subseteq \Delta_3$, et si

- $n \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_1}$,
- $c \in \llbracket A^{\Delta'} \rightarrow A \text{ list}^{\Delta'} \rightarrow B \uplus \Delta' \rightarrow B \uplus \Delta' \rrbracket^{\Delta_2}$,
- $l \in \llbracket A \text{ list}^{\Delta'} \uplus \Delta'' \rrbracket^{\Delta_3} = \llbracket A \text{ list} \rrbracket^{\Delta_3 \cup \Delta'} \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta'' \}$,

alors $\text{fold } n \ c \ l \in \llbracket B \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$.

Si $l \in \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta'' \}$, alors on a $\text{fold } n \ c \ l \rightarrow^* \text{fold } n \ c \ \text{raise } \varepsilon \rightarrow \text{raise } \varepsilon$ et puisque $\text{raise } \varepsilon \in \llbracket B \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$ par définition et que l'interprétation est close par anti-réduction (lemme 5.2), on a bien $\text{fold } n \ c \ l \in \llbracket B \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$.

Sinon, $l \in \llbracket A \text{ list} \rrbracket^{\Delta_3 \cup \Delta'}$. En utilisant Θ pour représenter soit $[]$, soit $\text{raise } \varepsilon$ (où $\varepsilon \in \Delta_3 \cup \Delta'$), alors par définition il existe $k \in \mathbb{N}$ et une famille de terme $(a_i)_{0 \leq i \leq k}$ tel que $a_i \in \llbracket A \rrbracket^{\Delta_3 \cup \Delta'}$ et $l \rightarrow^* \text{cons } a_0 \ (\dots (\text{cons } a_k \ \Theta) \dots)$. Donc $\text{fold } n \ c \ l \rightarrow^* \text{fold } n \ f \ (\text{cons } a_0 \ (\dots (\text{cons } a_k \ \Theta) \dots))$ et on va prouver que

$$\text{fold } n \ f \ (\text{cons } a_0 \ (\dots (\text{cons } a_k \ \Theta) \dots)) \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_3}$$

par induction sur k (ce qui nous permettra de conclure puisque $\llbracket B \uplus \Delta' \rrbracket^{\Delta_3} \subseteq \llbracket B \uplus (\Delta' \cup \Delta'') \rrbracket^{\Delta_3}$ par définition de ces interprétations) :

- Si $k = 0$, alors :
 - Soit $l \rightarrow^* []$, alors $\text{fold } n \ c \ [] \rightarrow n$ et puisque $n \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_1}$ et $\Delta_1 \subseteq \Delta_3$, on a $n \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_3}$ (lemme 5.6). On conclut donc par clôture de l'interprétation par anti-réduction (lemme 5.2).
 - Soit $l \rightarrow^* \text{raise } \varepsilon$ où $\varepsilon \in \Delta_3 \cup \Delta'$, alors $\text{fold } n \ c \ \text{raise } \varepsilon \rightarrow \text{raise } \varepsilon$. Or si $\varepsilon \in \Delta_3$, $\text{raise } \varepsilon \in \llbracket B \rrbracket^{\Delta_3}$ (lemme 5.5) et donc par définition de l'interprétation de l'union, si $\varepsilon \in \Delta_3 \cup \Delta'$, alors $\text{raise } \varepsilon \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_3}$. On conclut par clôture de l'interprétation par anti-réduction (lemme 5.2).
- Si $k = k' + 1$, c'est donc que $l \rightarrow^* \text{cons } a \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots))$. On a alors la réduction

$$\begin{aligned} & \text{fold } n \ c \ (\text{cons } a \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots))) \\ & \rightarrow c \ a \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots)) \ (\text{fold } n \ c \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots))). \end{aligned}$$

Par hypothèse d'induction $\text{fold } n \ c \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots)) \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_3}$ et par définition de l'interprétation, $\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots) \in \llbracket A \text{ list}^{\Delta'} \rrbracket^{\Delta_3}$. Comme de plus $c \in \llbracket A^{\Delta'} \rightarrow A \text{ list}^{\Delta'} \rightarrow B \uplus \Delta' \rightarrow B \uplus \Delta' \rrbracket^{\Delta_2}$ et $\Delta_2 \subseteq \Delta_3$, on montre que

$$c \ a \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots)) \ (\text{fold } n \ c \ (\text{cons } a_0 \ (\dots (\text{cons } a_{k'} \ \Theta) \dots))) \in \llbracket B \uplus \Delta' \rrbracket^{\Delta_3}$$

et on conclut alors par clôture de l'interprétation par anti-réduction (lemme 5.2).

Maintenant que l'on a montré que le typage des listes était correct vis-à-vis de l'interprétation, on peut montrer le lemme suivant :

Lemme 6.11 *Si t est un terme tel que $\vdash t : \text{nat list}$, alors il existe un entier $k \geq 0$ et une famille d'entier $(n_i)_{0 \leq i \leq k}$ tel que*

$$t \rightarrow^* \text{cons} (S^{n_0} 0) (\dots (\text{cons} (S^{n_k} 0) []) \dots)$$

Preuve :

C'est un conséquence direct de la correction du typage vis-à-vis de l'interprétation et de la définition de l'interprétation.

6.3 Des entiers courageux

Les entiers de λ_ε sont paresseux. Le terme $S t$ est une valeur quel que soit le terme t et en particulier si t n'est pas évalué. Par conséquent $S (\text{raise } \varepsilon)$ est une valeur et la construction `try` ne capture pas l'exception d'un tel terme. Nous avons fait ce choix dans λ_ε car nous le pensons cohérent avec la vision que nous voulons de l'appel par nom. La fonction successeur est vue comme n'importe quelle autre fonction du calcul et en tant que tel ne force pas l'évaluation de son argument. Mais nous avons également fait ce choix parce que nous pensons qu'il est particulièrement général.

On peut en effet considérer que la fonction successeur n'est pas une fonction comme les autres et en ce sens lui donner un comportement différent des autres fonctions, à savoir évaluer son argument. Mais calculer avec un tel successeur peut être vu comme une simple restriction de λ_ε . Ou plus exactement, ce nouveau successeur non paresseux que nous allons noter $\dot{S}$ se code par :

$$\dot{S} \equiv \lambda n. \text{eval}_{\text{nat}} (S n)$$

où eval_{nat} est la fonction définie section 4.3. En particulier, on peut vérifier que ce nouveau successeur a la réduction suivante

$$\dot{S} (\text{raise } \varepsilon) \rightarrow \text{raise } \varepsilon$$

et a entre autres les types suivants :

$$\begin{aligned} \dot{S} &: \text{nat} \rightarrow \text{nat} \\ &: \text{nat}^\Delta \rightarrow \text{nat} \uplus \Delta \\ &: \text{nat} \uplus \Delta \rightarrow \text{nat} \uplus \Delta \end{aligned}$$

Bien sûr ce codage a un coût algorithmique et si en pratique on ne souhaite conserver que ce second récursif et ne pas subir le coût algorithmique de eval_{nat} , on peut reformuler de façon un peu plus simple le calcul en supprimant les termes de la forme $S t$ de la définition de valeur régulière (et donc par extension en les supprimant de la définition de valeur) et en ajoutant la règle de réduction

$$S (\text{raise } \varepsilon) \rightarrow \text{raise } \varepsilon$$

On ajoute alors la règle de sous-typage

$$\text{nat}^\Delta \leq \text{nat} \uplus \Delta$$

et on peut simplifier la règle de typage du récursur en lui donnant simplement la règle :

$$\frac{}{\Gamma \vdash \text{rec} : A \rightarrow (\text{nat} \rightarrow A \rightarrow A) \rightarrow \text{nat} \uplus \Delta \rightarrow A \uplus \Delta}$$

Mais le calcul ainsi formulé est un cas particulier de λ_ε , raison pour laquelle nous avons choisi d'étudier λ_ε dans toute sa généralité.

6.4 Ajouter des arguments aux exceptions ?

Dans la plupart des langages de programmation possédants un mécanisme d'exceptions, celles-ci ont la possibilité de transporter des arguments. Ceci permet d'ajouter des informations précises sur le cas qui a levé l'exception, informations qui peuvent alors être utilisées lorsque l'exception est rattrapée.

Il est donc naturel de se poser la question de l'ajout d'arguments aux exceptions du calcul λ_ε . Cependant, la notion de corruption ne rend pas évidente une telle extension. La présente section montre un certain nombre de difficultés posées par le typage d'arguments d'exceptions. La possibilité de résoudre ces difficultés reste une question ouverte de cette thèse.

Termes et leurs réductions. Commençons par remarquer que l'ajout d'arguments d'exceptions ne pose pas de challenge majeur d'un point de vue non typé, c'est à dire pour les termes et leur réductions. Pour ce faire, on peut par exemple proposer de modifier la syntaxe de λ_ε de la façon suivante :

$$t ::= x \mid \lambda x. t \mid t t \mid \text{raise } \varepsilon(t) \mid \text{try } t \text{ with } \varepsilon(x) \mapsto t \mid 0 \mid S \mid \text{rec}$$

Lorsqu'on lève une exception, on doit également passer son argument. Lorsqu'on capture une exception, on récupère cet argument, qui peut alors être utilisé dans le gestionnaire du try. Pour cela, dans le terme $\text{try } t \text{ with } \varepsilon(x) \mapsto u$, la variable x est liée dans u .

Remarquons que cette proposition de syntaxe considère que chaque exception prend toujours un et un seul argument. Mais ceci sans perte de généralité puisqu'il est en pratique facile de simuler des exceptions sans argument avec le type `unit` (section 6.2.2) et les exceptions avec plusieurs arguments avec les produits (section 6.2.3) ou les listes (section 6.2.5).

Ces changements n'imposent en revanche aucune modification de la définition des valeurs régulières qui sont toujours définies par :

$$v ::= \lambda x. t \mid 0 \mid S \mid S t \mid \text{rec } t \mid \text{rec } t t$$

Enfin, pour les réductions de ces nouvelles versions de `raise` et `try`, on modifie la notion de réduction de λ_ε de la façon suivante :

$$\begin{array}{lll}
(\lambda x. t) u & \rightarrow & t\{x := u\} & (\beta) \\
(\text{raise } \varepsilon(t)) u & \rightarrow & \text{raise } \varepsilon(t) & (\text{rapp}) \\
\text{try } (\text{raise } \varepsilon(t)) \text{ with } \varepsilon(x) \mapsto u & \rightarrow & u\{x := t\} & (\text{catch}) \\
\text{try } (\text{raise } \varepsilon'(t)) \text{ with } \varepsilon(x) \mapsto u & \rightarrow & \text{raise } \varepsilon'(t) & (\text{trai}) \\
\text{try } v \text{ with } \varepsilon(x) \mapsto u & \rightarrow & v & (\text{tval}) \\
\text{rec } z \text{ f } 0 & \rightarrow & s & (\text{recz}) \\
\text{rec } z \text{ f } (S n) & \rightarrow & f n (\text{rec } z \text{ f } n) & (\text{recs}) \\
\text{rec } z \text{ f } (\text{raise } \varepsilon(t)) & \rightarrow & \text{raise } \varepsilon(t) & (\text{recr})
\end{array}$$

La réduction $\rightarrow$ étant toujours définie comme la clôture contextuelle de la notion de réduction $\rightarrow$.

Typage. Comment tenir compte du type des arguments d'exception ? Adapter le système de type de λ_ε à cette extension pose certaines difficultés que nous allons présenter ici.

Une première idée naïve serait de fixer *a priori* un type pour l'exception ε , c'est à dire à associer à chaque exception ε un certain type $\tau(\varepsilon)$. On typerait alors les constructions `raise` et `try` avec les règles suivantes :

$$\frac{\vdash t : \tau(\varepsilon)}{\Gamma \vdash \text{raise } \varepsilon(t) : A \uplus \{\varepsilon\}} \quad (t\text{-raise}) \qquad \frac{\Gamma \vdash t : A \uplus \{\varepsilon\} \quad \Gamma, x : \tau(\varepsilon) \vdash u : A}{\Gamma \vdash \text{try } t \text{ with } \varepsilon(x) \mapsto u : A} \quad (t\text{-try})$$

Un avantage de cette idée est que ni la syntaxe des types, ni le sous-typage ne nécessitent de modification. Un désavantage en est toutefois l'absence de préservation des types par réduction. En effet, supposons que pour les exceptions ε et ε' , le type de leurs arguments soit nat , c'est à dire, $\tau(\varepsilon) = \tau(\varepsilon') = \text{nat}$ et considérons le terme $t = \lambda x. \text{raise } \varepsilon(x)$. Ce terme est de type $\text{nat} \rightarrow A \uplus \{\varepsilon\}$ et par corruption est de type $\text{nat}^{\{\varepsilon'\}} \rightarrow A^{\{\varepsilon'\}} \uplus \{\varepsilon\}$. Donc $t(\text{raise } \varepsilon'(0))$ a le type $A^{\{\varepsilon'\}} \uplus \{\varepsilon\}$. Mais $t(\text{raise } \varepsilon'(0))$ se réduit sur $\text{raise } \varepsilon(\text{raise } \varepsilon'(0))$ qui n'est pas bien typé (car $\text{raise } \varepsilon'(0)$ n'est pas de type nat).

Autrement dit, comme pour tout terme, il est possible qu'un argument d'exception soit corrompu lors de la réduction et on ne peut donc pas définir *a priori* le type de l'argument d'une exception.

La solution à ce problème semble être de modifier la syntaxe des types pour y mentionner le type de l'argument d'une exception. Pour cela, dans les constructions $A \uplus \Delta$ et A^Δ , Δ ne doit plus se contenter d'indiquer les noms des exceptions, mais aussi les types des arguments de ces exceptions. Ainsi, par exemple, le type

$$A^{\{\varepsilon(\text{nat}), \varepsilon'(\text{nat} \rightarrow \text{nat})\}}$$

sera le type des termes de type A où on a remplacé certains sous-termes par $\text{raise } \varepsilon(t)$, avec t un terme de type nat , ou par $\text{raise } \varepsilon'(u)$, avec u un terme de type $\text{nat} \rightarrow \text{nat}$.

L'adaptation du sous-typage de λ_ε à cette nouvelle sémantique des types n'est cependant pas claire. Considérons par exemple le terme

$$t = \lambda x. f x (\text{raise } \varepsilon(x))$$

où f est une fonction quelconque de type $\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$. Ce terme est de type

$$\text{nat} \rightarrow \text{nat}^{\{\varepsilon(\text{nat})\}}$$

et donc selon le principe de corruption, il est aussi de type

$$\text{nat}^{\{\varepsilon'(\text{nat})\}} \rightarrow (\text{nat}^{\{\varepsilon(\text{nat})\}})^{\{\varepsilon'(\text{nat})\}}.$$

Ainsi, le terme $t(\text{raise } \varepsilon'(0))$ a le type

$$(\text{nat}^{\{\varepsilon(\text{nat})\}})^{\{\varepsilon'(\text{nat})\}}.$$

D'après les intuitions que nous avons données et ce terme se réduisant sur

$$f(\text{raise } \varepsilon'(0))(\text{raise } \varepsilon(\text{raise } \varepsilon'(0))),$$

on doit alors avoir l'égalité de type

$$(\text{nat}^{\{\varepsilon(\text{nat})\}})^{\{\varepsilon'(\text{nat})\}} \doteq \text{nat}^{\{\varepsilon(\text{nat}^{\{\varepsilon'(\text{nat})\}}), \varepsilon'(\text{nat})\}}$$

Peut-on adapter le sous-typage de λ_ε à cette nouvelle sémantique des types et avoir de telles égalités de type ? Nous n'avons pas exploré cette problématique plus loin dans cette thèse et laissons cette question comme perspective.

Exceptions et second-ordre

Nous présentons une extension du λ -calcul du second-ordre reprenant les exceptions et le typage introduit dans les chapitres précédents, montrant ainsi que ce typage est compatible avec le polymorphisme de type.

La corruption est définie par sous-typage et nous allons donc étendre le λ -calcul du second-ordre avec sous-typage. Nous commencerons par donner quelques rappels sur ce dernier calcul dans la section 7.1 avant de présenter notre extension en section 7.2, le calcul F_ε . Puis, dans la section 7.3, nous présenterons un modèle de réalisabilité de F_ε , défini à l'aide de techniques d'orthogonalité. Enfin, en section 7.4, nous évoquerons les codages au second-ordre des types de données dans F_ε .

7.1 Le λ -calcul du second-ordre avec sous-typage

Introduit de manière indépendante par J.Y. Girard [Gir72] et J.C. Reynolds [Rey74], le λ -calcul du second-ordre (ou système F) ajoute au système de types du λ -calcul simplement typé le polymorphisme de types dans sa forme la plus générale. L'intérêt de ce calcul réside également dans sa correspondance, à travers l'isomorphisme de Curry-Howard, avec l'arithmétique du second ordre [GTL89].

Le système F (à la Curry¹) ajoute aux types du λ -calcul simplement typé (à la Curry) les variables de type α et la quantification universelle $\forall\alpha. A$. Les règles de typage pour ces nouvelles constructions sont la règle de généralisation (*gen*) et la règle d'instanciation (*inst*) rappelées ci-dessous :

$$\frac{\Gamma \vdash t : A \quad \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash t : \forall\alpha. A} \text{ (gen)} \qquad \frac{\Gamma \vdash t : \forall\alpha. A}{\Gamma \vdash t : A\{\alpha := B\}} \text{ (inst)}$$

où $\text{FV}(\Gamma)$ représente l'ensemble des variables de types libres mentionnées par l'environnement de type Γ (voir définition 7.5 pour une définition formelle de cette notion).

Le système F vérifie la propriété de préservation du typage pour la β -réduction, mais il est connu qu'il ne vérifie pas cette propriété pour la η -réduction. Par exemple, le type des

¹C'est à dire où une abstraction ne mentionne pas le type de son argument, contrairement au système F présenté à la Church où l'abstraction se note $\lambda x : A. t$ avec A le type de la variable x .

entiers (`nat`) et le type des booléens (`bool`) sont codables dans le système F où l'on peut dériver le jugement suivant :

$$f : \text{nat} \rightarrow (\forall \beta. \beta) \vdash \lambda x. f x : (\forall \alpha. \alpha) \rightarrow \text{bool}$$

Mais, bien que $\lambda x. f x \rightarrow_\eta f$, on ne peut *pas* dériver le jugement :

$$f : \text{nat} \rightarrow (\forall \beta. \beta) \vdash f : (\forall \alpha. \alpha) \rightarrow \text{bool}$$

Une première façon de résoudre ce problème consiste à « forcer » la préservation du type par η -réduction en ajoutant la règle de typage suivante :

$$\frac{\Gamma \vdash \lambda x. t x : A \quad x \notin \text{FV}(t)}{\Gamma \vdash t : A} \text{ (eta)}$$

Cette idée d'introduire « de force » la règle η afin d'améliorer les propriétés du typage a été introduite par J.B Wells [Wel96].

Une autre solution est due à J.C. Mitchell [Mit88] qui a proposé d'étendre le système F à l'aide d'une relation de sous-typage. Le système de Mitchell est équivalent au système F auquel est ajouté la règle (*eta*) ci-dessus, raison pour laquelle nous y ferons référence sous le nom de système $F\eta$.

Un intérêt du système $F\eta$ est qu'il représente un cadre particulièrement général pour l'étude du sous-typage en présence de polymorphisme de type. C'est pour cette généralité que nous avons choisi de l'utiliser comme base pour notre extension avec exceptions, mais il est à noter que ses bonnes propriétés vis-à-vis de la η -réduction ne sont pas particulièrement nécessaires à notre mécanisme d'exceptions. Nous n'ajouterons d'ailleurs pas explicitement la règle η à notre calcul.

Le calcul F_ε ne se base pas sur la présentation du système $F\eta$ original de Mitchell. On y préférera une présentation similaire à celle de [TU96]) et consistant à ajouter système F une relation de sous-typage $\leq$ qui est d'une part, réflexive, transitive, covariante à droite des flèches et contravariante à gauche des flèches :

$$\begin{array}{c} \frac{}{A \leq A} \text{ (st-id)} \quad \frac{A \leq B \quad B \leq C}{A \leq C} \text{ (st-trans)} \\[10pt] \frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'} \text{ (st-arrow)} \end{array}$$

et d'autre part, vérifie les trois règles suivantes :

$$\begin{array}{c} \frac{A \leq B \quad \alpha \notin \text{FV}(A)}{A \leq \forall \alpha. B} \text{ (st-gen)} \quad \frac{}{\forall \alpha. A \leq A\{\alpha := B\}} \text{ (st-inst)} \\[10pt] \frac{\alpha \notin \text{FV}(A)}{\forall \alpha. (A \rightarrow B) \leq A \rightarrow \forall \alpha. B} \text{ (st-distr)} \end{array}$$

La règle (*st-gen*) autorise à quantifier sur une variable qui n'est pas libre, la règle (*st-inst*) est la règle d'instanciation de la quantification universelle et la règle (*st-distr*) est une version faible de distributivité de la quantification sur la flèche. Notons que la règle d'instanciation (*st-inst*) rend caduque la règle de typage d'instanciation (*inst*) du système F qui n'est donc pas présente dans le système $F\eta$. Par contre, la règle de typage de généralisation (*gen*) ne peut pas, elle, être internalisée dans le sous-typage.

Remarquons que ce sous-typage est monotone vis-à-vis de la quantification :

Lemme 7.1 Dans un calcul où le sous-typage possède les règles (st-id), (st-trans), (st-arrow), (st-gen), (st-inst) et (st-distr), si A et B sont des types tels que $A \leq B$, alors quelle que soit la variable de type α ,

$$\forall \alpha. A \leq \forall \alpha. B$$

Preuve :

Avec (st-inst), on a $\forall \alpha. A \leq A$ (puisque $A\{\alpha := \alpha\} = A$) et donc $\forall \alpha. A \leq B$. Finalement, puisque $\alpha \notin \text{FV}(\forall \alpha. A)$, en appliquant la règle (st-gen), on a $\forall \alpha. A \leq \forall \alpha. B$.

De plus, la règle (st-distr) de distributivité de la quantification sur la flèche est prise sous une version faible car la version générale est dérivable :

Lemme 7.2 Dans un calcul où le sous-typage possède les règles (st-id), (st-trans), (st-arrow), (st-gen), (st-inst) et (st-distr), si A et B sont des types et α est une variable de types, alors

$$\forall \alpha. (A \rightarrow B) \leq (\forall \alpha. A) \rightarrow (\forall \alpha. B)$$

Preuve :

Par contravariance à gauche de la flèche (règle (st-arrow)) et puisque par (st-inst), $\forall \alpha. A \leq A$, $A \rightarrow B \leq (\forall \alpha. A) \rightarrow B$. Alors, avec le lemme 7.1, $\forall \alpha. A \rightarrow B \leq \forall \alpha. (\forall \alpha. A) \rightarrow B$. Finalement, avec (st-distr), on a $\forall \alpha. (A \rightarrow B) \leq (\forall \alpha. A) \rightarrow (\forall \alpha. B)$.

7.2 Le système F_ϵ

Nous présentons le calcul F_ϵ qui est une extension du système F_η (voir section 7.1) à l'aide d'un mécanisme d'exception et d'entiers primitifs.

7.2.1 Termes : syntaxe et réductions

De même que les termes et les réductions du système F_η (à la Curry) sont les mêmes que ceux du λ -calcul simplement typé, le langage F_ϵ reprend les termes et réductions du langage λ_ϵ et en partage les propriétés. On rappelle figure 7.1 la syntaxe des termes, des valeurs et valeurs régulières et les règles de réductions pour F_ϵ et on se reportera au chapitre 3 pour de plus amples informations sur ces définitions (ainsi que pour les définitions omises ici comme la réduction de tête faible). En particulier, la réduction de F_ϵ est confluente et le théorème de standardisation y est valide.

7.2.2 Le système de types

Les types de F_ϵ sont construits au dessus de ceux de λ_ϵ en y ajoutant les variables de types et la quantification universelle.

Syntaxe des termes

$$t ::= x \mid \lambda x. t \mid t t \mid \text{raise } \varepsilon \mid \text{try } t \text{ with } \varepsilon \mapsto t \mid 0 \mid S \mid \text{rec}$$

Valeurs régulières

$$v ::= \lambda x. t \mid 0 \mid S \mid S t \mid \text{rec} \mid \text{rec } t \mid \text{rec } t t$$

Valeurs

$$v ::= v \mid \text{raise } \varepsilon$$

Notion de réduction

$$\begin{array}{lll}
(\lambda x. t) u & \rightarrow & t\{x := u\} & (\beta) \\
(\text{raise } \varepsilon) t & \rightarrow & \text{raise } \varepsilon & (rapp) \\
\text{try } (\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto u & \rightarrow & u & (catch) \\
\text{try } (\text{raise } \varepsilon') \text{ with } \varepsilon \mapsto u & \rightarrow & \text{raise } \varepsilon' & (traï) \\
\text{try } v \text{ with } \varepsilon \mapsto u & \rightarrow & v & (tval) \\
\text{rec } z f 0 & \rightarrow & s & (recz) \\
\text{rec } z f (S n) & \rightarrow & f n (\text{rec } z f n) & (recs) \\
\text{rec } z f (\text{raise } \varepsilon) & \rightarrow & \text{raise } \varepsilon & (recr)
\end{array}$$

Réduction

$$\begin{array}{ll}
\frac{t \rightarrow t'}{t \rightarrow t'} (nred) & \frac{t \rightarrow t'}{\lambda x. t \rightarrow \lambda x. t'} (red_\lambda) \\
\frac{t \rightarrow t'}{t u \rightarrow t' u} (red_{app1}) & \frac{u \rightarrow u'}{t u \rightarrow t u'} (red_{app2}) \\
\frac{t \rightarrow t'}{\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow \text{try } t' \text{ with } \varepsilon \mapsto u} (red_{try1}) & \\
\frac{u \rightarrow u'}{\text{try } t \text{ with } \varepsilon \mapsto u \rightarrow \text{try } t \text{ with } \varepsilon \mapsto u'} (red_{try2}) &
\end{array}$$

FIG. 7.1 – Rappel des termes et réductions de λ_ε réutilisées par F_ε .

Définition 7.3 (Types de F_ϵ)

Un type de F_ϵ est définie comme une production de la grammaire suivante :

$A ::=$	α	(variables de type)
	nat	(entiers naturel)
	$A \rightarrow A$	(fonctions)
	$\forall \alpha. A$	(quantification universelle)
	$A \uplus \Delta$	(union)
	A^Δ	(corruption)

Dans les constructions $A \uplus \Delta$ et A^Δ , Δ est un ensemble fini de noms d'exceptions (et donc $\Delta \subseteq \mathcal{E}$). Les précédences sont, de la plus grande à la plus petite : les constructions $A \uplus \Delta$ et A^Δ , la quantification universelle, le type des flèches. Ainsi, $A \rightarrow B^\Delta$ doit se comprendre comme $A \rightarrow (B^\Delta)$, $\forall \alpha. A^\Delta$ comme $\forall \alpha. (A^\Delta)$ et $\forall \alpha. A \rightarrow B$ comme $(\forall \alpha. A) \rightarrow B$. De plus la flèche est associative à droite : le type $A \rightarrow B \rightarrow C$ doit se comprendre $A \rightarrow (B \rightarrow C)$.

L'ensemble de toutes les variables de types est noté $\mathcal{A}$. La construction $\forall \alpha. A$ lie la variable de type α dans le type A . Les notions de variables de type *libres* et *liées* sont définies comme habituellement, de même que la *substitution* dans un type A d'une variable de type α par un type B , notée $A\{\alpha := B\}$. L'ensemble des variables de type libre d'un type A est noté $\text{FV}(A)$ et un type est dit *clos* si il ne possède aucune variable libre ($\text{FV}(A) = \emptyset$). Enfin, les types seront toujours considérés modulo α -équivalence. La substitution des variables de type vérifie la propriété suivante :

Lemme 7.4 Si A, B et C sont des types quelconques et α et β sont deux variables de type tels que $\alpha \neq \beta$ et $\alpha \notin \text{FV}(B)$, alors on a

$$C\{\alpha := A\}\{\beta := B\} = C\{\beta := B\}\{\alpha := A\{\beta := B\}\}$$

Preuve :

La preuve de ce lemme est classique, les nouvelles constructions $A \uplus \Delta$ et A^Δ sont transparentes pour la substitution.

Typage**Définition 7.5 (Environnement de typage)**

Un environnement de typage Γ est un ensemble fini de déclarations ayant la forme

$$\Gamma \equiv x_1 : A_1, \dots, x_n : A_n,$$

où $x_1, \dots, x_n$ sont des variables de termes deux à deux distinctes et $A_1, \dots, A_n$ sont des types. L'ensemble des variables libres d'un environnement Γ , noté $\text{FV}(\Gamma)$, est l'union des ensembles de variables de type libres des types contenus dans Γ :

$$\text{FV}(x_1 : A_1, \dots, x_n : A_n) = \bigcup_{i \in \{1 \dots n\}} \text{FV}(A_i)$$

Définition 7.6 (Jugement de typage)

Le système de type de λ_ϵ est défini par le jugement de typage

$$\Gamma \vdash t : A$$

qui se lit : « Dans l'environnement de typage Γ , le terme t à le type A ». Ce jugement est défini inductivement par les règles de déduction de la figure 7.2.

Règles de typage du système $F\eta$

$$\begin{array}{c} \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \text{ (t-ax)} \\[10pt] \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} \text{ (t-abs)} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B} \text{ (t-app)} \\[10pt] \frac{\Gamma \vdash t : A \quad \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash t : \forall \alpha. A} \text{ (t-gen)} \\[10pt] \frac{\Gamma \vdash t : A \quad A \leq B}{\Gamma \vdash t : B} \text{ (t-subtyp)} \end{array}$$

Règles de typage des entiers naturels

$$\begin{array}{c} \frac{}{\Gamma \vdash 0 : \text{nat}} \text{ (t-zero)} \quad \frac{}{\Gamma \vdash S : \text{nat} \rightarrow \text{nat}} \text{ (t-succ)} \\[10pt] \frac{}{\Gamma \vdash \text{rec} : \forall \alpha. \alpha \multimap \Delta \rightarrow (\text{nat}^\Delta \rightarrow \alpha \multimap \Delta \rightarrow \alpha \multimap \Delta) \rightarrow \text{nat}^\Delta \multimap \Delta' \rightarrow \alpha \multimap (\Delta \cup \Delta')} \text{ (t-rec)} \end{array}$$

Règles spécifiques au typage des exceptions

$$\frac{}{\Gamma \vdash \text{raise } \epsilon : \forall \alpha. \alpha \multimap \{\epsilon\}} \text{ (t-raise)} \quad \frac{\Gamma \vdash t : A \multimap \{\epsilon\} \quad \Gamma \vdash u : A}{\Gamma \vdash \text{try } t \text{ with } \epsilon \mapsto u : A} \text{ (t-try)}$$

FIG. 7.2 – Règles de typage du langage F_ϵ .

Remarquons tout d'abord que les règles du récurseur et de `raise` utilisent le polymorphisme du calcul. Ainsi, la règle $(t\text{-raise})$ n'est plus, comme dans λ_ϵ , un schéma de règles de typage, et la règle $(t\text{-rec})$ n'est un schéma que pour les ensembles de noms d'exceptions Δ et Δ' . La règle $(t\text{-gen})$ est la règle habituel de généralisation du système F . Les autres règles sont toutes reprises sans modification du système de types de λ_ϵ .

Sous-typage

Définition 7.7 (Sous-typage de F_ϵ)

La relation de sous-typage entre deux types A et B est notée $A \leq B$. À partir de cette relation, on définit la notion d'égalité de sous-typage $A \doteq B$ définie comme la conjonction des deux règles $A \leq B$ et $B \leq A$. La relation de sous-typage est définie comme la plus petite relation dérivable à partir des règles de la figure 7.3. Dans cette figure, le sens de la notation $A \doteq B$ dépend de sa position :

- En position de prémisses, elle figure pour deux prémisses :

$$\frac{A \doteq B}{C} \quad \text{est une notation pour} \quad \frac{A \leq B \quad B \leq A}{C}$$

- En conclusion, elle exprime deux règles :

$$\frac{C}{A \doteq B} \quad \text{est une notation pour} \quad \frac{C}{A \leq B} \quad \text{et} \quad \frac{C}{B \leq A}$$

Toutes les règles de sous-typage de λ_ϵ sont également des règles de sous-typage de F_ϵ . Sont ajoutées en plus les règles propres au système F_η gérant la quantification universelle : les règles (*st-gen*), (*st-inst*) et (*st-distr*). Enfin, on ajoute également les règles (*st-uni-fall*) et (*st-corr-fall*). Ces règles expriment la commutation de la quantification universelle avec l'union et la corruption (les lemmes 7.9 et 7.10 montrent d'ailleurs que l'on a commutation dans les deux sens). Notons que cette commutation est toujours valide car les ensembles de noms d'exception ne peuvent mentionner les variables de types.

Propriétés dérivables. Parce que l'ensemble des règles de sous-typage de F_ϵ est un sur-ensemble de celles de λ_ϵ , en particulier les lemmes 4.9, 4.11 et 4.12 de λ_ϵ sont également valides pour F_ϵ . De plus, le sous-typage est stable par corruption :

Lemme 7.8 Si A et B sont des types tels que $A \leq B$, alors quel que soit l'ensemble de noms d'exceptions Δ , on a

$$A^\Delta \leq B^\Delta$$

Preuve :

Comme pour λ_ϵ (lemme 4.10) la preuve est une induction sur la dérivation de $A \leq B$ dont tous les cas se résolvent simplement. En particulier pour les nouvelles règles, on utilise le fait que la corruption commute avec la quantification universelle (voir en particulier les deux lemmes qui suivent).

D'autre part, les règles (*st-uni-fall*) et (*st-corr-fall*) doivent être comprises comme des égalités de sous-typage au vu des deux lemmes qui suivent :

Règles standard de sous-typage

$$\frac{}{A \leq A} \text{ (st-id)} \qquad \frac{A \leq B \quad B \leq C}{A \leq C} \text{ (st-trans)}$$

$$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'} \text{ (st-arrow)}$$

Règles spécifiques au système $F\eta$

$$\frac{A \leq B \quad \alpha \notin \text{FV}(A)}{A \leq \forall \alpha. B} \text{ (st-gen)} \qquad \frac{}{\forall \alpha. A \leq A\{\alpha := B\}} \text{ (st-inst)}$$

$$\frac{\alpha \notin \text{FV}(A)}{\forall \alpha. (A \rightarrow B) \leq A \rightarrow \forall \alpha. B} \text{ (st-distr)}$$

Règles de simplification

$$\frac{}{A^\emptyset \leq A} \text{ (st-cor-noex)}$$

$$\frac{}{(A \uplus \Delta) \uplus \Delta' \doteq A \uplus (\Delta \cup \Delta')} \text{ (st-uni-merge)}$$

$$\frac{}{(A^\Delta)^{\Delta'} \doteq A^{(\Delta \cup \Delta')}} \text{ (st-cor-merge)}$$

Sous-typage des exceptions

$$\frac{}{(A \rightarrow B) \uplus \Delta \leq A \rightarrow B \uplus \Delta} \text{ (st-uni-arrow)} \qquad \frac{A \leq B}{A \uplus \Delta \leq B \uplus \Delta} \text{ (st-uni-ctx)}$$

$$\frac{}{A \leq A \uplus \Delta} \text{ (st-uni)} \qquad \frac{}{A \uplus \Delta \leq A^\Delta} \text{ (st-cor)}$$

$$\frac{}{(A \uplus \Delta)^{\Delta'} \doteq A^{\Delta'} \uplus \Delta} \text{ (st-uni-cor)}$$

$$\frac{}{\forall \alpha. (A \uplus \Delta) \leq (\forall \alpha. A) \uplus \Delta} \text{ (st-uni-fall)} \qquad \frac{}{\forall \alpha. (A^\Delta) \leq (\forall \alpha. A)^\Delta} \text{ (st-cor-fall)}$$

$$\frac{}{(A \rightarrow B)^\Delta \doteq A^\Delta \rightarrow B^\Delta} \text{ (st-cor-arrow)}$$

FIG. 7.3 – Règles de sous-typage du langage F_ϵ .

Lemme 7.9 Si A est un type, α une variable de type et Δ un ensemble de noms d'exceptions, alors on a

$$(\forall \alpha. A) \uplus \Delta \leq \forall \alpha. (A \uplus \Delta)$$

Preuve :

Puisque $\forall \alpha. A \leq A$ (*st-inst*) et $A \leq A \uplus \Delta$ (*st-uni*), $\forall \alpha. A \leq A \uplus \Delta$. Donc $(\forall \alpha. A) \uplus \Delta \leq (A \uplus \Delta) \uplus \Delta$ (*st-uni-ctx*). Mais $(A \uplus \Delta) \uplus \Delta \doteq A \uplus \Delta$ et en utilisant (*st-gen*) avec $(\forall \alpha. A) \uplus \Delta \leq A \uplus \Delta$, on obtient bien que $(\forall \alpha. A) \uplus \Delta \leq \forall \alpha. (A \uplus \Delta)$.

Lemme 7.10 Si A est un type, α une variable de type et Δ un ensemble de noms d'exceptions, alors on a

$$(\forall \alpha. A)^\Delta \leq \forall \alpha. A^\Delta$$

Preuve :

Puisque $\forall \alpha. A \leq A$ (*st-inst*) et $A \leq A^\Delta$ (*st-cor*), $\forall \alpha. A \leq A^\Delta$. Donc $(\forall \alpha. A)^\Delta \leq (A^\Delta)^\Delta$ (lemme 7.8). Mais $(A^\Delta) \uplus \Delta \doteq A^\Delta$ et en utilisant (*st-gen*) avec $(\forall \alpha. A)^\Delta \leq A^\Delta$, on obtient bien que $(\forall \alpha. A)^\Delta \leq \forall \alpha. (A^\Delta)$.

7.2.3 Propriétés

Préservation du typage par réduction et ses conséquences ?

La préservation du typage par réduction en présence de sous-typage et de polymorphisme implicite est difficile à obtenir. En particulier, les preuves indirectes de préservation du typage par réduction pour le système $F\eta$ utilisant le fait que le système $F\eta$ est la clôture par η -expansion du système F et que le système F est stable par β -réduction ne sont pas adaptables facilement à nos extensions.

Nous n'avons pas montré la propriété de préservation du typage par réduction pour F_ϵ mais la conjecturons néanmoins :

Conjecture (Préservation du typage par réduction pour F_ϵ) Si t et t' sont deux termes, A un type et Γ un environnement de typage tel que $\Gamma \vdash t : A$ et $t \rightarrow t'$, alors

$$\Gamma \vdash t' : A$$

Par ailleurs, certaines des propriétés du calcul induites par la préservation du typage seront des conséquences de la correction du typage vis-à-vis de l'interprétation du modèle de réalisabilité présenté en section 7.3 (par exemple, la sûreté du typage pour les entiers naturels).

Toutefois, nous allons prouver les lemmes d'inversion du typage pour l'abstraction, l'application et la construction `try` car ceux-ci ne posent pas de difficultés particulières et vont nous être utiles pour la preuve du théorème de corruption qui suit (théorème 7.15).

Pour le premier de ces lemmes d'inversion, celui de l'abstraction, on introduit la notation suivante :

Notation

On désignera par $\vec{\alpha}$ un vecteur de variables de type (i.e. $\vec{\alpha} = \alpha_1 \dots \alpha_n$ pour un certain n et certaines variables de type α_i). On utilisera alors la notation $\forall \vec{\alpha}. A$ pour désigner le type

$$\forall \alpha_1. \dots \forall \alpha_n. A$$

Lemme 7.11 (Inversion de l'abstraction) *Si t est un terme, C un type et Γ un environnement de typage tel que $\Gamma \vdash \lambda x. t : C$, alors il existe des types A et B et des variables de type $\vec{\alpha} \notin \text{FV}(\Gamma)$ tels que*

$$\begin{aligned} \Gamma, x : A &\vdash t : B \\ \forall \vec{\alpha}. (A \rightarrow B) &\leq C \end{aligned}$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash \lambda x. t : C$. Les seules règles qui ont pu être appliquées sont :

(*t-abs*) Alors il existe des types A et B tel que $C \equiv A \rightarrow B$ et $\Gamma, x : A \vdash t : B$, et donc en particulier $C \leq A \rightarrow B$.

(*t-gen*) On a $\Gamma \vdash \lambda x. t : \forall \alpha_0. C$ car $\Gamma \vdash \lambda x. t : C$ et $\alpha_0 \notin \text{FV}(\Gamma)$. Par hypothèse d'induction, il existe des types A et B et des variables $\vec{\alpha}$ tels que $\Gamma, x : A \vdash t : B$ et $\forall \vec{\alpha}. (A \rightarrow B) \leq C$. Et $\forall \alpha_0. \forall \vec{\alpha}. (A \rightarrow B) \leq \forall \vec{\alpha}. (A \rightarrow B)$ (*st-inst*) donc $\forall \alpha_0. \forall \vec{\alpha}. (A \rightarrow B) \leq C$. Et puisque $\alpha_0 \notin \text{FV}(\forall \alpha_0. \forall \vec{\alpha}. (A \rightarrow B))$, avec (*st-gen*) on a finalement $\forall \alpha_0. \forall \vec{\alpha}. (A \rightarrow B) \leq \forall \alpha_0. C$.

(*t-subtyp*) On a $\Gamma \vdash \lambda x. t : D$ et $D \leq C$. Par hypothèse d'induction, il existe des types A et B et des variables $\vec{\alpha}$ tels que $\Gamma, x : A \vdash t : B$ et $\forall \vec{\alpha}. (A \rightarrow B) D \leq$. Mais alors on a $\forall \vec{\alpha}. (A \rightarrow B) \leq C$.

Lemme 7.12 (Inversion de l'application) *Si t et u sont deux termes, B un type et Γ un environnement de typage tel que $\Gamma \vdash t u : B$, alors il existe un type A tel que*

$$\begin{aligned} \Gamma &\vdash t : A \rightarrow B \\ \Gamma &\vdash u : A \end{aligned}$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash t u : B$. Les seules règles qui ont pu être appliquées sont :

(*t-app*) On a directement le résultat.

- (*t-gen*) On a $\Gamma \vdash t u : \forall \alpha. B$ car $\Gamma \vdash t u : B$ et $\alpha \notin \text{FV}(\Gamma)$. Par hypothèse d'induction on a qu'il existe A tel que $\Gamma \vdash t : A \rightarrow B$ et $\Gamma \vdash u : A$. Mais comme $\alpha \notin \text{FV}(\Gamma)$, $\Gamma \vdash t : \forall \alpha. (A \rightarrow B)$ et $\Gamma \vdash u : \forall \alpha. A$. Le lemme 7.2 nous donne alors $\forall \alpha. (A \rightarrow B) \leq (\forall \alpha. A) \rightarrow (\forall \alpha. B)$ et nous permet de conclure.
- (*t-subtyp*) On a donc que $\Gamma \vdash t u : C$ avec $C \leq B$. Alors, par hypothèse d'induction, il existe un type A tel que $\Gamma \vdash t : A \rightarrow C$ et $\Gamma \vdash u : A$. Mais comme $A \rightarrow C \leq A \rightarrow B$ (grâce à la règle (*st-arrow*)), on a $\Gamma \vdash t : A \rightarrow B$.

Lemme 7.13 (Inversion du try) Si t et u sont deux termes, B un type, Γ un environnement de typage et ε un nom d'exception tel que $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$, alors

$$\begin{array}{c} \Gamma \vdash t : A \uplus \{\varepsilon\} \\ \Gamma \vdash u : A \end{array}$$

Preuve :

On procède par induction sur la dérivation du jugement $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$. Les seules règles qui ont pu être appliquées sont :

(*t-try*) On a directement le résultat.

(*t-gen*) On a $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : \forall \alpha. A$ car $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : A$ et $\alpha \notin \text{FV}(\Gamma)$. Par hypothèse d'induction, $\Gamma \vdash t : A \uplus \{\varepsilon\}$ et $\Gamma \vdash u : A$. Mais puisque $\alpha \notin \text{FV}(\Gamma)$, avec la règle (*t-gen*), on obtient $\Gamma \vdash t : \forall \alpha. A \uplus \{\varepsilon\} \doteq (\forall \alpha. A) \uplus \{\varepsilon\}$ et $\Gamma \vdash u : \forall \alpha. A$.

(*t-subtyp*) On a $\Gamma \vdash \text{try } t \text{ with } \varepsilon \mapsto u : B$ avec $B \leq A$. Alors, par hypothèse d'induction, $\Gamma \vdash t : B \uplus \{\varepsilon\}$ et $\Gamma \vdash u : B$. Et donc, $\Gamma \vdash u : A$ et puisque $B \uplus \{\varepsilon\} \leq A \uplus \{\varepsilon\}$ (grâce à la règle (*st-uni-ctx*)), $\Gamma \vdash t : A \uplus \{\varepsilon\}$.

Corruption des termes

Comme pour λ_ε , on montre formellement que F_ε respecte le principe selon lequel on peut toujours remplacer un sous-terme d'un terme de F_ε par une exception. Pour cela, on rappelle la définition de la *relation de corruption* entre termes (qui est celle de λ_ε puisque les termes de λ_ε et de F_ε sont les mêmes) :

Définition 7.14 (Rappel de la relation de corruption)

Si t et u sont deux termes, la relation $t \sqsubseteq_\Delta u$ qui se lit « t est davantage corrompu par les exceptions de Δ que u » est définie comme la plus petite relation dérivable avec les règles d'inférence de la figure 7.4.

On montre alors le théorème suivant :

Théorème 7.15 (Corruption) Si t et u sont deux termes, A un type et Δ un ensemble de noms d'exceptions tel que $\Gamma \vdash t : A$ et $t \sqsubseteq_\Delta u$, alors

$$\Gamma \vdash u : A^\Delta$$

$$\begin{array}{c}
\frac{}{t \sqsubseteq_{\Delta} t} (c-id) \quad \frac{\varepsilon \in \Delta}{t \sqsubseteq_{\Delta} \text{raise } \varepsilon} (c-rai) \quad \frac{t \sqsubseteq_{\Delta} t'}{\lambda x. t \sqsubseteq_{\Delta} \lambda x. t'} (c-lam) \\
\\
\frac{\frac{t \sqsubseteq_{\Delta} t' \quad u \sqsubseteq_{\Delta} u'}{t u \sqsubseteq_{\Delta} t' u'} (c-app)}{\frac{t \sqsubseteq_{\Delta} t' \quad u \sqsubseteq_{\Delta} u'}{\text{try } t \text{ with } \varepsilon \mapsto u \sqsubseteq_{\Delta} \text{try } t' \text{ with } \varepsilon \mapsto u'} (c-try)}
\end{array}$$

FIG. 7.4 – Rappel de la figure 4.7 définissant la relation de corruption

Preuve :

La preuve procède par induction sur la dérivation du jugement $t \sqsubseteq_{\Delta} u$. On reprend la preuve du théorème de corruption pour λ_{ε} (théorème 4.50) pour presque tous les cas et en particulier pour $(c-app)$ et $(c-try)$ puisque les lemmes d'inversion pour F_{ε} de l'application (lemme 7.12) et du try (lemme 7.13) sont semblables à ceux de λ_{ε} . Seul le cas de $(c-lam)$ diffère puisque le lemme d'inversion de l'abstraction (lemme 7.11) pour F_{ε} diffère de celui pour λ_{ε} .

Dans le cas de $(c-lam)$, on a $\Gamma \vdash \lambda x. t : A$ et $t \sqsubseteq_{\Delta} t'$. En utilisant le lemme d'inversion de l'abstraction (lemme 7.11), il existe des types B et C et des variables de types $\vec{\alpha} \notin \text{FV}(\Gamma)$ tel que $\forall \vec{\alpha}. (B \rightarrow C) \leq A$ et $\Gamma, x : B \vdash t : C$. Donc, par hypothèse d'induction, $\Gamma, x : B \vdash t' : C^{\Delta}$ et alors $\Gamma \vdash \lambda x. t' : B \rightarrow C^{\Delta}$. Mais comme

$$B \rightarrow C^{\Delta} \leq (B \rightarrow C^{\Delta})^{\Delta} \doteq B^{\Delta} \rightarrow C^{\Delta} \doteq (B \rightarrow C)^{\Delta},$$

$\Gamma \vdash \lambda x. t' : B \rightarrow C^{\Delta}$. Puisque $\vec{\alpha} \notin \text{FV}(\Gamma)$, en appliquant tant de fois que nécessaire $(t\text{-gen})$ on obtient $\Gamma \vdash \lambda x. t' : \forall \vec{\alpha}. (B \rightarrow C^{\Delta})$ et puisque $\forall \vec{\alpha}. (B \rightarrow C^{\Delta}) \doteq (\forall \vec{\alpha}. (arrBC))^{\Delta}$, en utilisant le fait que le sous-typage est stable par corruption (lemme 7.8), on a finalement $\Gamma \vdash \lambda x. t' : A^{\Delta}$.

7.3 Un Modèle de réalisabilité pour F_{ε}

Cette section présente un modèle de réalisabilité pour le calcul F_{ε} . Comme pour le modèle de λ_{ε} (section 5.2), les termes seront interprétés par eux-mêmes et les types seront interprétés par des ensembles de termes. Le modèle de F_{ε} ne peut cependant pas être une simple extension de celui de λ_{ε} pour deux raisons :

1. L'interprétation pour λ_{ε} est paramétrée par un ensemble de noms d'exceptions (de façon à gérer le type de la corruption). Cette approche ne permet pas de traiter le cas des variables de type. L'interprétation du type de la corruption du modèle de F_{ε} sera « directe ».
2. La sémantique usuelle de la quantification universelle est celle d'une intersection. C'est à dire qu'intuitivement,

$$\llbracket \forall \alpha. A \rrbracket = \bigcap_{T \in \mathfrak{T}} \llbracket A\{\alpha := T\} \rrbracket$$

Il n'est cependant pas évident de définir l'ensemble $\mathfrak{T}$ sur lequel porte cette intersection. Intuitivement, le « bon » choix est l'ensemble des ensembles cibles de l'interprétation, mais c'est justement cet ensemble qui est en train d'être défini.

Ce problème de circularité (imprédicativité) est bien connu et possède une solution standard : la définition d'une notion de candidat de réalisabilité. Une façon élégante et générale pour définir ces candidats est l'utilisation de techniques d'orthogonalité et le modèle de F_ε va utiliser ces techniques. De plus, l'orthogonalité va nous permettre de définir l'interprétation de la corruption de façon particulièrement simple.

Nous allons d'abord rappeler le principe de l'orthogonalité, puis définir les différentes notions (*daimon*, contextes, ...) préliminaires à la définition du modèle.

7.3.1 L'orthogonalité

Une des premières présentation des structures d'orthogonalité est due à G. Birkhoff [Bir67], sous le nom de *polarity structures*. Une telle structure est définie comme suit :

Définition 7.16 (Structure d'orthogonalité)

Une structure d'orthogonalité est la donnée d'un triplet $(\mathcal{A}^+, \mathcal{A}^-, \perp)$ où $\mathcal{A}^+$ et $\mathcal{A}^-$ sont deux ensembles et $\perp$ est une relation binaire sur $\mathcal{A}^+ \times \mathcal{A}^-$ appelée la relation d'orthogonalité. Si $x \in \mathcal{A}^+$ et $y \in \mathcal{A}^-$, on dira que x est orthogonal à y (respectivement, que y est orthogonal à x) si $x \perp y$.

Alors,

- à tout sous-ensemble X de $\mathcal{A}^+$, on associe l'ensemble $X^\perp$, nommé orthogonal de X , qui est le sous-ensemble de $\mathcal{A}^-$ défini par :

$$X^\perp = \{ y \mid \forall x \in X, x \perp y \}$$

- à tout sous-ensemble Y de $\mathcal{A}^-$, on associe l'ensemble $Y^\perp$, nommé orthogonal de Y , qui est le sous-ensemble de $\mathcal{A}^+$ défini par :

$$Y^\perp = \{ x \mid \forall y \in Y, x \perp y \}$$

Ces structures vérifient les propriétés suivantes :

Lemme 7.17 Si $(\mathcal{A}^+, \mathcal{A}^-, \perp)$ est une structure d'orthogonalité, si S et T sont deux sous-ensembles de $\mathcal{A}^+$ (respectivement, deux sous-ensembles de $\mathcal{A}^-$) et si I est un ensemble quelconque et $(S_i)_{i \in I}$ une famille de sous-ensembles de $\mathcal{A}^+$ (respectivement, une famille de sous-ensembles de $\mathcal{A}^-$), alors :

$$S \subseteq S^{\perp\perp} \tag{1}$$

$$S \subseteq T \Rightarrow T^\perp \subseteq S^\perp \tag{2}$$

$$S^\perp = S^{\perp\perp\perp} \tag{3}$$

$$\left(\bigcup_{i \in I} S_i \right)^\perp = \bigcap_{i \in I} S_i^\perp \tag{4}$$

Preuve :

Les définitions étant symétriques, la preuve pour le cas où S et T sont des sous-ensembles de $\mathcal{A}^+$ et celle pour le cas où ils sont des sous-ensembles de $\mathcal{A}^-$ sont également symétriques, nous ne donnerons donc que le cas de $\mathcal{A}^+$:

- (1) Soit $s \in S$, on veut montrer que $s \in S^{\perp\perp} = \{x \mid \forall y \in S^\perp, x \perp y\}$. Soit $y \in S^\perp$, par définition $s \perp y$ et donc on a bien $s \in S^{\perp\perp}$.
- (2) Supposons $S \subseteq T$, on va montrer que $T^\perp \subseteq S^\perp$. Soit $t \in T^\perp = \{y \mid \forall x \in T, x \perp y\}$. Soit $x \in S$, on a donc $x \in T$ et donc $x \perp t$. Finalement on a donc bien $t \in S^\perp$.
- (3) On montre les deux inclusions suivantes :
 - $S^\perp \subseteq S^{\perp\perp\perp}$: C'est un cas particulier de (1) (en utilisant $A^\perp$).
 - $S^{\perp\perp\perp} \subseteq S^\perp$: Avec (1), $S \subseteq S^{\perp\perp}$ et donc avec (2), $S^{\perp\perp\perp} \subseteq S^\perp$.
- (4) On montre les deux inclusions suivantes :
 - $(\bigcup_{i \in I} S_i)^\perp \subseteq \bigcap_{i \in I} S_i^\perp$: Soit $i \in I$, comme $S_i \subseteq \bigcup_{i \in I} S_i$, avec (2), $(\bigcup_{i \in I} S_i)^\perp \subseteq S_i^\perp$. Et donc, on a $(\bigcup_{i \in I} S_i)^\perp \subseteq \bigcap_{i \in I} S_i^\perp$.
 - $\bigcap_{i \in I} S_i^\perp \subseteq (\bigcup_{i \in I} S_i)^\perp$: Soit $i \in I$, comme $\bigcap_{i \in I} S_i^\perp \subseteq S_i^\perp$, avec (2), $S_i^{\perp\perp} \subseteq (\bigcap_{i \in I} S_i^\perp)^\perp$. Comme de plus $S_i \subseteq S_i^{\perp\perp}$ avec (1), on a finalement $\bigcup_{i \in I} S_i \subseteq (\bigcap_{i \in I} S_i^\perp)^\perp$. Alors avec (2), $(\bigcap_{i \in I} S_i^\perp)^{\perp\perp} \subseteq (\bigcup_{i \in I} S_i)^\perp$ et on conclut en appliquant (1).

L'idée d'orthogonalité est au cœur de plusieurs travaux de sémantique des langages de programmation [Par93, Kri03, VM04].

7.3.2 Définitions préliminaires

Le *daimon*

On ajoute à la syntaxe de F_ε un nouveau terme, appelé le *daimon* et dénoté par $\blacklozenge$ (La notation et la terminologie viennent de [Gir01]).

Le *daimon* est un terme qui « absorbe » son contexte d'évaluation. Dans le cadre de λ_ε , il doit être compris comme une exception non rattrapable. Ses règles de réduction sont données à la figure 7.5.

$\blacklozenge u$	$\rightarrow$	$\blacklozenge$		
$\text{try } \blacklozenge \text{ with } u \mapsto$	$\rightarrow$	$\blacklozenge$		<i>(appdaim)</i>
$\text{rec } z \text{ f } \blacklozenge$	$\rightarrow$	$\blacklozenge$		<i>(recdaim)</i>

FIG. 7.5 – Règles de réduction du *daimon*.

Le *daimon* n'a aucune règle de typage et ne peut apparaître en cours de réduction. Un terme bien typée de F_ε ne peut donc pas le mentionner. En ce sens, L'addition du *daimon* est purement technique et ne change en aucun cas le calcul.

De plus, on peut toujours voir le *daimon* comme une exception dont le nom, privé, n'est jamais utilisé dans un `try`. En conséquence de quoi, toutes les propriétés des réductions de F_ε (confluence, standardisation, ...) restent valables avec l'ajout du *daimon*.

Contextes

Définition 7.18 (Contexte)

Un contexte de F_ε est une production de la grammaire suivante :

$$C ::= [] \mid C u \mid \text{try } C \text{ with } \varepsilon \mapsto \boxtimes \mid \text{rec } z \text{ f } C$$

Le symbole $[]$ est nommé le trou du contexte et le terme obtenu en remplissant le trou du contexte C par le terme t est noté $C[t]$. L'ensemble de tous les contextes est noté $\mathcal{C}$.

Notons que cette notion de contexte est plus restrictive que la notion habituelle où un contexte est n'importe quel terme avec un trou.

On peut remarquer que si on met de côté la restriction au *daimon* dans les gestionnaires des contextes de la forme `try`, ces contextes sont les contextes d'évaluation pour la réduction en appel par nom.

Par ailleurs, forcer cette restriction au *daimon* dans les gestionnaires des contextes de la forme `try` permet d'avoir la propriété suivante :

Fait 7.19 Si C est un contexte et ε un nom d'exception, alors

$$C[\text{raise } \varepsilon] \rightarrow^* \boxtimes \quad \text{ou} \quad C[\text{raise } \varepsilon] \rightarrow^* \text{raise } \varepsilon.$$

Nous verrons dans la section 7.3.3 que cette propriété permet de simplifier l'interprétation du type de la corruption.

L'ensemble de tous les noms d'exceptions $\mathcal{E}$ est un ensemble dénombrable. On fixe une bijection ϕ quelconque de $\mathcal{E}$ dans les entiers naturels ($\phi : \mathcal{E} \rightarrow \mathbb{N}$) et on définit la notation suivante :

Notation

Si Δ est un ensemble (fini) de noms d'exceptions, alors la notation

$$\text{try } [] \text{ with } \Delta$$

désigne

– le contexte

$$\text{try } (\dots (\text{try } [] \text{ with } \varepsilon_1 \mapsto \boxtimes) \dots) \text{ with } \varepsilon_n \mapsto \boxtimes$$

si $\Delta \neq \emptyset$ et si $\varepsilon_1, \dots, \varepsilon_n$ sont les éléments de Δ ordonnées selon ϕ (c'est à dire vérifiant $\phi(\varepsilon_1) < \phi(\varepsilon_2) < \dots < \phi(\varepsilon_n)$).

– le contexte vide, $[]$, sinon.

Une propriété des contextes est :

Lemme 7.20 *Si t est un terme et C un contexte tel que $C[t]$ a une valeur, alors t a une valeur.*

Preuve :

On raisonne par induction sur le contexte C :

- Si $C = []$, alors par hypothèse t a une valeur.
- Si $C = C' u$, par hypothèse $C[t] = C'[t] u$ a une valeur. Alors avec le lemme 3.22, $C'[t]$ a une valeur et on conclut en invoquant l'hypothèse d'induction.
- Si $C = \text{try } C' \text{ with } \varepsilon \mapsto \boxtimes$, par hypothèse $C[t] = \text{try } C' \text{ with } \varepsilon \mapsto \boxtimes$ a une valeur. Alors avec le lemme 3.23, $C'[t]$ a une valeur et on conclut en invoquant l'hypothèse d'induction.
- Si $C = \text{rec } x y C'$, par hypothèse $C[t] = \text{rec } x y C'[t]$ a une valeur. Alors avec le lemme 3.24, $C'[t]$ a une valeur et on conclut en invoquant l'hypothèse d'induction.

La relation d'orthogonalité

On va travailler dans la structure d'orthogonalité où $\mathcal{A}^+$ est $\mathcal{T}$ l'ensemble des termes de F_ε et $\mathcal{A}^-$ est $\mathcal{C}$ l'ensemble des contextes. La *relation d'orthogonalité* $\perp$ met donc en relation un terme et un contexte. Elle est définie par :

Définition 7.21 (Relation d'orthogonalité)

Si t est un terme et si C est un contexte, alors $t \perp C$ (et on dira alors que t et C sont orthogonaux) si et seulement si $C[t] \rightarrow^* \boxtimes$.

Dorénavant, on travaillera exclusivement dans la structure d'orthogonalité $(\mathcal{T}, \mathcal{C}, \perp)$. Dans cette structure, l'orthogonal $S^\perp$ d'un ensemble de contextes S est par définition l'ensemble de termes défini par :

$$S^\perp = \{ t \mid \forall C \in S, C[t] \rightarrow^* \boxtimes \}$$

De plus, nous dirons parfois que le contexte C *reconnaît* le terme t lorsque l'on a $t \perp C$ (i.e., si $C[t] \rightarrow^* \boxtimes$).

Opérations sur les ensembles de contextes

On définit les opérations suivantes sur les ensembles de contextes et de termes.

Définition 7.22 (Composition d'ensembles de contextes)

Si S et T sont deux ensembles de contextes, alors la composition de S avec T est définie par

$$S \circ T = \{ C[D []] \mid C \in S, D \in T \}$$

Définition 7.23 (Concaténation d'un ensemble de termes et d'un ensemble de contextes)

Si A est un ensemble de termes et S un ensemble de contextes, alors la concaténation de A avec S est définie par

$$A \cdot S = \{ C[[] u] \mid C \in S, u \in A \}$$

On définit également deux opérations sur les ensembles de contextes qui seront utilisées pour l'interprétation de la corruption et de l'union. Nous reportons la discussion sur l'utilité de ces deux constructions à la présentation du modèle.

Définition 7.24

Si S est un ensemble de contextes et si Δ est un ensemble de noms d'exceptions, alors on définit les deux opérations suivantes :

$$\begin{aligned} \downarrow_{\Delta} S &= S \circ \{ \text{try } [] \text{ with } \Delta \} \\ \uparrow_{\Delta} S &= \{ \text{try } [] \text{ with } \Delta \} \circ S \end{aligned}$$

Notons que par définition on a :

Fait 7.25 Si A est un ensemble de termes, S un ensemble de contextes et Δ et Δ' des ensembles de noms d'exceptions, alors

$$\begin{aligned} \uparrow_{\Delta} (A \cdot S) &= A \cdot \uparrow_{\Delta} S \\ \uparrow_{\Delta} (\downarrow_{\Delta'} S) &= \downarrow_{\Delta'} (\uparrow_{\Delta} S) \end{aligned}$$

En revanche, au vu de la définition de la notation $\text{try } [] \text{ with } \Delta$, il est clair que les contextes $\text{try } (\text{try } [] \text{ with } \Delta') \text{ with } \Delta$ et $\text{try } [] \text{ with } (\Delta \cup \Delta')$ sont différents et donc si S est un ensemble de contextes,

$$\downarrow_{\Delta} (\downarrow_{\Delta'} S) \neq \downarrow_{\Delta \cup \Delta'} S \quad \text{et} \quad \uparrow_{\Delta} (\uparrow_{\Delta'} S) \neq \uparrow_{\Delta \cup \Delta'} S.$$

Il est cependant important de remarquer qu'ils désignent les mêmes ensembles d'orthogonaux, c'est à dire :

Lemme 7.26 Si S est un ensemble de contextes et Δ et Δ' sont deux ensembles de noms d'exceptions, alors

$$\begin{aligned} (\downarrow_{\Delta} (\downarrow_{\Delta'} S))^{\perp} &= (\downarrow_{\Delta \cup \Delta'} S)^{\perp} \\ (\uparrow_{\Delta} (\uparrow_{\Delta'} S))^{\perp} &= (\uparrow_{\Delta \cup \Delta'} S)^{\perp} \end{aligned}$$

Preuve :

On va se contenter de donner la preuve que $(\uparrow_{\Delta} (\uparrow_{\Delta'} S))^{\perp} = (\uparrow_{\Delta \cup \Delta'} S)^{\perp}$ (la preuve que $(\downarrow_{\Delta} (\downarrow_{\Delta'} S))^{\perp} = (\downarrow_{\Delta \cup \Delta'} S)^{\perp}$ étant similaire :

- Montrons que $(\uparrow_{\Delta} (\uparrow_{\Delta'} S))^{\perp} \subseteq (\uparrow_{\Delta \cup \Delta'} S)^{\perp}$: Soit $t \in (\uparrow_{\Delta} (\uparrow_{\Delta'} S))^{\perp}$ et $C \in \uparrow_{\Delta \cup \Delta'} S$, par définition, $C = \text{try } D \text{ with } (\Delta \cup \Delta')$ où $D \in S$. Donc $\text{try } (\text{try } D[t] \text{ with } \Delta') \text{ with } \Delta \rightarrow^* \text{⊥}$ et

- Soit $D[t] \rightarrow^* \boxtimes$, mais alors on a bien $C[t] \rightarrow^* \boxtimes$.
- Soit $D[t] \rightarrow^*$ raise ε pour $\varepsilon \in \Delta'$ ou $\varepsilon \in \Delta$. Mais donc on a bien aussi toujours $C[t] \rightarrow^* \boxtimes$.
- Montrons que $(\uparrow_{\Delta \cup \Delta'} S)^\perp \subseteq (\uparrow_\Delta (\uparrow_{\Delta'} S))^\perp$: Soit $t \in (\uparrow_{\Delta \cup \Delta'} S)^\perp$ et $C \in \uparrow_\Delta (\uparrow_{\Delta'} S)$, par définition, $C = \text{try}(\text{try } D \text{ with } \Delta') \text{ with } \Delta$ où $D \in S$. Donc $\text{try } D[t] \text{ with } (\Delta \cup \Delta') \rightarrow^* \boxtimes$ et
 - Soit $D[t] \rightarrow^* \boxtimes$, mais alors on a bien $C[t] \rightarrow^* \boxtimes$.
 - Soit $D[t] \rightarrow^*$ raise ε pour $\varepsilon \in \Delta \cup \Delta'$. Mais donc on a bien aussi toujours $C[t] \rightarrow^* \boxtimes$.

Remarquons d'ailleurs qu'avec le fait 7.25, ce lemme implique le lemme suivant :

Lemme 7.27 *Si S est un ensemble de contextes, A un ensemble de termes et Δ et Δ' deux ensembles de noms d'exceptions, alors*

$$(A \cdot \uparrow_\Delta (\uparrow_{\Delta'} S))^\perp = (A \cdot \uparrow_{\Delta \cup \Delta'} S)^\perp$$

Preuve :

Ce lemme découle du fait 7.25 et du lemme 7.26 puisque

$$(A \cdot \uparrow_\Delta (\uparrow_{\Delta'} S))^\perp = (\uparrow_\Delta (\uparrow_{\Delta'} (A \cdot S)))^\perp = (\uparrow_{\Delta \cup \Delta'} (A \cdot S))^\perp = (A \cdot \uparrow_{\Delta \cup \Delta'} S)^\perp$$

Ce lemme sera nécessaire pour montrer que le modèle valide l'égalité de sous-typage $(A \rightarrow B)^\Delta \doteq A^\Delta \rightarrow B^\Delta$ (voir lemme 7.41).

7.3.3 Définition et propriétés de l'interprétation

On peut résumer le principe du modèle que nous allons présenter ainsi :

1. À chaque type A , on associe l'ensemble de contextes $|A|_\rho$ qui est nommé sa co-interprétation ;
2. L'interprétation $\llbracket A \rrbracket_\rho$ du type A , qui est un ensemble de termes, est alors uniformément définie comme l'orthogonal de $|A|_\rho$.

Pour gérer l'interprétation des variables de types, les interprétations $|\cdot|_\rho$ et $\llbracket \cdot \rrbracket_\rho$ sont paramétrés par une fonction de valuation ρ :

Définition 7.28

Une fonction de valuation ρ est une fonction associant à chaque variable de type un ensemble de contextes non vide (i.e., $\rho : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{E})^+$).

La restriction sur la valuation d'une variable de type à ne pas être vide assure que l'interprétation d'un type ne soit jamais vide (voir lemme 7.32).

On utilisera la notation suivante pour définir la valuation d'une certaine variable :

Notation

Si ρ est une fonction de valuation, α une variable de type et S un ensemble de contextes, on note $\rho; \alpha \leftarrow S$ la fonction de valuation définie par :

$$\rho; \alpha \leftarrow S = \beta \mapsto \begin{cases} \rho(\alpha) & \text{si } \beta \neq \alpha \\ S & \text{si } \beta = \alpha \end{cases}$$

On donne maintenant les définitions de $|\cdot|_\rho$ et $\llbracket \cdot \rrbracket_\rho$:

Définition 7.29 (Co-interprétation des types)

Si A est un type, alors l'ensemble de contextes $|A|_\rho$ est défini inductivement par :

$$\begin{aligned} |\alpha|_\rho &= \rho(\alpha) \\ |\mathbf{nat}|_\rho &= \{ \mathbf{rec} \ \boxtimes \ (\lambda y. \lambda x. x) \ [] \} \\ |A \uplus \Delta|_\rho &= \downarrow_\Delta |A|_\rho \\ |A^\Delta|_\rho &= \uparrow_\Delta |A|_\rho \\ |A \rightarrow B|_\rho &= \bigcup_{\Delta \subseteq \mathcal{E}} |A^\Delta|_\rho^\perp \cdot |B^\Delta|_\rho \\ |\forall \alpha. A|_\rho &= \bigcup_{S \subseteq \mathcal{P}(\mathcal{E})^+} |A|_{\rho; \alpha \leftarrow S} \end{aligned}$$

Définition 7.30 (Interprétation des types)

Si A est un type, l'ensemble $\llbracket A \rrbracket_\rho$ est défini par :

$$\llbracket A \rrbracket_\rho = |A|_\rho^\perp = \{ t \mid \forall C \in |A|_\rho, C[t] \rightarrow^* \boxtimes \}$$

Premières propriétés

Avant de décrire plus précisément l'interprétation de chaque construction de type, on commence par montrer quelques propriétés générales de cette interprétation. D'abord, on montre que l'interprétation des types est close par β -équivalence :

Lemme 7.31 (Cloture par β -équivalence) Si t et u sont deux termes, A un type et ρ une fonction de valuation tels que $t \in \llbracket A \rrbracket_\rho$ et $t =_\beta u$, alors

$$u \in \llbracket A \rrbracket_\rho.$$

Preuve :

On a $t \in \llbracket A \rrbracket_\rho$ et $t =_\beta u$ et on veut montrer que $u \in \llbracket A \rrbracket_\rho$. Soit $C \in |A|_\rho$, par définition $C[t] \rightarrow^* \boxtimes$. Mais puisque $t =_\beta u$, $C[t] =_\beta C[u]$. Par confluence de la réduction (théorème 3.7), $C[u] \rightarrow^* \boxtimes$ et donc $u \in \llbracket A \rrbracket_\rho$.

En particulier, l'interprétation est donc close par réduction et par anti-réduction.

L'interprétation et la co-interprétation d'un type ne sont jamais vides. Et en particulier, le *daimon* habite l'interprétation de n'importe quel type :

Lemme 7.32 *Si A est un type et ρ une fonction de valuation, alors*

$$\begin{aligned} |A|_\rho &\neq \emptyset \\ \boxtimes &\in \llbracket A \rrbracket_\rho \end{aligned}$$

Preuve :

On raisonne par induction sur A . Notons d'abord que si $|A|_\rho \neq \emptyset$, alors $\boxtimes \in \llbracket A \rrbracket_\rho$ puisque quel que soit le contexte C , $C[\boxtimes] \rightarrow^* \boxtimes$. Dans le cas des variables, on utilise le fait que l'image d'une variable de type par une fonction de valuation n'est jamais vide (et ce, par définition des fonctions de valuation). Le cas de nat est trivial et les cas de $A \uplus \Delta$, A^Δ et $\forall \alpha. A$ sont une conséquence directe de l'hypothèse d'induction. Pour $A \rightarrow B$, on utilise le fait que par hypothèse d'induction $|A|_\rho$ et $|B|_\rho$ sont non vide et donc en particulier $(\uparrow_\Delta |A|_\rho)^\perp$ n'est pas vide (il contient $\boxtimes$).

D'autre part, pour n'importe quel type, les termes habitants son interprétation ont tous une valeur :

Lemme 7.33 *Si t est un terme, A un type et ρ une fonction de valuation telle que $t \in \llbracket A \rrbracket_\rho$, alors t a une valeur.*

Preuve :

Par définition de $t \in \llbracket A \rrbracket_\rho$,

$$t \in \{ t \mid \forall C \in |A|_\rho, C[t] \rightarrow^* \boxtimes \}.$$

Donc, si C est un contexte de $|A|_\rho$, $C[t]$ a une valeur (et cette valeur est $\boxtimes$) et, avec le lemme 7.20, on en déduit que t a une valeur. Il suffit de vérifier que quelque soit A et quelque soit ρ , l'ensemble $|A|_\rho$ n'est pas vide, mais c'est le résultat du lemme 7.32.

Enfin, on montre que l'interprétation du type $\llbracket A^\Delta \rrbracket_\rho$ contient à la fois les termes *raise* ε pour $\varepsilon \in \Delta$ et tous les termes de $\llbracket A \rrbracket_\rho$:

Lemme 7.34 *Si A est un type, ρ une fonction de valuation et Δ un ensemble fini de noms d'exceptions, alors pour tout $\varepsilon \in \Delta$,*

$$\text{raise } \varepsilon \in \llbracket A^\Delta \rrbracket_\rho$$

Preuve :

Soit un contexte $C \in |A^\Delta|_\rho$. Par définition, $C = \{\text{try } [\] \text{ with } \Delta\} \circ D$ où $D \in |A|_\rho$. Mais le fait 7.19 nous dit que soit $D[\text{raise } \varepsilon] \rightarrow^* \bot$, soit $D[\text{raise } \varepsilon] \rightarrow^* \text{raise } \varepsilon$. Or, dans les deux cas on a $C[\text{raise } \varepsilon] \rightarrow^* \bot$.

Lemme 7.35 Si A est un type, ρ une fonction de valuation et Δ un ensemble fini de noms d'exceptions, alors

$$\llbracket A \rrbracket_\rho \subseteq \llbracket A^\Delta \rrbracket_\rho$$

Preuve :

Soit $t \in \llbracket A \rrbracket_\rho$ et soit $C \in |A^\Delta|_\rho$. On veut montrer que $C[t] \rightarrow^* \bot$. Par définition, il existe un contexte $D \in |A|_\rho$ tel que $C = \text{try } [\] \text{ with } \Delta \circ D$. Mais comme $t \in \llbracket A \rrbracket_\rho$, $D[t] \rightarrow^* \bot$ et donc $C[t] = \text{try } D[t] \text{ with } \Delta \rightarrow^* \bot$.

Description de l'interprétation

Les entiers naturels. L'interprétation des entiers naturels est standard. Le contexte

$$\text{rec } \bot (\lambda y. \lambda x. x) [\]$$

reconnaît tous les entiers, mais par définition du *daimon*, il reconnaît également tous les termes de la forme $S^n \bot$. Plus précisément, on prouve le résultat suivant :

Lemme 7.36 Si ρ est une fonction de valuation,

$$\llbracket \text{nat} \rrbracket_\rho = \{ t \mid t \rightarrow^* S^n 0, n \in \mathbb{N} \} \cup \{ t \mid t \rightarrow^* S^n \bot, n \in \mathbb{N} \}.$$

Preuve :

Par définition, $\llbracket \text{nat} \rrbracket_\rho = \{ t \mid \text{rec } \bot (\lambda y. \lambda x. x) t \rightarrow^* \bot \}$. On montre les deux sens de l'égalité voulue séparément :

- Si $t \in \{ t \mid t \rightarrow^* S^n 0, n \in \mathbb{N} \} \cup \{ t \mid t \rightarrow^* S^n \bot, n \in \mathbb{N} \}$, alors si Φ désigne 0 ou $\bot$, il existe $n \in \mathbb{N}$ tel que $t \rightarrow^* S^n \Phi$. On montre alors par induction sur n que $\text{rec } \bot (\lambda y. \lambda x. x) t \rightarrow^* \bot$.
- Si $n = 0$, t se réduit sur 0 ou $\bot$. Dans tous les cas, il est facile de vérifier que l'on a bien $\text{rec } \bot (\lambda y. \lambda x. x) t \rightarrow^* \bot$.
- Sinon, $t \rightarrow^* S(S^n \Phi)$. Alors

$$\begin{aligned} & \text{rec } \bot (\lambda y. \lambda x. x) t \\ & \rightarrow^* \text{rec } \bot (\lambda y. \lambda x. x) (S(S^n \Phi)) \\ & \rightarrow (\lambda y. \lambda x. x) (S^n \Phi) (\text{rec } \bot (\lambda y. \lambda x. x) (S^n \Phi)) \\ & \rightarrow \text{rec } \bot (\lambda y. \lambda x. x) (S^n \Phi) \end{aligned}$$

et on conclut par hypothèse d'induction.

- Si $t \in \llbracket \text{nat} \rrbracket_\rho$, alors $\text{rec } \boxtimes (\lambda y. \lambda x. x) t \rightarrow^* \boxtimes$. En utilisant le lemme de standardisation (lemme 3.21), on a $\text{rec } \boxtimes (\lambda y. \lambda x. x) t \rightarrow_{wh}^* \boxtimes$. Autrement dit, il existe k tel que $\text{rec } \boxtimes (\lambda y. \lambda x. x) t \rightarrow_{wh}^k \boxtimes$. On va alors montrer que quel que soit k et quel que soit t , si $\text{rec } \boxtimes (\lambda y. \lambda x. x) t \rightarrow_{wh}^{k'} \boxtimes$ pour $k' \leq k$, alors $t \rightarrow_{wh}^* S^n \Phi$ où Φ représente 0 ou $\boxtimes$ et ce, par induction sur k :
 - On ne peut avoir $k = 0$, puisque $\text{rec } \boxtimes (\lambda y. \lambda x. x) t$ n'est pas $\boxtimes$.
 - Autrement, $\text{rec } \boxtimes (\lambda y. \lambda x. x) t \rightarrow_{wh} t_0 \rightarrow_{wh}^k \boxtimes$ et par hypothèse d'induction, quel que soit $k' \leq k$ et quel que soit t' , si $\text{rec } \boxtimes (\lambda y. \lambda x. x) t' \rightarrow_{wh}^{k'} \boxtimes$, alors $t \rightarrow_{wh}^* S^n \Phi$. On raisonne par cas sur la règle utilisée pour dériver $\text{rec } \boxtimes (\lambda y. \lambda x. x) t \rightarrow_{wh} t_0$:
 - Soit on a utilisé une des règles (*recz*) ou (*recdaim*) et alors t est respectivement 0 ou $\boxtimes$. Dans tout ces cas on conclut directement.
 - Sinon, la seule autre règle que l'on a pu utiliser est (*recs*). On a alors $t = S t'$ et $t_0 = (\lambda y. \lambda x. x) t' (\text{rec } \boxtimes (\lambda y. \lambda x. x) t')$. Mais de plus $t_0 \rightarrow_{wh}^k \boxtimes$. Il est évident que k ne peut alors pas être 0 et la seule réduction possible est $t_0 \rightarrow_{wh} \text{rec } \boxtimes (\lambda y. \lambda x. x) t' \rightarrow_{wh}^{k-1} \boxtimes$. Par hypothèse d'induction, on a donc $t' = S^n \Phi$ et puisque $t = S t'$, on a bien $t = S^{n+1} \Phi$. On a donc montré que $t \rightarrow_{wh}^* S^n \Phi$. Donc $t \rightarrow^* S^n \Phi$.

De plus, pour prouver que l'interprétation valide le typage, nous utiliserons le lemme suivant qui caractérise les entiers corrompus :

Lemme 7.37 *Si ρ est une fonction de valuation,*

$$\llbracket \text{nat}^\Delta \rrbracket_\rho = \llbracket \text{nat} \rrbracket_\rho \cup \{ t \mid t \rightarrow^* S^n \text{raise } \varepsilon, n \in \mathbb{N}, \varepsilon \in \Delta \}.$$

Preuve :

Par définition, $\llbracket \text{nat}^\Delta \rrbracket_\rho = \{ t \mid \text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow^* \boxtimes \}$. On montre les deux sens de l'égalité voulue séparément :

- Si $t \in \llbracket \text{nat} \rrbracket_\rho \cup \{ t \mid t \rightarrow^* S^n \text{raise } \varepsilon, n \in \mathbb{N}, \varepsilon \in \Delta \}$, puisque $\llbracket \text{nat} \rrbracket_\rho = \{ t \mid t \rightarrow^* S^n 0, n \in \mathbb{N} \} \cup \{ t \mid t \rightarrow^* S^n \boxtimes, n \in \mathbb{N} \}$ (lemme 7.36), alors si Φ désigne 0 ou $\text{raise } \varepsilon$ (pour $\varepsilon \in \Delta$) ou $\boxtimes$, il existe $n \in \mathbb{N}$ tel que $t \rightarrow^* S^n \Phi$. On montre alors par induction sur n que $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow^* \boxtimes$
 - Si $n = 0$, t se réduit sur 0 ou $\boxtimes$ ou $\text{raise } \varepsilon$. Dans tous les cas, il est facile de vérifier que l'on a bien $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow^* \boxtimes$.
 - Sinon, $t \rightarrow^* S (S^n \Phi)$. Alors

$$\begin{aligned} & \text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \\ \rightarrow^* & \text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) (S (S^n \Phi))) \text{ with } \Delta \\ \rightarrow & \text{try}((\lambda y. \lambda x. x) (S^n \Phi) (\text{rec } \boxtimes (\lambda y. \lambda x. x) (S^n \Phi))) \text{ with } \Delta \\ \rightarrow & \text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) (S^n \Phi)) \text{ with } \Delta \end{aligned}$$

et on conclut par hypothèse d'induction.

- Si $t \in \llbracket \text{nat}^\Delta \rrbracket_\rho$, alors $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow^* \boxtimes$. En utilisant le lemme de standardisation (lemme 3.21), on a $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow_{wh}^* \boxtimes$. Autrement dit, il existe k tel que $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow_{wh}^k \boxtimes$. On va alors montrer que quel que soit k et quel que soit t , si $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow_{wh}^{k'} \boxtimes$ pour $k' \leq k$, alors $t \rightarrow_{wh}^* S^n \Phi$ où Φ représente 0 ou $\boxtimes$ ou $\text{raise } \varepsilon$ (pour $\varepsilon \in \Delta$) et ce, par induction sur k :
 - On ne peut avoir $k = 0$, puisque $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta$ n'ait pas $\boxtimes$.
 - Autrement, $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow_{wh} t_0 \rightarrow_{wh}^k \boxtimes$ et par hypothèse d'induction, quel que soit $k' \leq k$ et quel que soit t' , si $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t') \text{ with } \Delta \rightarrow_{wh}^{k'} \boxtimes$

$\boxtimes$, alors $t \rightarrow_{wh}^* S^n \Phi$. On raisonne par cas sur la règle utilisée pour dériver $\text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t) \text{ with } \Delta \rightarrow_{wh} t_0$:

- Soit on a utilisé une des règles (*recz*), (*recr*) ou (*recdaim*) et alors t est respectivement 0, raise ε ou $\boxtimes$. Dans tout ces cas on conclut directement.
- Sinon, la seule autre règle que l'on a pu utiliser est (*recs*). On a alors $t = S t'$ et $t_0 = \text{try}((\lambda y. \lambda x. x) t' (\text{rec } \boxtimes (\lambda y. \lambda x. x) t')) \text{ with } \Delta$. Mais de plus $t_0 \rightarrow_{wh}^k \boxtimes$. Il est évident que k ne peut alors pas être 0 et la seule réduction possible est $t_0 \rightarrow_{wh} \text{try}(\text{rec } \boxtimes (\lambda y. \lambda x. x) t') \text{ with } \Delta \rightarrow_{wh}^{k-1} \boxtimes$. Par hypothèse d'induction, on a donc $t' = S^n \Phi$ et puisque $t = S t'$, on a bien $t = S^{n'} \Phi$.

On a donc montré que $t \rightarrow_{wh}^* S^n \Phi$. Donc $t \rightarrow^* S^n \Phi$.

Le type des fonctions. Dans ce modèle par orthogonalité, la flèche s'interprète de la même manière que dans le modèle décrit à la section 5.1.

Lemme 7.38 Si A et B sont deux types et ρ une fonction de valuation, alors

$$\llbracket A \rightarrow B \rrbracket_\rho = \bigcap_{\Delta \subseteq \mathcal{E}} \{ t \mid \forall u \in \llbracket A^\Delta \rrbracket_\rho, t u \in \llbracket B^\Delta \rrbracket_\rho \}.$$

Preuve :

Par définition, $\llbracket A \rightarrow B \rrbracket_\rho = (\bigcup_{\Delta \subseteq \mathcal{E}} |A^\Delta|_\rho^\perp \cdot |B^\Delta|_\rho)^\perp$. On va montrer les deux sens de l'égalité séparément :

- Soit t tel que $\forall C \in \bigcup_{\Delta \subseteq \mathcal{E}} |A^\Delta|_\rho^\perp \cdot |B^\Delta|_\rho, C[t] \rightarrow^* \boxtimes$. Soit Δ un ensemble de noms d'exceptions, soit $u \in \llbracket A^\Delta \rrbracket_\rho$ et soit $D \in |B^\Delta|_\rho$. Par définition, $D[[] u] \in |A^\Delta|_\rho^\perp \cdot |B^\Delta|_\rho$ et donc $D[t u] \rightarrow^* \boxtimes$. On a donc bien que $t u \in \llbracket B^\Delta \rrbracket_\rho$.
- Soit $t \in \bigcap_{\Delta \subseteq \mathcal{E}} \{ t \mid \forall u \in \llbracket A^\Delta \rrbracket_\rho, t u \in \llbracket B^\Delta \rrbracket_\rho \}$. Soit Δ un ensemble de noms d'exceptions et soit $C \in |A^\Delta|_\rho^\perp \cdot |B^\Delta|_\rho$. Par définition, $C = D[[] u]$ où $D \in |B^\Delta|_\rho$ et $u \in \llbracket A^\Delta \rrbracket_\rho$. Mais donc par hypothèse $t u \in \llbracket B^\Delta \rrbracket_\rho$ et donc $C[t] = D[t u] \rightarrow^* \boxtimes$.

La quantification universelle. L'interprétation de la quantification universelle est une intersection :

Lemme 7.39 Si A est un type et ρ une fonction de valuation, alors

$$\llbracket \forall \alpha. A \rrbracket_\rho = \bigcap_{S \subseteq \mathcal{P}(\mathcal{E})^+} \llbracket A \rrbracket_{\rho; \alpha \leftarrow S}$$

Preuve :

Par définition,

$$\llbracket \forall \alpha. A \rrbracket_\rho = | \forall \alpha. A |_ \rho^\perp = (\bigcup_{S \subseteq \mathcal{P}(\mathcal{E})^+} |A|_{\rho; \alpha \leftarrow S})^\perp$$

et on conclut en utilisant le fait que l'orthogonal d'une union est l'intersection des orthogonaux (lemme 7.17).

Union et corruption. L'interprétation de l'union et de la corruption est conforme aux intuitions que l'on a donné de ces constructions. Les co-interprétations de $A \uplus \Delta$ et A^Δ sont construites à partir de la co-interprétation de A en associant à chaque contexte C de $|A|_\rho$,

- le contexte $C[\text{try } [] \text{ with } \Delta]$ dans $|A \uplus \Delta|_\rho$;
- le contexte $\text{try } C[] \text{ with } \Delta$ dans $|A^\Delta|_\rho$.

Intuitivement, l'idée est que pour un certain terme t , le terme $C[\text{try } t \text{ with } \Delta]$ se réduira sur $\boxtimes$ soit, si t lève une exception de Δ , et sinon, seulement si $t \in |A|_\rho$. Par contraste, $\text{try } C[t] \text{ with } \Delta$ se réduira sur $\boxtimes$ si t est un terme de $|A|_\rho$ où certains sous-termes peuvent avoir été remplacés par une exception de Δ , puisqu'alors $C[t]$ se réduira sur $\text{raise } \varepsilon$ pour $\varepsilon \in \Delta$ (ou sur $\boxtimes$) et donc $\text{try } C[t] \text{ with } \Delta$ se réduira bien sur $\boxtimes$.

Notons que l'interprétation de A^Δ utilise la restriction à $\boxtimes$ dans la définition des contextes de forme try , puisque sans cette restriction, on n'a pas le fait que si t mentionne $\text{raise } \varepsilon$, alors $C[t]$ se réduira soit sur $\text{raise } \varepsilon$, soit sur $\boxtimes$. Sans cette restriction, l'interprétation de A^Δ serait donc plus difficile à définir.

Notons que l'interprétation du type $A \uplus \Delta$ dans F_ε est en fait la même que dans λ_ε :

Lemme 7.40 *Si A est un type, Δ un ensemble de noms d'exceptions et ρ une fonction de valuation, alors*

$$\llbracket A \uplus \Delta \rrbracket_\rho = \llbracket A \rrbracket_\rho \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta \}.$$

Preuve :

Par définition, $\llbracket A \uplus \Delta \rrbracket_\rho = (\downarrow_\Delta |A|_\rho)^\perp = \{ t \mid \forall C \in |A|_\rho, C[\text{try } t \text{ with } \Delta] \rightarrow^* \boxtimes \}$. On montre alors les deux sens de l'égalité voulue séparément :

- Soit $t \in \{ t \mid \forall C \in |A|_\rho, C[\text{try } t \text{ with } \Delta] \rightarrow^* \boxtimes \}$, alors $\text{try } t \text{ with } \Delta \in \llbracket A \rrbracket_\rho$. Alors $\text{try } t \text{ with } \Delta$ est faiblement normalisant (lemme 7.33) et donc t aussi (lemme 7.20) :
- Soit $t \rightarrow^* \text{raise } \varepsilon$ pour $\varepsilon \in \Delta$ et on conclut.
- Soit $t \rightarrow^* v$ où v est une valeur régulière et $\text{try } t \text{ with } \Delta \rightarrow^* v$. Mais comme $\llbracket A \rrbracket_\rho$ est clos par réduction (lemme 7.31), puisque $\text{try } t \text{ with } \Delta \in \llbracket A \rrbracket_\rho$, $v \in \llbracket A \rrbracket_\rho$. Enfin, puisque $\llbracket A \rrbracket_\rho$ est également clos par anti-réduction (lemme 7.31), $t \in \llbracket A \rrbracket_\rho$.
- Soit $t \in \llbracket A \rrbracket_\rho \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta \}$ et soit $C \in |A|_\rho$ (on doit montrer que $C[\text{try } t \text{ with } \Delta] \rightarrow^* \boxtimes$),
 - Si $t \rightarrow^* \text{raise } \varepsilon$ pour $\varepsilon \in \Delta$, alors $\text{try } t \text{ with } \Delta \rightarrow^* \boxtimes$ et donc $C[\text{try } t \text{ with } \Delta] \rightarrow^* \boxtimes$.
 - Sinon $t \in \llbracket A \rrbracket_\rho$, alors $\text{try } t \text{ with } \Delta \in \llbracket A \rrbracket_\rho$ (car t étant faiblement normalisant (lemme 7.33), il existe une valeur v tel que $t \rightarrow^* v$ et donc $\text{try } t \text{ with } \Delta \rightarrow^* v$ et on conclut par clôture par réduction et anti-réduction de l'interprétation (lemmes 7.31 et 7.31)). Finalement, par définition de l'orthogonalité $C[\text{try } t \text{ with } \Delta] \rightarrow^* \boxtimes$.

7.3.4 Correction du sous-typage vis-à-vis du modèle

Nous commençons par montrer que les principales égalités du sous-typage (commutation de la corruption avec les autres constructions de type et union d'ensembles d'excep-

tions) sont validées par le modèle. Précisons que ces égalités seront utilisées dans la suite sans toujours être mentionnées.

Lemme 7.41 *Si A et B sont deux types, ρ une fonction de valuation et Δ un ensemble de noms d'exceptions, alors*

$$\llbracket (A \rightarrow B)^\Delta \rrbracket_\rho = \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$$

Preuve :

En utilisant le lemme 7.17 qui dit que l'orthogonal d'une union est une intersection et en utilisant le fait 7.25 et les lemmes 7.26 et 7.27, on montre que

$$\begin{aligned} \llbracket (A \rightarrow B)^\Delta \rrbracket_\rho &= \bigcap_{\Delta' \subseteq \mathcal{E}} ((\uparrow_{\Delta'} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} | B |_\rho)^\perp \\ \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho &= \bigcap_{\Delta' \subseteq \mathcal{E}} ((\uparrow_{\Delta \cup \Delta'} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} | B |_\rho)^\perp \end{aligned}$$

Il s'ensuit alors directement que $\llbracket (A \rightarrow B)^\Delta \rrbracket_\rho \subseteq \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$. En effet, si $t \in \llbracket (A \rightarrow B)^\Delta \rrbracket_\rho$ alors quel que soit Δ'' , $t \in ((\uparrow_{\Delta''} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta''} | B |_\rho)^\perp$. Donc si Δ' est un ensemble de noms d'exceptions quelconque et si $C \in (\uparrow_{\Delta \cup \Delta'} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} | B |_\rho$, alors $C \in (\uparrow_{\Delta''} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta''} | B |_\rho$ (en prenant $\Delta'' = \Delta \cup \Delta'$) et donc on a bien $C[t] \rightarrow^* \star$.

Mais d'autre part, on montre avec les lemmes 7.45 et 7.35 que quel que soit les ensembles Δ et Δ' , $(\uparrow_{\Delta'} | A |_\rho)^\perp \subseteq (\uparrow_{\Delta \cup \Delta'} | A |_\rho)^\perp$. Donc si $t \in \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$, si Δ' est un ensemble de noms d'exceptions et si $C \in (\uparrow_{\Delta'} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} | B |_\rho$, alors $C \in (\uparrow_{\Delta \cup \Delta'} | A |_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} | B |_\rho$ et on a bien $C[t] \rightarrow^* \star$.

Lemme 7.42 *Si A est un type, ρ une fonction de valuation et Δ un ensemble de noms d'exceptions, alors*

$$|(\forall \alpha. A)^\Delta|_\rho = |\forall \alpha. A^\Delta|_\rho$$

Preuve :

Cela revient à montrer que l'opérateur $\uparrow_\Delta$ commute avec une union ne portant pas sur des noms d'exceptions, ce qui est direct.

Lemme 7.43 *Si A est un type, ρ une fonction de valuation et Δ un ensemble de noms d'exceptions, alors*

$$|(\forall \alpha. A) \uplus \Delta|_\rho = |\forall \alpha. (A \uplus \Delta)|_\rho$$

Preuve :

Comme pour le lemme précédent, la commutation entre l'opérateur $\downarrow_\Delta$ et l'union de l'interprétation de la quantification universelle est directe.

Lemme 7.44 Si A est un type, ρ une fonction de valuation et Δ et Δ' deux ensembles de noms d'exceptions, alors

$$\| (A \uplus \Delta)^{\Delta'} \|_{\rho} = \| (A^{\Delta'}) \uplus \Delta \|_{\rho}$$

Preuve :

C'est une conséquence directe du fait 7.25.

Par orthogonalité, les lemmes 7.42, 7.43 et 7.44 ont le corollaire suivant :

Corollaire Si A est un type, ρ une fonction de valuation et Δ et Δ' deux ensembles de noms d'exceptions, alors on a les égalités suivantes :

$$\begin{aligned} \| (\forall \alpha. A)^{\Delta} \|_{\rho} &= \| \forall \alpha. A^{\Delta} \|_{\rho} \\ \| (\forall \alpha. A) \uplus \Delta \|_{\rho} &= \| \forall \alpha. A \uplus \Delta \|_{\rho} \\ \| (A \uplus \Delta)^{\Delta'} \|_{\rho} &= \| (A^{\Delta'}) \uplus \Delta \|_{\rho} \end{aligned}$$

On a également les deux lemmes suivants :

Lemme 7.45 Si A est un type, ρ une fonction de valuation et Δ et Δ' deux ensembles de noms d'exceptions, alors

$$\| (A^{\Delta})^{\Delta'} \|_{\rho} = \| A^{\Delta \cup \Delta'} \|_{\rho}$$

Preuve :

C'est une conséquence directe du lemme 7.26.

Lemme 7.46 Si A est un type, ρ une fonction de valuation et Δ et Δ' deux ensembles de noms d'exceptions, alors

$$\| (A \uplus \Delta) \uplus \Delta' \|_{\rho} = \| A \uplus (\Delta \cup \Delta') \|_{\rho}$$

Preuve :

C'est aussi une conséquence directe du lemme 7.26.

On a maintenant tous les éléments pour montrer que l'interprétation valide le sous-typage de λ_{ε} . Notons que l'on ne peut pas montrer simplement que si $A \leq B$, alors $\| A \|_{\rho} \subseteq \| B \|_{\rho}$ a cause du cas de *(st-arrow)*. On renforce donc l'hypothèse d'induction en montrant que pour tout Δ , $\| A^{\Delta} \|_{\rho} \subseteq \| B^{\Delta} \|_{\rho}$. Par ailleurs, ce résultat plus général est nécessaire pour la preuve du théorème de correction du typage vis-à-vis de l'interprétation (voir section 7.3.5).

Lemme 7.47 (Correction du sous-typage vis-à-vis de l'interprétation) Si A et B sont deux types et ρ une fonction de valuation tel que

$$A \leq B,$$

alors pour tout ensemble de noms d'exceptions Δ ,

$$\llbracket A^\Delta \rrbracket_\rho \subseteq \llbracket B^\Delta \rrbracket_\rho$$

Preuve :

On procède par induction sur la dérivation de $A \leq B$:

(st-id) Direct par réflexivité de l'inclusion d'ensemble.

(st-trans) Direct par hypothèse d'induction et transitivité de l'inclusion d'ensemble.

(st-arrow) Soit $\Delta \subseteq \mathcal{E}$ et $t \in \llbracket (A \rightarrow B)^\Delta \rrbracket_\rho = \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$. On doit montrer que $t \in \llbracket (A' \rightarrow B')^\Delta \rrbracket_\rho = \llbracket A'^\Delta \rightarrow B'^\Delta \rrbracket_\rho$. Donc, soit $C \in |A'^\Delta \rightarrow B'^\Delta|_\rho$, on va montrer que $C[t] \rightarrow^* \mathbf{\star}$.

Par définition de $|A'^\Delta \rightarrow B'^\Delta|_\rho$, il existe un ensemble de nom d'exceptions Δ' , un terme $u \in \llbracket A'^{\Delta \cup \Delta'} \rrbracket_\rho$ et un contexte $D \in |B'^{\Delta \cup \Delta'}|_\rho$ tel que $C = D[\llbracket \] u]$. Par hypothèse d'induction (et on a ici besoin de l'hypothèse d'induction renforcée), $\llbracket A'^{\Delta \cup \Delta'} \rrbracket_\rho \subseteq \llbracket A^{\Delta \cup \Delta'} \rrbracket_\rho$. Mais comme $t \in \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$, avec le lemme 7.38, on a $t u \in \llbracket B^{\Delta \cup \Delta'} \rrbracket_\rho$. Et par hypothèse d'induction, on a $\llbracket B^{\Delta \cup \Delta'} \rrbracket_\rho \subseteq \llbracket B'^{\Delta \cup \Delta'} \rrbracket_\rho$. Finalement, on a bien $C[t] = D[t u] \rightarrow^* \mathbf{\star}$.

(st-gen) Soit $\Delta \subseteq \mathcal{E}$ et $t \in \llbracket A^\Delta \rrbracket_\rho$. On doit montrer que $t \in \llbracket (\forall \alpha. B)^\Delta \rrbracket_\rho = \llbracket \forall \alpha. B^\Delta \rrbracket_\rho$. Soit $C \in |\forall \alpha. B^\Delta|_\rho$, il existe, par définition, un ensemble de contexte S tel que $C \in |B^\Delta|_{\rho; \alpha \leftarrow S}$. De plus, puisque $\alpha \notin \text{FV}(A)$, $\llbracket A^\Delta \rrbracket_\rho = \llbracket A^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$ et comme par hypothèse d'induction $\llbracket A^\Delta \rrbracket_{\rho; \alpha \leftarrow S} \subseteq \llbracket B^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$, on a finalement que $t \in \llbracket B^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$, d'où on déduit que $C[t] \rightarrow^* \mathbf{\star}$.

(st-inst) Soit $\Delta \subseteq \mathcal{E}$, on va montrer que $|(A\{\alpha := B\})^\Delta|_\rho \subseteq |(\forall \alpha. A)^\Delta|_\rho$ et on conclura par orthogonalité. Soit $C \in |(A\{\alpha := B\})^\Delta|_\rho$, on montre facilement (par induction sur A) que $|(A\{\alpha := B\})^\Delta|_\rho = |A^\Delta|_{\rho; \alpha \leftarrow |B|_\rho}$. Alors par définition de $|\forall \alpha. A^\Delta|_\rho$, on a $|A^\Delta|_{\rho; \alpha \leftarrow |B|_\rho} \subseteq |\forall \alpha. A^\Delta|_\rho$ et donc $C \in |\forall \alpha. A^\Delta|_\rho$. On conclut avec le lemme 7.42.

(st-distr) Soit $\Delta \subseteq \mathcal{E}$ et $t \in \llbracket (\forall \alpha. (A \rightarrow B))^\Delta \rrbracket_\rho = \llbracket \forall \alpha. (A^\Delta \rightarrow B^\Delta) \rrbracket_\rho$. Soit $C \in |A^\Delta \rightarrow \forall \alpha. B^\Delta|_\rho$, alors par définition, il existe un ensemble de noms d'exceptions Δ' , un terme $u \in \llbracket A^{\Delta \cup \Delta'} \rrbracket_\rho$ et un contexte $D \in |(\forall \alpha. B^\Delta \Delta')^{\Delta'}|_\rho = |\forall \alpha. B^{\Delta \cup \Delta'}|_\rho$ tel que $C = D[\llbracket \] u]$. Par définition, il existe un ensemble de contexte S , tel que $D \in |B^{\Delta \cup \Delta'}|_{\rho; \alpha \leftarrow S}$. Et puisque $\alpha \notin \text{FV}(A)$, on a $u \in \llbracket A^{\Delta \cup \Delta'} \rrbracket_{\rho; \alpha \leftarrow S}$. Donc, par définition de $\llbracket \forall \alpha. (A^\Delta \rightarrow B^\Delta) \rrbracket_\rho$ et en utilisant le lemme 7.38, on a $t u \in \llbracket B^{\Delta \cup \Delta'} \rrbracket_{\rho; \alpha \leftarrow S}$ et finalement $C[t] = D[t u] \rightarrow^* \mathbf{\star}$.

(st-cor-noex) Avec le lemme 7.45, on a $\llbracket ((A^\emptyset)^{\Delta'})^\Delta \rrbracket_\rho = \llbracket A^{\Delta \cup \Delta'} \rrbracket_\rho = \llbracket (A^\Delta)^{\Delta'} \rrbracket_\rho$.

(st-uni-merge) Avec les lemmes 7.46 et 7.44, on a $\llbracket ((A \uplus \Delta') \uplus \Delta'')^\Delta \rrbracket_\rho = \llbracket A^\Delta \uplus (\Delta' \cup \Delta'') \rrbracket_\rho = \llbracket (A \uplus (\Delta' \cup \Delta''))^\Delta \rrbracket_\rho$.

(st-cor-merge) Avec le lemme 7.45, on a $\llbracket ((A^{\Delta'})^{\Delta''})^\Delta \rrbracket_\rho = \llbracket A^{\Delta \cup \Delta' \cup \Delta''} \rrbracket_\rho = \llbracket (A^{\Delta' \cup \Delta''})^\Delta \rrbracket_\rho$.

(st-uni-arrow) Soit Δ un ensemble de noms d'exceptions et soit $t \in \llbracket ((A \rightarrow B) \uplus \Delta')^\Delta \rrbracket_\rho = \llbracket (A^\Delta \rightarrow B^\Delta) \uplus \Delta' \rrbracket_\rho$, on doit montrer que $t \in \llbracket (A \rightarrow B \uplus \Delta')^\Delta \rrbracket_\rho = \llbracket A^\Delta \rightarrow B^\Delta \uplus \Delta' \rrbracket_\rho$. Donc, avec le lemme 7.38, soit Δ'' est un ensemble de noms d'exceptions et soit $u \in \llbracket A^{\Delta \cup \Delta''} \rrbracket_\rho$, on doit montrer que $t u \in \llbracket B^{\Delta \cup \Delta''} \uplus \Delta' \rrbracket_\rho$. Or, le lemme 7.40 nous dit que

- Soit $t \rightarrow^* \text{raise } \varepsilon$ pour $\varepsilon \in \Delta'$, mais alors $t u \rightarrow^* \text{raise } \varepsilon$ et donc $t u \in \llbracket B^{\Delta \cup \Delta'} \uplus \Delta' \rrbracket_\rho$ (lemme 7.40).
 - Soit $t \in \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$, mais alors, avec le lemme 7.38, $t u \in \llbracket B^{\Delta \cup \Delta'} \rrbracket_\rho$ et donc $t u \in \llbracket B^{\Delta \cup \Delta'} \uplus \Delta' \rrbracket_\rho$ (lemme 7.40).
- (*st-uni-ctx*) Soit $\Delta \subseteq \mathcal{E}$, par hypothèse d'induction, $\llbracket A^\Delta \rrbracket_\rho \subseteq \llbracket B^\Delta \rrbracket_\rho$. On conclut alors facilement grâce aux deux égalités suivantes :
 $\llbracket (A \uplus \Delta')^\Delta \rrbracket_\rho = \llbracket A^\Delta \uplus \Delta' \rrbracket_\rho = \llbracket A^\Delta \rrbracket_\rho \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta' \}$ et
 $\llbracket (B \uplus \Delta')^\Delta \rrbracket_\rho = \llbracket B^\Delta \uplus \Delta' \rrbracket_\rho = \llbracket B^\Delta \rrbracket_\rho \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta' \}$ (lemme 7.40).
- (*st-uni*) Soit $\Delta \subseteq \mathcal{E}$, on doit montrer que $\llbracket A^\Delta \rrbracket_\rho \subseteq \llbracket (A \uplus \Delta')^\Delta \rrbracket_\rho = \llbracket A^\Delta \uplus \Delta' \rrbracket_\rho$. Mais avec le lemme 7.40, on a $\llbracket A^\Delta \uplus \Delta' \rrbracket_\rho = \llbracket A^\Delta \rrbracket_\rho \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta' \}$, ce qui nous permet de conclure directement.
- (*st-cor*) Soit $\Delta \subseteq \mathcal{E}$, on doit montrer que $\llbracket (A \uplus \Delta')^\Delta \rrbracket_\rho = \llbracket A^\Delta \uplus \Delta' \rrbracket_\rho \subseteq \llbracket (A^\Delta)^\Delta \rrbracket_\rho = \llbracket A^{\Delta \cup \Delta'} \rrbracket_\rho$. Mais avec le lemme 7.40, on a $\llbracket A^\Delta \uplus \Delta' \rrbracket_\rho = \llbracket A^\Delta \rrbracket_\rho \cup \{ t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta' \}$ et on conclut donc avec les lemmes 7.34 et 7.35.
- (*st-uni-cor*) On conclut facilement avec les lemmes 7.44 et 7.45.
- (*st-uni-fall*) On conclut facilement avec les lemmes 7.43, 7.44 et lemme 7.42.
- (*st-cor-fall*) C'est une application directe des lemmes 7.42 et 7.45.
- (*st-cor-arrow*) C'est une application directe des lemmes 7.41 et 7.45.

Remarquons que si on était capable de prouver que $A \leq B$ implique $|B^\Delta|_\rho \subseteq |A^\Delta|_\rho$, ce résultat impliquerait le lemme précédent par orthogonalité. Cependant, cette implication est fausse. Il est en effet facile de vérifier que $|\text{nat} \uplus \Delta|_\rho \not\subseteq |\text{nat}|_\rho$ puisque

$$\begin{aligned} |\text{nat} \uplus \Delta|_\rho &= \{ \text{rec } \boxtimes (\lambda y. \lambda x. x) (\text{try } [] \text{ with } \Delta) \} \\ |\text{nat}|_\rho &= \{ \text{rec } \boxtimes (\lambda y. \lambda x. x) [] \} \end{aligned}$$

7.3.5 Correction du typage vis-à-vis du modèle

La notion de corruption de type s'étend naturellement aux environnements de typage :

Définition 7.48 (Corruption d'un environnement de typage)

Si $\Gamma = x_1 : A_1, \dots, x_n : A_n$ est un environnement de typage, alors l'environnement de typage Γ^Δ est défini par

$$\Gamma^\Delta = x_1 : A_1^\Delta, \dots, x_n : A_n^\Delta$$

L'interprétation des types s'étend alors également très naturellement à cette notion d'environnement de typage corrompu :

Définition 7.49 (Interprétation d'un environnement de typage)

Si Γ est un environnement de typage, Δ un ensemble de nom d'exceptions et ρ une fonction de valuation,

$$\llbracket \Gamma^\Delta \rrbracket_\rho = \{ \sigma \mid \forall (x : A) \in \Gamma, \sigma(x) \in \llbracket A^\Delta \rrbracket_\rho \}.$$

Dans la suite, on notera plus simplement $\llbracket \Gamma \rrbracket_\rho$ l'ensemble $\llbracket \Gamma^\emptyset \rrbracket_\rho$. Remarquons que comme $\llbracket A \rrbracket_\rho \subseteq \llbracket A^\Delta \rrbracket_\rho$ (lemme 7.35), on a en particulier $\llbracket \Gamma \rrbracket_\rho \subseteq \llbracket \Gamma^\Delta \rrbracket_\rho$, et plus généralement $\llbracket \Gamma^\Delta \rrbracket_\rho \subseteq \llbracket \Gamma^{\Delta'} \rrbracket_\rho$ si $\Delta \subseteq \Delta'$.

De façon similaire au cas de la correction du sous-typage vis-à-vis de l'interprétation, pour montrer que le l'interprétation valide le typage, on ne peut pas se contenter de montrer que si $\Gamma \vdash t : A$ alors $t[\sigma] \in \llbracket A \rrbracket_\rho$ (pour $\sigma \in \llbracket \Gamma \rrbracket_\rho$). Le problème est le cas de la règle (*t-abs*). Intuitivement, cela n'est pas étonnant puisque que l'interprétation de la flèche de λ_e est plus précise que la flèche de la réalisabilité usuelle (voir section 5.1). On renforce donc l'hypothèse d'induction et on montre le théorème suivant (où on utilisera la substitution parallèle définie pour le premier modèle (définition 5.9)) :

Théorème 7.50 (Correction du typage vis-à-vis de l'interprétation) *Si t est un terme, A un type et Γ un environnement de typage tels que*

$$\Gamma \vdash t : A,$$

alors pour toute fonction de valuation ρ , pour tout ensemble fini de noms d'exceptions $\Delta \subseteq \mathcal{E}$ et pour toute substitution $\sigma \in \llbracket \Gamma^\Delta \rrbracket_\rho$, on a

$$t[\sigma] \in \llbracket A^\Delta \rrbracket_\rho$$

Preuve :

On procède par induction sur la dérivation de $\Gamma \vdash t : A$. Notons que grâce au lemme 7.47, on a que $\llbracket A \rrbracket_\rho \subseteq \llbracket A^\Delta \rrbracket_\rho$ et on se contentera donc, lorsque cela est possible, de prouver que $t[\sigma] \in \llbracket A \rrbracket_\rho$:

(t-ax) Montrer que $x[\sigma] \in \llbracket A^\Delta \rrbracket_\rho$ si $(x : A) \in \Gamma$ est une conséquence direct du fait que $\sigma \in \llbracket \Gamma^\Delta \rrbracket_\rho$.

(t-abs) Soit un contexte $C \in \mid (A \rightarrow B)^\Delta \mid_\rho = \mid A^\Delta \rightarrow B^\Delta \mid_\rho$, on veut montrer que $C[\lambda x. t[\sigma]] \rightarrow^* \star$. Par définition, il existe un ensemble de noms d'exceptions Δ' , un terme $u \in \llbracket A^{\Delta'} \rrbracket_\rho$ et un contexte $D \in \mid B^{(\Delta \cup \Delta')} \mid_\rho$ tel que $C = D[\mid \mid u]$. Alors, soit $\delta = \sigma + \{x \rightarrow u\}$, on a que $\delta \in \llbracket (\Gamma, x : A)^{(\Delta \cup \Delta')} \rrbracket_\rho$ (car $\sigma \in \llbracket \Gamma^{(\Delta \cup \Delta')} \rrbracket_\rho \subseteq \llbracket \Gamma^{(\Delta \cup \Delta')} \rrbracket_\rho$ et $u \in \llbracket A^{\Delta'} \rrbracket_\rho \subseteq \llbracket A^{\Delta \cup \Delta'} \rrbracket_\rho$) et donc, par hypothèse d'induction, $t[\delta] \in \llbracket B^{(\Delta \cup \Delta')} \rrbracket_\rho$. Or, $(\lambda x. t[\sigma]) u \rightarrow t[\delta]$ et $\llbracket B^{(\Delta \cup \Delta')} \rrbracket_\rho$ est clos par anti-réduction (lemme 7.31), donc $(\lambda x. M[\sigma]) u \in \llbracket B^{(\Delta \cup \Delta')} \rrbracket_\rho$ et finalement $C[\lambda x. M[\sigma]] = D[(\lambda x. M[\sigma]) u] \rightarrow^* \star$.

(t-app) Par hypothèse d'induction, $t[\sigma] \in \llbracket (A \rightarrow B)^\Delta \rrbracket_\rho = \llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho$ et $u[\sigma] \in \llbracket A^\Delta \rrbracket_\rho$. Et comme, grâce au lemme 7.38, on a $\llbracket A^\Delta \rightarrow B^\Delta \rrbracket_\rho \subseteq \{ t \mid \forall u \in \llbracket A^\Delta \rrbracket_\rho, t u \in \llbracket B^\Delta \rrbracket_\rho \}$, d'où on obtient $t[\sigma] u[\sigma] \in \llbracket B^\Delta \rrbracket_\rho$.

(t-gen) Soit $C \in \mid (\forall \alpha. A)^\Delta \mid_\rho = \mid \forall \alpha. A^\Delta \mid_\rho$, par définition il existe un ensemble de contexte non vide S tel que $C \in \mid A^\Delta \mid_{\rho; \alpha \leftarrow S}$. D'autre part, puisque $\alpha \notin \text{FV}(\Gamma)$, $\llbracket \Gamma^\Delta \rrbracket_\rho = \llbracket \Gamma^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$ et on peut donc appliquer l'hypothèse d'induction pour montrer que $t[\sigma] \in \llbracket A^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$. Donc finalement $C[t[\sigma]] \rightarrow^* \star$ et on a $t[\sigma] \in \llbracket (\forall \alpha. A)^\Delta \rrbracket_\rho$.

(t-zero) Ce cas est direct grâce au lemme 7.37.

(t-succ) On doit montrer que $S \in \llbracket \text{nat} \rightarrow \text{nat} \rrbracket_\rho$ et donc que pour tout ensemble de noms d'exceptions Δ , $S \in \{ t \mid \forall u \in \llbracket \text{nat}^\Delta \rrbracket_\rho, t u \in \llbracket \text{nat}^\Delta \rrbracket_\rho \}$ (lemme 7.38). Donc soit $u \in \llbracket \text{nat}^\Delta \rrbracket_\rho$, on doit montrer que $S u \in \llbracket \text{nat}^\Delta \rrbracket_\rho$, mais ceci est facile avec le lemme 7.37.

- (t-rec)** Montrons que $\text{rec} \in \llbracket \forall \alpha. \alpha \uplus \Delta \rightarrow (\text{nat}^\Delta \rightarrow \alpha \uplus \Delta \rightarrow \alpha \uplus \Delta) \rightarrow \text{nat}^\Delta \uplus \Delta' \rightarrow \alpha \uplus (\Delta \cup \Delta') \rrbracket_\rho$. En utilisant les lemmes 7.39 et 7.38, on se ramène à montrer que, quel que soit l'ensemble de contexte non vide S et les termes $z \in \llbracket \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S}$, $f \in \llbracket \text{nat}^\Delta \rightarrow \alpha \uplus \Delta \rightarrow \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S}$ et $n \in \llbracket \text{nat}^\Delta \uplus \Delta' \rrbracket_{\rho; \alpha \leftarrow S}$, on a $\text{rec } z f n \in \llbracket \alpha \uplus (\Delta \cup \Delta') \rrbracket_{\rho; \alpha \leftarrow S}$. En utilisant le lemme 7.40, on a ou bien $n \rightarrow^* \text{raise } \varepsilon$ pour $\varepsilon \in \Delta'$, ou bien $n \in \llbracket \text{nat}^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$. Si $n \rightarrow^* \text{raise } \varepsilon$ pour $\varepsilon \in \Delta'$, alors $\text{rec } z f n \rightarrow^* \text{raise } \varepsilon$ et on a $\text{rec } z f n \in \llbracket \alpha \uplus (\Delta \cup \Delta') \rrbracket_{\rho; \alpha \leftarrow S}$ (on utilise le lemme 7.40 et la clôture de l'interprétation par anti-réduction, lemme 7.31). Sinon, $n \in \llbracket \text{nat}^\Delta \rrbracket_\rho$ et, avec le lemme 7.36, il existe un entier $m \geq 0$ tel que $n \rightarrow^* S^m \Phi$, où Φ est soit 0, soit $\bot$, soit $\text{raise } \varepsilon$ pour $\varepsilon \in \Delta$. On raisonne alors par induction sur m pour montrer que $\text{rec } z f (S^m \Phi) \in \llbracket \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S}$:
- Si $m = 0$, alors $n = \Phi$ et donc
 - Soit $n = 0$, mais alors $\text{rec } z f n \rightarrow^* z$ et on conclut directement.
 - Soit $n = \bot$, mais alors $\text{rec } z f n \rightarrow^* \bot$ et on conclut avec le lemme 7.32.
 - Soit $n = \text{raise } \varepsilon$ pour $\varepsilon \in \Delta$, mais alors $\text{rec } z f n \rightarrow^* \text{raise } \varepsilon$ et on conclut avec le lemme 7.40.
 - Sinon, par hypothèse d'induction on a $\text{rec } z f (S^m \Phi) \in \llbracket \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S}$ et on doit montrer que $\text{rec } z f (S (S^m \Phi)) \in \llbracket \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S}$. Mais $\text{rec } z f (S (S^m \Phi)) \rightarrow f (S^m \Phi) (\text{rec } z f (S^m \Phi))$. Or, par définition $S^m \Phi \in \llbracket \text{nat}^\Delta \rrbracket_{\rho; \alpha \leftarrow S}$ et donc, en utilisant l'hypothèse d'induction et le lemme 7.38, on a $f (S^m \Phi) (\text{rec } z f (S^m \Phi)) \in \llbracket \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S}$ et on conclut par clôture par anti-réduction de l'interprétation.
- Finalement $\text{rec } z f n \in \llbracket \alpha \uplus (\Delta) \rrbracket_{\rho; \alpha \leftarrow S}$ et on conclut alors facilement puisque $\llbracket \alpha \uplus \Delta \rrbracket_{\rho; \alpha \leftarrow S} \subseteq \llbracket \alpha \uplus (\Delta \cup \Delta') \rrbracket_{\rho; \alpha \leftarrow S}$ (lemme 7.40).
- (t-raise)** Montrer que $\text{raise } \varepsilon \in \llbracket \emptyset \uplus \{\varepsilon\} \rrbracket_\rho$ est direct en utilisant le fait que $\llbracket A \uplus \Delta \rrbracket_\rho = \llbracket A \rrbracket_\rho \cup \{t \mid t \rightarrow^* \text{raise } \varepsilon, \varepsilon \in \Delta\}$ (lemme 7.40).
- (t-try)** Par application de l'hypothèse d'induction, on a $t[\sigma] \in \llbracket (A \uplus \{\varepsilon\})^\Delta \rrbracket_\rho = \llbracket A^\Delta \uplus \{\varepsilon\} \rrbracket_\rho$ et $u[\sigma] \in \llbracket A^\Delta \rrbracket_\rho$. En utilisant le lemme 7.40, on est dans l'un des deux cas suivants :
- $t[\sigma] \in \llbracket A \rrbracket_\rho$ et alors on utilise le lemme 7.33 pour montrer que $t[\sigma]$ a une valeur. Alors, ou bien $\text{try } t[\sigma] \text{ with } \varepsilon \mapsto u[\sigma] \rightarrow^* t'$ avec $t[\sigma] \rightarrow^* t'$ (et comme l'interprétation est closes par réduction (lemme 7.31), $t' \in \llbracket A^\Delta \rrbracket_\rho$), ou bien $\text{try } t[\sigma] \text{ with } \varepsilon \mapsto u[\sigma] \rightarrow^* u[\sigma]$. Dans les deux cas, ces termes appartiennent à $\llbracket A^\Delta \rrbracket_\rho$ et on conclut en utilisant le lemme de clôture par anti-réduction (lemme 7.31).
 - $t[\sigma] \rightarrow^* \text{raise } \varepsilon$ et alors $\text{try } t[\sigma] \text{ with } \varepsilon \mapsto u[\sigma] \rightarrow^* u[\sigma]$ et on conclut car $\llbracket A^\Delta \rrbracket_\rho$ est clos par anti-réduction (lemme 7.31).
- (t-subtyp)** La preuve est une application direct de l'hypothèse d'induction et du lemme 7.47.

7.3.6 Propriétés du calcul dérivées du modèle

La principale propriété que l'on obtient grâce au modèle est le fait que tout programme de F_ε se réduit sur une valeur :

Théorème 7.51 (Réduction sur une valeur) *Si t est un terme et A un type tel que $\vdash t : A$, alors t a une valeur.*

Preuve :

Ce théorème est une conséquence directe du théorème de correction du typage vis-à-vis de l'interprétation (théorème 7.50) et du fait que tout terme du modèle a une valeur (lemme 7.33).

De plus, on déduit du modèle la sûreté du typage vis-à-vis des entiers naturels :

Lemme 7.52 *Si t est un terme tel que $\vdash t : \text{nat}$, alors il existe un entier $n \geq 0$ tel que*

$$t \rightarrow^* S^n 0.$$

Preuve :

Si $\vdash t : \text{nat}$, par correction de l'interprétation vis-à-vis du typage (théorème 7.50), on a $t \in \llbracket \text{nat} \rrbracket_\rho$. Or, avec le lemme 7.36,

$$\llbracket \text{nat} \rrbracket_\rho = \{ t \mid t \rightarrow^* S^n 0, n \in \mathbb{N} \} \cup \{ t \mid t \rightarrow^* S^n \boxtimes, n \in \mathbb{N} \}.$$

Et comme t ne peut contenir $\boxtimes$ (le *daimon* n'a aucune règle de typage), on a bien que $t \rightarrow^* S^n 0$.

7.4 Les codages du second-ordre

Les entiers naturels sont un type de données primitif de F_ε . Cependant, le calcul F_ε est une extension du système F et supporte donc la définition de types de données à l'aide de leurs codages au second-ordre [GTL89]. Dans cette section, nous montrons comment de tels codages se comportent vis-à-vis des exceptions.

On rappelle le codage au second-ordre des entiers naturels :

$$\begin{aligned} \overline{\text{nat}} &\equiv \forall \alpha. (\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha) \\ \overline{0} &\equiv \lambda z. \lambda s. z : \overline{\text{nat}} \\ \overline{1} &\equiv \lambda z. \lambda s. s z : \overline{\text{nat}} \\ \overline{n} &\equiv \lambda z. \lambda s. \underbrace{s (\dots (s z) \dots)}_n : \overline{\text{nat}} \\ \overline{\text{succ}} &\equiv \lambda n. \lambda z. \lambda s. s (n z s) : \overline{\text{nat}} \rightarrow \overline{\text{nat}} \end{aligned}$$

Ce codage des entiers est particulièrement intéressant car ses formes normales sont exactement les termes de la forme $\overline{n}$ définie ci-dessus. Ce type n'est pas choisi au hasard puisqu'il est connu qu'en prenant une variante de ce type aussi simple que

$$\forall \alpha. ((\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha),$$

qui change simplement l'ordre des arguments, on se retrouve avec deux termes pour représenter l'entier naturel 1, les termes $\lambda s. \lambda z. s z$ et $\lambda s. s$.

Dans F_ε , on a montré avec le lemme 7.52 que les formes normales de type nat sont exactement les termes de la forme $S^n 0$, désignés plus simplement par n . On peut associer à chaque entier n sa représentation sous forme d'entiers codé au second-ordre $\lceil n \rceil$ en prenant

$$\lceil S^n 0 \rceil = \lambda z. \lambda s. \underbrace{s (\dots (s z) \dots)}_n$$

Si pour un type A , on note $fn(A)$ l'ensemble de ses formes normales, alors on a

$$\lceil fn(\text{nat}) \rceil = fn(\overline{\text{nat}})$$

Mais les entiers codés au second-ordre offrent plus de possibilité de corruption que les entiers primitifs. En effet, les formes normales de type nat^Δ sont de la forme $S^n 0$ ou $S^n (\text{raise } \varepsilon)$ pour $\varepsilon \in \Delta$ (voir lemme 7.37). On peut alors représenter un entier corrompu par un codage au second-ordre en associant aux nouvelles formes normales :

$$[S^n (\text{raise } \varepsilon)] = \lambda z. \lambda s. \underbrace{s \dots (s (\text{raise } \varepsilon))}_{n} \dots$$

Cette représentation est injective, mais pas surjective :

$$[fn(\text{nat}^\Delta)] \subset fn(\overline{\text{nat}}^\Delta)$$

En effet, les termes $\text{raise } \varepsilon$ et $\lambda z. \text{raise } \varepsilon$ ont tous les deux le type $\overline{\text{nat}}^\Delta$ mais ne sont la représentation d'aucun entier primitif. On a plus d'entiers corrompus avec les codages au second-ordre qu'avec les entiers primitifs.

Par ailleurs, le principale problème des entiers dans leur codage au second-ordre est pour « décorrompre » une entier corrompu. Autrement dit, peut on écrire une fonction $\overline{cbind}_{\text{nat}}$ ayant le type $\forall \alpha. \overline{\text{nat}}^\Delta \rightarrow (\overline{\text{nat}} \rightarrow \alpha) \rightarrow \alpha \uplus \Delta$ et dont le comportement est semblable à celle de la fonction $cbind_{\text{nat}}$ de la section 4.3, mais pour les entiers codés au second-ordre ? D'un point de vue purement calculatoire, oui. En utilisant les entiers comme leur propre récursur, on peut coder cette fonction par :

$$\overline{cbind}_{\text{nat}} \equiv \lambda n. \lambda f. (n (\lambda a. f a) (\lambda g. \lambda a. g (\overline{\text{succ}} a))) \bar{0}$$

Ce codage est similaire à celui utilisé pour les entiers primitifs, mais le système de type n'est pas assez précis pour typer cette fonction avec le type $\forall \alpha. \overline{\text{nat}}^\Delta \rightarrow (\overline{\text{nat}} \rightarrow \alpha) \rightarrow \alpha \uplus \Delta$.

Notons que pour le modèle de réalisabilité de F_ε , la fonction $\overline{cbind}_{\text{nat}}$ est bien dans l'interprétation du type $\forall \alpha. \overline{\text{nat}}^\Delta \rightarrow (\overline{\text{nat}} \rightarrow \alpha) \rightarrow \alpha \uplus \Delta$. Ainsi, une solution *ad-hoc* consiste à forcer ce typage.

Cependant, l'utilisation des codages du second-ordre dans F_ε n'est donc pas pleinement satisfaisante en pratique et on y préférera l'utilisation des entiers primitifs.

8

Perspectives

8.1 Types dépendants

Une perspective naturelle des travaux de cette thèse est l'extension des calculs présentés au produit dépendant. De telles extensions demandent toutefois une réorganisation complète de la syntaxe. Néanmoins, nous savons déjà comment étendre le modèle de réalisabilité de F_ε pour le produit dépendant. En effet, si T est un type et U_x une famille de type indexée par x , nous prendrons

$$|\Pi x : T. U| = \bigcup_{\Delta \subseteq \mathcal{E}} \{ M \cdot C \mid M \in \llbracket T^\Delta \rrbracket \wedge C \in |(U_M)^\Delta| \}$$

D'autre part, dans le cadre d'une présentation sous forme de système de types pur (PTS) [Bar92] où les types sont des termes comme les autres, ceux-ci pourront être corrompus et on devra donc être capable de gérer un terme/type comme

$$\text{nat} \rightarrow \text{raise } \varepsilon$$

Il est sans doute possible d'interdire la formation d'un tel terme par sortage. Cependant, le *leitmotiv* de la corruption est d'autoriser le remplacement de n'importe quel sous-terme par $\text{raise } \varepsilon$ et il serait donc plus intéressant de donner à ce terme le type $s^{\{\varepsilon\}}$ où s est une certaine sorte.

Le typage du terme $\text{nat} \rightarrow \text{raise } \varepsilon$ oblige par ailleurs à modifier les règles de formation des produits dépendants pour tenir compte de ces sortes corrompues, et on proposera par exemple la règle de formation suivante :

$$\frac{\Gamma \vdash A : s1^\Delta \quad \Gamma, x : A \vdash B : s2^\Delta \quad (s1, s2, s3) \in \text{Sortes}}{\Gamma \vdash \Pi x:A. B : s3^\Delta}$$

Par ailleurs, les types pouvant mentionner des termes et en particulier des variables de terme libres, le jugement de sous-typage devra tenir compte de l'environnement de typage et le jugement de sous-typage y sera donc de la forme :

$$\Gamma \vdash A \leq B$$

Le sous-typage devra également tenir compte de la possibilité de convertir les types, mais ceci peut-être résolu en remplaçant la règle de sous-typage de l'identité par la règle suivante :

$$\frac{\Gamma \vdash A : s^\Delta \quad \Gamma \vdash B : s'^{\Delta'} \quad A =_\beta B}{\Gamma \vdash A \leq B}$$

Nous pensons que les idées que nous venons d'esquisser en quelques lignes sont un bon point de départ pour proposer une extension des systèmes de types purs avec nos exceptions et leur typage. Nous n'avons cependant pas encore poursuivi cette piste plus avant. Les bonnes propriétés des PTS sont-elles conservées ? Un tel système est-il suffisamment souple en pratique pour la preuve de programme utilisant des exceptions ?

8.2 Inférence de type

Un autre sujet que nous n'avons pas abordé dans ce document est l'inférence de type. Il n'est bien sûr pas question de chercher un algorithme d'inférence de type pour F_ε [Wel95], mais l'inférence de type pour λ_ε ou pour un fragment de F_ε mérite que l'on s'y intéresse.

Liée à cette question est aussi la question de types principaux avec les notions d'union et de corruption. Autrement dit, puisqu'un terme t de λ_ε ou F_ε a plusieurs types (une infinité de types pour être exact), existe-t-il toujours un type A tel que t ait le type A et tel que pour tout type B de t , on ait $A \leq B$?

8.3 Intégration dans Haskell

Calculatoirement, le mécanisme d'exception présenté dans cette thèse est proche des exceptions imprécises [PJRH⁺99], mais sans l'imprécision. L'imprécision consiste à considérer une valeur exceptionnelle non plus comme une seule exception, mais comme un ensemble d'exceptions (voir section 2.3.2 pour plus de détails). Les constructions de types que nous proposons dans cette thèse considèrent toutefois également des ensembles d'exceptions et nous pensons que le typage que nous proposons peut s'adapter presque sans modifications au cas des exceptions imprécises.

Les exceptions imprécises [PJRH⁺99] sont implémentées par le compilateur Haskell GHC [ghc]. Il serait donc intéressant d'essayer d'intégrer notre typage des exceptions à cette implémentation. Pour cela, on devra cependant adapter nos constructions de type (union et corruption) au système de types d'Haskell. Notons que l'étude de l'inférence de type évoquée dans la section précédente est particulièrement complémentaire de cette perspective d'implémentation.

8.4 Arguments d'exceptions

Enfin, il est naturel de vouloir étendre les calculs que nous avons présentés en ajoutant des arguments aux exceptions. Nous avons déjà évoqué les défis à relever pour une telle extension dans la section 6.4.

Bibliographie

- [Bar84] H.P. Barendregt. *The lambda calculus*. North-Holland, 1984.
- [Bar92] H.P. Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science*, 2:117–309, 1992.
- [BHT97] G. Barthe, J. Hatcliff, and P. Thiemann. Monadic type systems: Pure type systems for impure settings. *Proceedings of the Second HOOTS Workshop, Stanford University, Palo Alto, CA. December, 1997*.
- [Bir67] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1967.
- [BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [BY05] L. Birkedal and N.H. Yang. Semantics of Separation-Logic Typing and Higher-Order Frame Rules. *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 260–269, 2005.
- [CCG⁺84] W.J. Cody, J.T. Coonen, D.M. Gay, K. Hanson, D. Hough, W. Kahan, R. Kar-pinski, J. Palmer, F.N. Ris, and D. Stevenson. A Proposed Radix-and Word-length-independent Standard for Floating-point Arithmetic. *Micro, IEEE*, 4(4):86–100, 1984.
- [DH90] C. Dornan and K. Hammond. Exception handling in lazy functional languages. *Technical Report CS90/R5, Department of Computing Science, University of Glasgow*, 1990.
- [DM04] R. David and G. Mounier. An intuitionistic λ -calculus with exceptions. *Journal of Functional Programming*, 15(01):33–52, 2004.
- [FFCA98] M. Fähndrich, J. Foster, J. Cu, and A. Aiken. Tracking down Exceptions in Standard ML Programs. *Tech. Rep. 98-996, University of California at Berkeley, CA, USA*, 1998.
- [Fil96] A. Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996.
- [Fil99] A. Filinski. Representing layered monads. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 175–188, 1999.

- [ghc] The Glasgow Haskell Compiler, Control.Exception Module. <http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Exception.html>.
- [Gim94] E. Gimenez. Codifying guarded definitions with recursive schemes. *Types for Proofs and Programs*, 94:39–59, 1994.
- [Gir72] J.Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'ordre Supérieur*. PhD thesis, Université Paris VII, 1972, 1972.
- [Gir01] J.Y. Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(03):301–506, 2001.
- [GJS96] J. Gosling, B. Joy, and G.L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.
- [Gov93] R. Govindarajan. Exception handlers in functional programming languages. *Software Engineering, IEEE Transactions on*, 19(8):826–834, 1993.
- [GS94] J. Guzman and A. Suarez. An extended type system for exceptions. *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, 1994.
- [GTL89] J.Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. *Cambridge Tracts in Theoretical Computer Science*, 7, 1989.
- [Kie98] R.B. Kieburtz. Taming effects with monadic typing. *International Conference on Functional Programming, ICFP'98*, 34(1):51–62, 1998.
- [Kri90] J.L. Krivine. Opérateurs de mise en mémoire et traduction de Gödel. *Archive for Mathematical Logic*, 30(4):241–267, 1990.
- [Kri03] J.L. Krivine. Dependent choice, 'quote' and the clock. *Theoretical Computer Science*, 308(1-3):259–276, 2003.
- [L⁺07] X. Leroy et al. The Objective Caml system, release 3.10. *Documentation and user's manual*, 2007.
- [Leb08] Sylvain Lebesne. A System F with call-by-name exceptions. In L. Aceto, editor, *ICALP 2008, part II*, volume 5126 of *LNCS*, pages 323–335. Springer Verlag, 2008.
- [LP99] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 276–290, 1999.
- [LW99] X. Leroy and P. Weis. *Le langage Caml*. Dunod, 1999.
- [Mit79] G. Mitschke. The standardization theorem in-calculus. *Z. Math. Logik Grundlag. Math.*, 25:29–31, 1979.
- [Mit88] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.
- [Mog91] E. Moggi. Notions of computation and monads. *INF. COMPUT.*, 93(1):55–92, 1991.

- [Mou99] G. Mounier. *Un λ -calcul intuitionniste avec exceptions*. PhD thesis, Universite de Savoie, 1999.
- [MP99] J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.
- [MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. The Definition of Standard ML: Revised 1997, 1997.
- [Nan05] A. Nanevski. A modal calculus for exception handling. *Intuitionistic Modal Logics and Applications Workshop (IMLA'05), Chicago, IL, June*, 2005.
- [Par93] M. Parigot. Strong normalization for second order classical natural deduction. *Logic in Computer Science, 1993. LICS'93., Proceedings of Eighth Annual IEEE Symposium on*, pages 39–46, 1993.
- [Pes00] F. Pessaux. *Détection statique d'exceptions non rattrapées en Objective Caml*. PhD thesis, Université Pierre et Marie Curie – Paris 6, 2000.
- [Pie02] B.C. Pierce. *Types and Programming Languages*. Mit Press, 2002.
- [PJe⁺03] S. Peyton Jones (editor) et al. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- [PJRH⁺99] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. *ACM SIGPLAN Notices*, 34(5):25–36, 1999.
- [Raf07] C. Raffalli. PML, Proved ML – Programmable Mathematical Logic. 2007.
- [Rei98] A. Reid. Handling exceptions in Haskell. *Research Report, Yale University*, August 1998.
- [Rey74] J.C. Reynolds. Towards a theory of type structure. *Symposium on Programmings*, 19:408–423, 1974.
- [Rey02] JC Reynolds. Separation logic: a logic for shared mutable data structures. *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002.
- [RT99] J.G. Riecke and H. Thielecke. Typed Exceptions and Continuations Cannot Macro-Express Each Other. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 635–644, 1999.
- [Spi90] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.
- [Ste90] G. Steele. *Common LISP: The Language*. Digital Press, 1990.
- [Tea06] The Coq Development Team. The Coq Proof Assistant, Reference Manual Version 8.1. <http://coq.inria.fr/V8.1pl3/refman/index.html>, 2006.
- [Thi02] H. Thielecke. Comparing Control Constructs by Double-Barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2):141–160, 2002.

- [TU96] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 74–85, 1996.
- [Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. *Functional Programming Languages and Computer Architecture*, 201:1–16, 1985.
- [VM04] J. Vouillon and P.A. Mellies. Semantic types: a fresh look at the ideal model for types. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 52–63, 2004.
- [Wad85] P. Wadler. How to Replace Failure by a List of Successes A method for exception handling, backtracking, and pattern matching. *Functional Programming Languages and Computer Architecture*, 1985.
- [Wad90] P. Wadler. Comprehending monads. *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.
- [Wel95] J.B. Wells. The undecidability of Mitchell's subtyping relation. *Technical Report 95-019, Boston University, Boston, Massachusetts*, 1995.
- [Wel96] J.B. Wells. *Type inference for System F with and without the eta rule*. PhD thesis, Boston University, 1996.
- [Wer94] B. Werner. *Une Theorie des Constructions Inductives, these de doctorat*. PhD thesis, 1994.
- [WT03] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003.
- [YR97] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. *Proceedings of the 4th International Static Analysis Symposium*, pages 98–113, 1997.

Les exceptions ont d'abord été introduites dans les langages en appel par valeur. Pour les langages en appel par nom, la solution habituelle consiste à les encoder à l'aide d'une construction monadique. Bien qu'élégante théoriquement, cette solution présente plusieurs désavantages pratiques: propagation non automatique, l'utilisation des exceptions force l'appel par valeur et la modularité est en partie compromise.

Dans cette thèse, nous faisons le choix d'un mécanisme d'exceptions primitif ne possédant pas les désavantages d'un encodage. Nous en proposons un système de type où les types d'un terme rendent compte des exceptions qu'il mentionne. À cette fin, nous introduisons deux constructions de types. La première, l'union, caractérise les termes pouvant s'évaluer sur une exception et est cruciale pour typer la capture d'exception. La seconde, la corruption, rend compte d'un phénomène propre à l'appel par nom: une exception peut se voir "enfouie" dans un terme par β -réduction, sans que le terme résultant ne s'évalue nécessairement sur l'exception en question. Reposant sur une relation de sous-typage, la corruption offre un typage des exceptions à la fois modulaire et favorisant la réutilisation de code.

Les premiers chapitres étudient ces constructions de types dans le cadre du λ -calcul simplement typé. Nous proposons ensuite quelques extensions naturelles du calcul obtenu. Enfin, nous étendons ce calcul aux types du second-ordre. De plus, nous proposons pour chacun des calculs des modèles de réalisabilité, offrant ainsi une sémantique précise à la principale notion originale de ces travaux, la corruption.

Initially, exceptions have been introduced in call-by-value languages. For call-by-name evaluation, the usual solution is to encode them with a monadic construction. While theoretically elegant, this solution has several practical drawbacks: non-automatic propagation, the use of exceptions forces call-by-value evaluation and modularity is somewhat compromised.

In this thesis, we make the choice of a primitive mechanism of exceptions which does not have the drawbacks of an encoding. We propose a type system where the types of a term give an account on which exceptions the term mentions. To that end, we introduce two type constructions. The first one, the union, characterises terms that may evaluate to an exception and is crucial to type the capture of exceptions. The second one, the corruption, gives an account on a phenomenon specific to call-by-name evaluation: an exception can be "buried" in a term by β -reduction without having the resulting term necessarily evaluating to the exception. Based on a subtyping relation, corruption provides a typing of exceptions which is both modular and favors code re-usability.

The first chapters study these type constructions in the setting of the simply typed λ -calculus. Afterwards we propose a few natural extensions of the obtained calculus. Lastly, we extend this calculus to second-order types. Moreover, we propose realizability models for each of the calculi, providing a precise semantics to the main original notion of this work, corruption.