N^o d'ordre :

THÈSE

présentée à

UNIVERSITÉ PIERRE & MARIE CURIE - PARIS 6

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité

INFORMATIQUE

soutenue par

François PESSAUX

Titre

DÉTECTION STATIQUE D'EXCEPTIONS NON RATTRAPÉES EN OBJECTIVE CAML

Jury

MM.	Christian	QUEINNEC	Président
	Manuel	FÄHNDRICH	Rapporteurs
	Daniel	LE MÉTAYER	
	Radhia	COUSOT	Examinateurs
	Pierre	CRÉGUT	
	Xavier	LEROY	
	Bernard	LORHO	

À ma Puce ♡,
À mes parents

Remerciements

BON NOMBRES DE PERSONNES ont fait que ce travail ait pu avoir lieu, et ce dans d'excellentes conditions, tant à l'intérieur du projet Cristal où j'ai effectué ma thèse que dans d'autres projets voire laboratoires. Je vais tenter de les remercier ici, en espérant que je n'en oublierai que très peu...

M. Christian Queinec m'a fait l'honneur de diriger officiellement cette thèse et de présider son jury. Même si nous n'avons en réalité que très peu travaillé ensemble, je le remercie de bien avoir voulu accepter cette charge, sans laquelle il m'aurait été impossible d'effectuer mon travail de recherche.

En second lieu, ma gratitude va à MM. Manuel Fähndrich et Daniel Le Métayer qui ont accepté le lourd travail de rapporteurs de ma thèse. Leur lecture minutieuse et pertinente m'a permis de corriger un certain nombre d'erreurs et de rendre ce document de meillleure facture. Je leur en sais gré et ai réellement apprécié leur aide.

Je souhaite vivement remercier Xavier Leroy, qui a bien voulu encadrer cette thèse et faire également partie de son jury. Avec énormément de patience et de perspicacité, il a su me conseiller, écouter les idées, proposer les siennes et en règle générale se tenir au courant de mes travaux. Même si parfois, au premier abord, comprendre ses intuitions ou ses déductions relevait pour moi de la lecture dans le texte d'un *.cmx*, il a toujours réussi à me les faire appréhender. Sa rapidité et sa justesse d'analyse et de synthèse auront été d'une aide très appréciable tout au cours de mon travail. Je le remercie. Le seul reproche à lui manifester est d'avoir triché en *group-ware* contre moi en se liguant avec Pierre Weis, Damien Doligez et François Rouaix par téléphone pour savoir où mon personnage se trouvait afin de le tuer plus facilement. À vrai dire, malgré l'adversité, j'ai quand même gagné cette partie ce soir-là...

C'est avec plaisir que j'ai su que M^{me} Radhia Cousot acceptait de participer à mon jury. Éminente dans le domaine de l'interprétation abstraite et des analyses statiques de programmes en règle générale, elle s'est intéressée à mon travail de thèse, ce que je considérerai comme un honneur qu'elle m'a fait.

M. Bernard Lorho a lui aussi accepté la charge d'être membre de mon jury et j'en suis très heureux. Outre l'intérêt intellectuel de l'avoir parmi les membres de mon jury, c'est avec plaisir que je retrouve un de mes anciens professeurs de faculté, le premier qui m'ait fait un cours de compilation vraiment intéressant, duquel on pouvait faire quelque chose (c'est-à-dire implanter) et qui a contribué à accroître mon intérêt dans le domaine au point de finir par faire une thèse sur les langages de programmation.

France Télécom et plus particulièrement le CNET ont accepté de financer mon travail de recherche. Je tiens à les en remercier, ainsi que M. Pierre Crégut qui a bien voulu, d'une part servir d'intermédiaire entre le projet Cristal et le CNET, et d'autre part faire également partie de mon jury de thèse.

La réussite de ce travail doit aussi beaucoup à M. Pierre Weis. Avec toujours une bonne humeur et un humour piquant, il s'est intéressé à mon travail, me suggérant des idées, des problèmes ou des solutions. En particulier, la partie de cette thèse consacrée au typage en présence de récursion polymorphe est le fruit d'un travail conjoint avec lui, sans qui les résultats obtenus n'auraient sans doute jamais vu le jour. Toujours ouvert, il m'a aussi donné de nombreuses idées de petits *hacks*, des *smops* de deux ou trois heures (qui ont tous duré deux ou trois semaines) et s'est toujours vivement intéressé à mes développements parallèles, voire y a même contribué. Pour finir en tant que « co-bureau », il a même consenti à me laisser une moitié d'étagère sur mon bureau pour y mettre mes affaires, moitié que j'ai dû défendre âprement contre les piles diverses et variées poussant sporadiquement dans notre bureau commun...

Je désire également remercier M. Didier Rémy avec qui nous avons beaucoup discuté du présent travail. Son extraordinaire perspicacité m'aura été d'une précieuse aide tout au long de ce travail. Même si j'ai parfois eu du mal à le suivre, son esprit de synthèse m'a fait me rendre compte de nombreux liens entre mon travail et d'autres travaux antérieurs, permettant ainsi une compréhension plus globale des problèmes soulevés.

Ma gratitute va également à François Rouaix, qui a été mon premier directeur de stage à l'INRIA alors que je n'étais encore qu'en Licence. Nous avons partagé de nombreux moments de recherche, de *hack* et de discussion ensemble. Son ouverture d'esprit et sa curiosité m'ont permis de découvrir des domaines insoupçonnés de l'informatique. Il m'a également permis de faire mes premières armes en Caml et ainsi de continuer sur la voie qui m'a amené à la thèse.

Je tiens à ne pas oublier François Pottier à qui j'ai « emprunté un peu » de macros LaTeX afin d'écrire ma thèse et avec qui nous avons eu de très intéressantes discussions à propos de nos travaux respectifs. Jérôme Vouillon m'a également éclairé sur son travail : le typage des objets en Objective Caml, lorsqu'il m'a fallu envisager l'analyse de ces constructions. Et enfin je dois dire merci à Alexandre Miquel qui, en tant que prof de maths, a réussi à me réconcilier avec les matrices, les espaces à n dimensions, à me faire comprendre la théorie du *texture mapping en 3D* et en règle générale à me convaincre que tout ce que l'on avait tenté de m'apprendre en mathématiques depuis la maternelle pouvait avoir une application pour les graphismes 3D en temps réel...

J'ai également une pensée pour Thérèse Hardin sans qui je n'aurais jamais vu l'informatique comme je la vois maintenant. En tant que Professeur d'informatique et responsable du cours en Licence, c'est elle qui a transformé ma conception de l'informatique (la bidouille) en m'enseignant plus formellement la programmation au travers de Caml à l'époque où je ne jurais encore que par mon compilateur C sur mon Amiga. Par la suite c'est elle qui m'a mis en relation avec François Rouaix pour effectuer mon premier stage. Sans tout cela, nul doute que je n'aurais jamais eu la joie de suivre le chemin qui m'a amené jusqu'ici et que je serais certainement resté un *hacker* sans une vue plus synthétique de l'informatique.

Je ne pourrai pas oublier de remercier ma Puce, Valérie, qui a supporté avec une extraordinaire patience les moments parfois difficiles, traversés durant ce travail. Elle a toujours su, avec une patience évangélique, me permettre de m'investir autant que je le souhaitais dans mes recherches, puis-je lui rendre un jour la même preuve d'amour.

En règle générale, toutes les personnes que j'ai pu côtoyer dans mon travail à l'INRIA m'ont apporté leur gentillesse et leur dévouement. Sans chercher à les citer toutes, elles se reconnaîtront dans ces lignes, je tiens à leur exprimer toute ma gratitude pour avoir contribué à faire que ces trois années de thèse soient aussi intéressantes et agréables. Enfin, je souhaite remercier l'INRIA de m'avoir accueilli en me fournissant un environnement de travail exceptionnel, tant au niveau intellectuel (par la qualité des personnes y travaillant) qu'au niveau matériel (par l'étendue de ses moyens d'accès à l'information). Le cadre de travail y aura également toujours été très agréable, baigné dans la verdure parmi les lapins et les champignons...

Pour terminer j'aurai une pensée émue pour ma *Renault 9 Louisiane* qui nous a quitté après 266468 Km de bons et loyaux services et qui, jusqu'à temps que ses bielles n'en puissent plus, m'a toujours accompagné à l'INRIA sans faux bonds. Je souhaite maintenant que sa petite sœur, ayant hérité de son capot, son coffre, son volant, ses pneus et bien d'autres accessoires encore, saura se montrer digne de son illustre prédécesseur...

Introduction

Il est de coutume d'entrer de plain pied dans le cercle des programmeurs par le programme qui affiche « Hello World » (ou « Bonjour le monde » devrais-je plutôt dire). En règle générale, ceci se fait plutôt aisément, tout bon livre d'initiation fournissant cet exemple dans le texte. Dès lors que l'on désire approfondir cet art qu'est la programmation d'un ordinateur, les programmes que l'on doit écrire deviennent de plus en plus complexes et apparaît très rapidement la notion tant redoutée : le *bug*, l'erreur fatidique dans le programme. La taxinomie de ces animaux peut être grossièrement divisée en deux familles. Tout d'abord les bugs d'algorithme : l'ordinateur fait quelque chose, rend un résultat, mais ce résultat n'est pas une réponse valide au problème à résoudre car l'algorithme du programme lui-même n'est pas la bonne façon de résoudre le problème. L'autre possibilité est que l'ordinateur interrompe brutalement son calcul car il se trouve dans l'impossibilité d'exécuter les ordres du programme.

La première classe d'erreurs est clairement indépendante du langage de programmation utilisé. Détecter ce genre d'anomalie de fonctionnement requiert la donnée d'une description, une *spécification* de ce que l'on attend du programme. Il faut alors démontrer mathématiquement que le programme écrit satisfait cette spécification. Ce travail nécessite bien souvent la manipulation de notions mathématiques et logiques complexes et ne peut certes pas être automatisé.

Envisageons maintenant le second cas de programmes indésirables. Il est aisé d'exhiber parmi tant d'autres quelques exemples de telles configurations : essayer d'utiliser un entier comme une chaîne de caractères ou bien considérer un booléen comme une fonction et l'appliquer à des arguments mènent à coup sûr vers une fin prématurée du programme. Par exemple dans le premier des cas, stocker la chaîne « gabuzomeu » dans la zone mémoire réservée à l'entier va occasionner une pollution de l'espace mémoire (la chaîne mesurant au moins 9 caractères donc 9 octets et l'entier 4 ou 8 octets). Ainsi des informations seront écrites en mémoire de façon incohérente, risquant d'altérer l'intégrité d'autres structures de données. Dans le second exemple, la valeur du booléen va être considérée comme l'adresse du code de la fonction à exécuter, adresse où il y a toutes les chances de se trouver autre chose que du code exécutable et donc le programme va tenter d'interpréter des ordres qui n'en sont pas. Il est donc intéressant d'arriver à garantir qu'un programme n'aura pas un tel comportement à l'exécution, autrement dit qu'il ne « plantera » pas. Le mécanisme de typage fort statique utilisé dans les langages ML assure une telle propriété.

Le typage est donc un mécanisme consistant à annoter les éléments du programme (*expressions*) de façon à restreindre leurs utilisations dans des agencements légaux. Afin de rendre ce mécanisme élégant et agréable, les types ne sont pas explicitement donnés par le programmeur, mais directement devinés (« inférés ») par le compilateur. Nous entendons par typage *fort* la caractéristique interdisant au programmeur d'outrepasser le type d'un objet du programme, c'est-à-dire l'impossibilité de « persuader » le compilateur qu'un objet a un autre type que celui inféré. Enfin, le terme *statique* exprime que la détermination des types des éléments constituant le programme est faite préalablement à l'exécution, lors d'une première phase de la compilation. Ainsi, tout programme accepté par le compilateur et donc *a fortiori* par le typeur ne pourra générer, durant son exécution, d'erreur due à une utilisation incorrecte, en terme de types, d'un de ses composants. Afin de conserver l'inférence de types décidable, le système de typage des langages ML restreint l'ensemble des programmes considérés corrects un peu plus que l'ensemble réels de programmes corrects vis-à-vis de la sémantique dynamique de ML. Néanmoins, dans la pratique cette restriction demeure tout à fait acceptable, procurant rarement une gêne quant à l'utilisation du langage pour implanter des algorithmes. Nous verrons d'ailleurs dans cette thèse une extension de ce système en vue de réduire encore un peu cette restriction, lorsque nous aborderons le typage en présence de récursion polymorphe.

Il reste encore une façon pour un programme de se terminer prématurément sans pour autant pouvoir le détecter avec un système de types comme celui présent en standard dans ML : l'interruption par *exception* non rattrapée. Les exceptions sont un mécanisme permettant d'interrompre un calcul pour rendre compte d'une erreur ou d'une configuration anormale durant celui-ci. Par exemple, tenter de calculer $\frac{1}{0}$ ou accéder à un élément en dehors des bornes d'un tableau ne peut prétendre réussir. Lorsque le programme se trouve dans une telle situation, il génère (*lève)* alors une exception pour rendre compte de l'impossibilité de fournir un résultat valide. Si cette exception n'est pas prise en compte par le programmeur, alors le programme se termine. Dans un tel cas, on ne peut pas réellement considérer pour autant que le programme « plante » au sens exposé précédemment car il n'effectue pas d'opération illégale. Il se contente en fait de s'arrêter *avant* de risquer d'effectuer une opération illégale. Prendre en compte une exception consiste à prévoir explicitement dans le programme le cas où une telle exception surviendrait et à écrire l'algorithme devant gérer ce cas. C'est ce que nous appellerons *rattraper une exception* par un *gestionnaire d'exceptions* (en Anglais : *handler*).

Dans cette thèse, nous allons donc étudier et tenter d'apporter des solutions afin de garantir que de tels programmes soient détectés préalablement à leur exécution. Nous tenterons donc de décrire une analyse statique permettant de détecter les exceptions non rattrapées dans les programmes écrits dans le dialecte *Objective Caml* de ML.

De nombreux langages de programmation comme Ada, Modula-3, ML et Java fournissent un support natif pour les exceptions : lever une exception à un endroit du programme transfère le contrôle au *gestionnaire* pour cette exception le plus proche dans la pile des appels dynamiques de ce programme. Dans les dialectes ML, lever une telle exception est réalisé par la construction `raise` suivie du nom de l'exception, par exemple :

if $x \neq 0$ **then** $\frac{1}{x}$ **else raise** `Div_by_0`

La récupération d'une exception se fait par une construction `try ... with` suivie d'une liste de cas d'exceptions, par exemple :

```
try ... with
 | Not_found → print « Inexistant »
 | Div_by_0 → print « Division par 0 »
```

Les exceptions fournissent donc un moyen de gestion des erreurs simple et flexible dans les applications : si une exception n'est pas explicitement gérée dans une fonction par le programmeur, elle est automatiquement propagée en amont dans le graphe d'appel des fonctions jusqu'à ce qu'une fonction « sachant » que faire de cette exception soit trouvée. Si aucun gestionnaire n'est fourni pour cette exception, alors l'exécution du programme s'interrompt, exhibant ainsi précisément une configuration non prévue dans l'algorithme. Nous voyons ainsi que ce mécanisme tranche radicalement avec celui utilisé pour reporter les erreurs dans le style C traditionnel dans lequel les "valeurs impossibles" retournées par les fonctions sont la plupart du temps encodées par un pointeur nul ou l'entier -1. En effet, dans cette approche, le programmeur doit écrire une quantité de code relativement conséquente pour propager les erreurs en amont dans l'arbre d'appels des fonctions. L'expérience montre que bien souvent il est facile d'oublier de reporter une telle erreur, ce qui peut « planter » le programme bien plus tard (rendant ainsi la recherche de l'erreur nettement plus compliquée), voire ne pas affecter la terminaison du programme mais faire en sorte qu'il renvoie un résultat erroné.

Il est également possible d'utiliser les exceptions comme structure de contrôle non locale. Considérons le cas d'une recherche dans un tableau, on peut décider d'interrompre le parcours de ce tableau dès que l'on y a trouvé l'élément satisfaisant le critère de recherche. Par exemple :

```
let exists_sup_0 tbl =
  try
   for i = 0 to (Array.length tbl) - 1 do
     if tbl.(i) > 0 then raise Found
   done ;
   false
  with Found → true
```

Il est possible de réécrire cette fonction sous forme récursive afin d'éviter de lever l'exception sous la forme :

```
let exists_sup_0 tbl =
  let rec rec_exists j =
    if j >= Array.length tbl then false
    else if tbl.(j) > 0 then true else rec_exists (j + 1) in
  rec_exists 0
```

Toutefois il peut être désirable, lorsque l'efficacité devient un critère critique de l'algorithme et que la fonction considérée n'est pas *récursive terminale*, de retourner le résultat de la recherche directement par une exception plutôt que de remonter tous les appels récursifs de la fonction. Par exemple :

```
let rec exists_sup_0 = function
 | Leaf x → x > 0
 | Node (left, right) → exists_sup_0 left || exists_sup_0 right
```

peut s'avérer plus efficace, (mais moins aisée à lire à mon sens) sous la forme :

```
let exists_sup_0 tree =
  let rec rec_exists = function
    | Leaf x → if x > 0 then raise Found
    | Node (left, right) → rec_exists left ; rec_exists right in
  try rec_exists tree ; false with
    | Found → true
```

Quoi qu'il en soit, le revers d'utiliser les exceptions comme moyen de signaler les erreurs ou comme structure de contrôle réside dans le fait qu'il est facile d'oublier de rattraper une exception « au bon moment », ce qui revient à oublier de gérer une condition d'erreur ou à ne pas compléter la structure de contrôle. Les compilateurs ML ne génèrent aucune erreur ni même avertissement dans de tels cas, ainsi l'erreur ne se trouve révélée que lors de l'exécution du programme, lors d'une phase de test ou dans le pire des cas lors de l'utilisation réelle de l'application. Le test exhaustif de programme est un art très difficile ; parfois même la phase de tests simples est purement et simplement éludée. Qui plus est, les terminaisons anormales dues à des oublis de tests d'erreurs sont bien souvent dures à reproduire, relativement sporadiques et pas forcément déterministes rendant ainsi la tâche de *debuggage* extrêmement fastidieuse et délicate.

Afin de résoudre ce problème, les langages comme Modula-3 et Java demandent au programmeur de déclarer, pour chaque fonction ou méthode, l'ensemble des exceptions pouvant en sortir. Ces déclarations sont ensuite vérifiées statiquement lors du typage grâce à une analyse intra-procédurale simple. Du point de vue de l'utilisateur du langage, c'est-à-dire du programmeur, cette méthode possède deux principaux inconvénients. Tout d'abord, elle oblige le programmeur à être conscient du flot d'exceptions de son programme entier, ce qui peut représenter un nombre très conséquent de fonctions. Dans le cas d'un développement en équipe, ceci nécessite une excellente coordination et une collaboration sans faille entre les membres de chaque équipe sous peine de rencontrer de sérieux problèmes de mise en commun des composants développés séparément par chaque équipe.

D'autre part, le programmeur se trouve face à une obligation d'annoter toutes les fonctions de son source avec des ensembles d'exceptions pouvant devenir relativement conséquents. Par expérience, ceci est fastidieux et il n'est pas rare d'oublier quelques exceptions par-ci par-là, obligeant d'incessants aller-retours entre les messages d'erreur du compilateur et les sources de ses programmes.

Déclarer les exceptions pouvant s'échapper dans les signatures des fonctions ou des méthodes fonctionne correctement dans le cas du premier ordre, pour des programmes monomorphes, mais s'avère inefficace dans le cadre de la programmation d'ordre supérieur hautement polymorphe comme encouragé par les dialectes ML. Considérons l'itérateur `map` sur les listes. En Modula-3 ou en Java, le programmeur doit déclarer un ensemble E d'exceptions que la fonction argument de `map` peut lever : ainsi `map` elle-même pourra lever lors de son utilisation ce même ensemble d'exceptions. Mais si E est fixé arbitrairement, `map` ne pourra pas être appliquée à des fonctions levant des exceptions ne se trouvant pas dans l'ensemble E. La généricité (en terme d'exceptions) de `map` peut être retrouvée en prenant pour E l'ensemble de toutes les exceptions du programme, au prix d'une sévère perte de précision de l'analyse : toutes les utilisations de `map` seront considérées comme levant potentiellement n'importe quelle exception ! Pour se comporter correctement avec les fonctions d'ordre supérieur, un langage très riche pour la déclaration des ensembles d'exceptions est requis, incluant au moins du polymorphisme sur les exceptions (c'est-à-dire des variables d'ensemble d'exceptions) ainsi que l'union sur ces ensembles (nous décrirons ultérieurement plus en détail ces nécessités dans le chapitre 1). Il semble qu'un tel langage, muni d'une telle complexité pour déclarer les exceptions levées dans les programmes est au-delà de ce que le programmeur désirerait utiliser.

Ainsi, l'alternative choisie dans ce travail consiste à inférer conservativement (c'est-à-dire sans en oublier, tout en s'autorisant à en mentionner « en trop », ce qui est alors une approximation pessimiste mais néanmoins correcte), tout comme ML infère les types présents dans un programme, l'ensemble des exceptions pouvant être levées et non rattrapées à partir d'un programme non annoté par le programmeur. En d'autres termes, nous envisageons le problème d'une telle détection comme un problème de « débuggage » statique, dans lequel des techniques d'analyses statiques sont appliquées aux programmes, non pas pour les rendre plus rapides par l'intermédiaire d'une meilleure génération de code machine, mais pour les rendre plus sûrs et pointer précisément les endroits où peuvent éventuellement se produire des erreurs à l'exécution. Plus précisément, notre analyse sera basée sur un système d'effets reposant sur le système de types de ML. Cette approche fournit plusieurs avantages par rapport à celle utilisée en Modula-3/Java : elle s'intègre plus élégamment dans l'esprit d'inférence présent dans ML ; elle ne change pas le langage sous-jacent et autorise l'analyse de programmes déjà existants ; elle permet l'utilisation d'ensembles complexes puisque le programmeur n'a pas à les écrire lui-même. Pour terminer cette analyse devra être compatible avec le système de modules de ML : en effet, il semble peu raisonnable de devoir ré-appliquer l'analyse sur un programme tout entier si seules certaines unités de compilation ont changé (le temps d'analyse doit rester raisonnable).

Plusieurs analyses d'exceptions pour ML ont été proposées, certaines basées sur des systèmes d'effets, d'autres sur des analyses de flot de contrôle, certaines sur une combinaison des deux systèmes. Nous examinerons certains de ces travaux plus en détail au fur et à mesure que nous avancerons dans cette thèse, et plus particulièrement dans le chapitre 3. L'analyse qui compose le noyau de cette thèse essaye de combiner l'efficacité des systèmes d'effets et la précision des analyses de flot. Elle est basée sur des algorithmes d'unification et d'inférence en présence d'une algèbre de types non standard, algorithmes possédant un excellent temps d'exécution, afin que l'application de cette analyse soit praticable à large échelle. Parallèlement, elle reste relativement précise, permettant en particulier d'approximer non seulement l'ensemble des noms d'exceptions pouvant survenir, mais aussi les arguments que ces dernières peuvent véhiculer. Nous pensons que cette dernière caractéristique est un impératif si l'on espère analyser précisément de nombreux programmes ML.

Parallèlement à notre analyse, cette thèse présente également un cadre théorique et un algorithme d'inférence de types pour ML en présence de *récursion polymorphe*. Dans le système de types conventionnel de ML, il est interdit qu'un identificateur lié par `let` puisse être utilisé polymorphiquement à l'intérieur de sa propre définition. Cette restriction provient de l'indécidabilité de l'inférence de types qui résulte de l'ajout d'une telle caractéristique dans ML. Un lien existe entre les deux parties de cette thèse : les résultats de notre analyse peuvent être rendus plus précis si on lève cette restriction durant l'inférence des ensembles d'exceptions. Toutefois nous nous devons de garder notre analyse décidable. C'est la raison pour laquelle, nous avons exploré ce chemin, au premier abord fortement éloigné de notre préoccupation initiale, afin de donner naissance à un nouvel algorithme d'inférence autorisant l'acceptation d'une large classe de programmes utilisant la récursion polymorphe.

Plan

Le reste de cette thèse s'organise de la façon suivante. Le premier chapitre présentera plus en détail les caractéristiques que nous souhaitons pour notre analyse. Nous y décrirons également les travaux réalisés dans le domaine de la détection d'exceptions non rattrapées en évoquant les avantages et les inconvénients de chacun d'entre eux.

Le chapitre 2 consistera en une présentation des aspects sémantiques d'un langage ML avec exceptions. Nous y décrirons la syntaxe du langage que nous choisissons de traiter, à savoir un sous-ensemble de ML étendu par les exceptions. Le choix d'inclure directement les exceptions est motivé par la nature même de notre analyse. Nous verrons ensuite la sémantique dynamique d'un tel langage, c'est-à-dire sa vision « exécutable » et enfin son système de types. En effet, notre analyse se basant sur un système d'effets, nous pensons nécessaire que le lecteur ait en main toutes les notions de bases afférentes à ce genre de techniques.

En troisième chapitre, nous aborderons les premières techniques que nous avons envisagées et essayées au cours de travaux préliminaires, avant de nous tourner vers la solution finalement adoptée. Nous présenterons principalement des analyses à base de contraintes d'inclusion et de méthodes itératives, décrirons leurs principes, nos prototypes d'implantation et les conclusions que nous en avons tirées.

Dans le chapitre 4, nous aborderons le principe de notre analyse proprement dite. Une discussion informelle débutera cette section en vue de fournir au lecteur une intuition du fonctionnement de l'analyse. En particulier nous introduirons progressivement la structure de notre algèbre de types ainsi que les opérations s'y rapportant au travers d'exemples. Nous espérons ainsi exposer, de manière pédagogique, les motivations qui nous ont permises d'aboutir au système d'analyse tel que nous le décrivons dans cette thèse. Ensuite, nous verrons les règles de typage associées au système de types obtenu, ce qui constituera une description plus formelle du mécanisme d'analyse. Pour terminer, nous donnerons quelques exemples d'expressions munies de leur type afin d'illustrer l'utilisation de notre nouveau système de types pour ML, système avec lequel nous travaillerons désormais pour le restant de cette thèse.

Le chapitre 5 sera réservé à la preuve de correction de notre système de types. Il s'agira ainsi essentiellement d'un chapitre technique destiné à justifier les fondements de notre analyse.

À l'instar du précédent, le sixième chapitre de cette thèse se voudra technique et établira la correction de l'algorithme d'inférence. Ayant montré précédemment la correction du typage, nous décrirons en détail l'algorithme d'inférence de types et montrerons sa correction (ce qui établira ainsi la correction de notre *analyse*) ainsi que sa principalité (c'est-à-dire l'existence d'un type principal trouvé par notre algorithme).

Ensuite, le chapitre 7 exposera différentes extensions du système de base en vue de le rendre utilisable sur un langage réel comme *Objective Caml*. Nous y étudierons principalement les différentes stratégies possibles d'approximation des types concrets, l'adjonction de types récursifs (*cycliques*), la gestion des valeurs mutables, l'adaptation à un système de modules et enfin la prise en compte des objets.

Le huitième chapitre exposera le second résultat de cette thèse, à savoir l'inférence de types en présence de *récursion polymorphe*. Nous verrons dans un premier temps que les résultats de notre analyse peuvent être affinés avec une telle technique d'inférence de types. Ensuite nous présenterons un système de types puis un algorithme d'inférence efficace en présence de récursion polymorphe, démontrerons leur correction, examinerons leur efficacité et la manière d'intégrer ce nouveau point dans le système de types de l'analyseur décrit précédemment.

Enfin le dernier et neuvième chapitre de cette thèse sera consacré à une présentation des aspects pratiques de ce travail. Nous y exposerons quelques considérations d'ingénierie, comme les problèmes dus à l'analyse séparée (module par module), la nécessité d'une interface utilisateur pour présenter les résultats de l'analyse et enfin la gestion des exceptions pour les définitions externes au langage (*externals*). Nous clorons ce chapitre par l'examen de quelques programmes de test en terme de précision et de rapidité de l'analyse afin de se faire une idée de la qualité pratique de l'analyse développée tout au long de ce travail.

Suggestion de parcours de cette thèse

Cette thèse est volontairement découpée en de nombreux chapitres afin de fournir une structure claire entre les différents aspects du travail étudié. Nous pouvons ici suggérer au lecteur un parcours possible au travers de ce document en fonction de ce qu'il recherche plus précisément.

Tout d'abord, deux résultats étant présentés, à savoir *l'analyse d'exceptions* et *l'inférence de types en présence de récursion polymorphe*, le lecteur déjà familier avec les systèmes de types à la ML et uniquement intéressé par le second aspect pourra directement consulter le chapitre 8.

La lecture du premier chapitre permettra au lecteur de comprendre les motivations du travail présenté. Dans le cas où le lecteur est déjà familier avec le langage ML et la notion d'exception, il pourra s'affranchir du second chapitre. Si le lecteur est principalement intéressé par l'analyse d'exceptions en elle-même, il pourra sauter le troisième chapitre ayant pour unique but d'exposer les autres solutions que nous avons envisagées et tentées. Le chapitre 4 présente le noyau du présent travail : le lecteur y trouvera ainsi le détail de l'analyse. La lecture de la section 5 est plus particulièrement conseillée aux experts. La première partie du chapitre 6 présente l'algorithme d'inférence de types et donc va de paire avec la présentation du système de types. Toutefois la seconde partie de ce chapitre pourra être survolée, étant elle aussi relativement technique. Pour terminer, le septième chapitre est destiné aux lecteurs ayant compris le principe de base et curieux d'examiner les problèmes d'implantation au sein d'un langage de taille réelle.

Chapitre 1

Exigences

Réservons les premières lignes de cette thèse à une présentation détaillée des caractéristiques que nous jugeons nécessaires pour obtenir une analyse d'exceptions pour ML efficace et utilisable dans le monde réel du développement d'applications. Montrons également que ces *desiderata* vont au-delà de ce qui peut être exprimé par le mécanisme de déclaration d'exceptions utilisé en Modula-3 ou en Java. Comme nous le verrons ultérieurement, d'autres analyses antérieures ont tenté de satisfaire ces contraintes, mais aucune d'entre elles, à notre connaissance, n'a réussi à proposer de solutions à tous ces problèmes en même temps.

1.1 Les fonctions d'ordre supérieur

Le comportement, en terme d'exceptions, des fonctions d'ordre supérieur dépend des exceptions pouvant être levées par leurs arguments fonctionnels. Nous entendons par *fonction d'ordre supérieure* une fonction dont l'un des arguments est lui-même une fonction. Nous voyons ainsi pointer la nécessité d'une forme de polymorphisme sur les ensembles d'exceptions s'échappant afin de pouvoir analyser précisément de telles fonctions.

Considérons le cas de l'itérateur `map` mentionné en introduction. Celui-ci prend en argument une fonction `f`, une liste `l` et applique cette fonction `f` à tous les éléments de la liste `l` afin de rendre la liste composée des résultats de ces applications. Cet itérateur, fonction d'ordre supérieur, s'écrit donc :

```
let rec map f = function
  | [] → []
  | h :: q → (f h) :: map f q
```

Une application `map f l` peut ainsi lever n'importe quelle exception que l'argument `f` peut lever. Si l'on choisit d'écrire $\tau \xrightarrow{\varphi} \tau'$ pour le type annoté des fonctions du type τ vers le type τ' et dont l'ensemble d'exceptions potentiellement levées et non rattrapées est φ, alors le comportement « exceptionnel » de `map` est capturé par le schéma de types suivant :

$$\texttt{map} : \forall \alpha, \beta, \varphi.\, (\alpha \xrightarrow{\varphi} \beta) \xrightarrow{\emptyset} (\alpha\ \texttt{list} \xrightarrow{\varphi} \beta\ \texttt{list})$$

dans lequel α, β sont des variables prenant leurs valeurs parmi les types et φ parmi les ensembles d'exceptions. En règle générale, l'ensemble des exceptions non rattrapées d'une fonction d'ordre supérieur est une combinaison $\varphi_1 \cup \ldots \cup \varphi_n \cup \{C_1; \ldots; C_n\}$ dans laquelle les φ_i sont des variables représentant les exceptions s'échappant des arguments fonctionnels et les C_j sont des constantes d'exceptions provenant du corps même de la fonction. Ainsi la composition de fonctions $\lambda f.\lambda g.\lambda x.f(g(x))$ aura pour schéma de types :

$$\forall \alpha, \beta, \gamma, \varphi, \psi.\, (\alpha \xrightarrow{\varphi} \beta) \xrightarrow{\emptyset} (\gamma \xrightarrow{\psi} \alpha) \xrightarrow{\emptyset} \gamma \xrightarrow{\varphi \cup \psi} \beta$$

dans lequel φ provient de l'argument fonctionnel `f` et ψ provient de `g`. L'ensemble des exceptions pouvant alors s'échapper finalement est l'union de ces deux ensembles. On notera la présence de l'ensemble vide étiquetant les seconde et quatrième flèches du type, dénotant ainsi l'impossibilité pour la composition de lever d'exceptions propres à ses calculs intermédiaires. Nous pouvons aisément exhiber un cas où cet ensemble de constantes d'exceptions est non vide. Ainsi la version modifiée (et totalement inutile) de `map` :

```
let rec mapnulle f l =
 raise Error ;
 match l with
  | [] → []
  | h :: q → (f h) :: mapnulle f q
```

aura pour schéma de types

$$\texttt{mapnulle} : \forall \alpha, \beta, \gamma, \varphi. \, (\alpha \xrightarrow{\varphi} \beta) \xrightarrow{\emptyset} \alpha \; \texttt{list} \xrightarrow{\varphi \cup \{\texttt{Error}\}} \beta \; \texttt{list}$$

dans lequel il apparaît clairement qu'un calcul intermédiaire dans le corps de `mapnulle` peut lever une exception sans la rattraper (l'exception `Error` en l'occurrence).

Étant donné l'usage fréquent fait des fonctions d'ordre supérieur dans les programmes ML, une analyse d'exceptions pour ce langage se doit de les traiter avec précision, comme le suggère les schémas de types montrés ci-dessus.

Un problème similaire survient lorsque l'on mémorise des fonctions dans des structures de données comme des listes, des tableaux ou des tables de hachage (par exemple afin de faire des tables de *callbacks*). Considérons l'exemple suivant :

```
let f1 x = raise Exc1
let f2 x = raise Exc2
let f3 x = raise Exc3
let tbl = [f1 ; f2]
(random_extract tbl) 1
```

L'analyse d'exceptions doit être capable de garder trace de l'union des exceptions pouvant être levées par chacune des fonctions présentes dans la structure de données. Il ne serait pas acceptable de dire que n'importe quelle exception peut survenir en appliquant une fonction extraite de cette structure. En l'occurrence, sur l'exemple ci-dessus nous souhaitons que l'ensemble des exceptions pouvant survenir dans la dernière expression soit uniquement composé de $\{\texttt{Exc}_1; \texttt{Exc}_2\}$, la fonction $\texttt{f}_3$ ne faisant pas partie de la liste à laquelle on accède (aléatoirement, par la fonction non explicitement décrite `random_extract`).

1.2 Les exceptions valeurs de première classe

En ML et en Java, les exceptions sont des valeurs de première classe : les valeurs d'exceptions peuvent être construites par le résultat d'un calcul et être échangées entre des fonctions (c'est-à-dire passées en argument) avant d'être levées. Considérons l'exemple artificiel suivant :

```
let test exn = try raise(exn) with C → 0
```

Le comportement en terme d'exceptions de cette fonction est tel que `test exn` lève l'exception contenue dans l'argument `exn`, sauf lorsque cette exception est en réalité l'exception de nom `C`. Ainsi dans ce dernier cas, aucune exception ne s'échappe de l'application de la fonction `test`. Nous souhaitons que notre analyse d'exceptions soit suffisamment précise pour capturer ce comportement.

Il est vrai que l'utilisation des exceptions comme valeurs de première classe est assez rare, même si l'on en trouve quand même quelques exemples dans de véritables programmes écrits en ML. Toutefois, il existe un idiome de programmation important dans lequel les valeurs d'exceptions sont utilisées de cette manière : la *finalisation*. Il s'agit d'une technique permettant d'intercepter temporairement une exception en vue d'effectuer un traitement sans pour autant absorber cette exception. Considérons l'exemple suivant :

```
let f x = try g(x)
  with
   | E → 0
   | exn → code de finalisation ; raise(exn)
```

Si l'on considère que la fonction g peut lever les exceptions E et E', nous voulons que l'analyse reconnaisse que la variable d'exception `exn` ne peut prendre que la valeur E'. Ainsi, l'expression `raise(exn)`, qui lève à nouveau l'exception après finalisation, ne peut que lever E', impliquant que `f` ne peut également laisser s'échapper que cette exception E'.

L'utilisation la plus simple et la plus évidente de la finalisation consiste à enrober un appel de fonction pouvant générer des erreurs, mais devant toutefois effectuer *du ménage* avant d'avorter. Ceci peut consister en fermer des fichiers ouverts, retirer des verrous sur des objets accédés en exclusion mutuelle, afficher des messages d'erreurs, etc.... Ainsi, il n'est pas rare de voir des programmes ML dont le point d'entrée se résume à la forme suivante :

```
try main () with e → printexc e ; cleanexit ()
```

dans lequel la fonction `printexc` est chargée d'imprimer le nom de l'exception inattendue, et `cleanexit` une fonction que le programmeur a défini pour terminer proprement son application.

1.3 Les arguments d'exceptions

Les exceptions de ML peuvent optionnellement être paramétrées par des arguments, tout comme les autres constructeurs de *type somme* (ce que l'on assimilerait à une combinaison type énuméré et union dans des langages à la C). Ainsi, de la même façon que l'on peut définir les types :

```
type couleur = Rouge | Orange | Vert
type feu = Fixe of couleur | Clignotant of couleur
```

il est possible de définir les exceptions :

```
exception Sens_unique
exception Interdiction_de of string
```

Les constructeurs de chacune des dernières déclarations colportent une information supplémentaire permettant de raffiner le sens de leur constructeur respectif.

De la même façon qu'il est possible de discriminer (grâce à la construction `match ... with ...`) par filtrage (en Anglais, *pattern-matching*) sur les arguments des types somme, on peut tester l'argument d'un constructeur d'exceptions en utilisant le filtrage (grâce à la construction `try ... with`) au sein d'un gestionnaire d'exceptions (en Anglais, *handler*), permettant ainsi de ne rattraper que certaines exceptions, colportant certains arguments. Considérons l'exemple suivant :

```
exception Failure of string
let f x = if ... then ... else raise(Failure "f")
let g x = try f(x) with Failure "f" → 0
```

Une analyse d'exceptions qui garderait trace du constructeur de tête de l'exception (c'est-à-dire ici `Failure`) mais pas de ses arguments (c'est-à-dire la chaîne de caractères `"f"`) ne pourrait pas analyser cet exemple de manière suffisamment précise. En effet, cette analyse mémoriserait le fait que la fonction `f` peut lever l'exception `Failure`, considérant du même coup que l'application `f(x)` dans `g` peut également laisser s'échapper `Failure` avec n'importe quel argument. Puisque le gestionnaire d'exceptions ne rattrape que `Failure "f"`, l'analyse conclurait que g peut lever l'exception Failure, alors qu'en réalité aucune exception ne peut s'échapper de cette fonction.

Ce manque de précision peut être considéré comme peu important et dû à un mauvais style de programmation. En effet, on peut toujours arguer que le programmeur aurait dû déclarer une exception constante spécifique Failure_f pour rapporter l'erreur survenue dans la fonction f plutôt que de s'appuyer sur l'exception générale et « multi-usage » Failure. Toutefois, des fragments de codes similaires à celui examiné précédemment apparaissent dans des applications déjà existantes écrites en Caml, applications que nous aimerions pouvoir analyser aussi précisément que les autres. Par exemple, l'interface entre Caml et les appels systèmes d'Unix utilise le schéma suivant pour rapporter les codes d'erreurs survenus du côté Unix :

```
type unix_error = EACCES | ENOENT | ENOSPC | ...
  (* type énuméré comportant 67 constructeurs
     représentant les codes d'erreur Unix *)
exception Unix_error of unix_error
```

Ceci permet au code de l'utilisateur d'intercepter toutes les erreurs en provenance de la partie Unix de son programme en une seule fois (try ... with Unix_error(_) → ...) mais aussi de rattraper seulement certaines de ces erreurs (try ... with Unix_error(ENOENT) → ...). Remplacer Unix_error par 67 exceptions distinctes, une pour chaque code d'erreur, rendrait le premier style d'interception réellement pénible. Il est tout à fait désirable que l'analyse arrive à montrer que certaines exceptions Unix_error avec des arguments représentant les erreurs les plus courantes (par exemple Unix_error(ENOENT) qui signifie « No such file or directory ») sont bien gérées par l'application et ainsi ne peuvent en aucun cas s'échapper accidentellement, tandis que d'autres, plus rares, pourront elles mettre fin prématurément à l'exécution du programme.

Le problème des exceptions paramétrées par des arguments est rendu encore plus crucial par le fait qu'il existe (au moins dans la bibliothèque standard de Caml) des fonctions prédéfinies permettant de lever des exceptions d'usage général, comme Failure que nous avons vue précédemment. Ainsi, le style d'exemple que nous avons étudié plus haut avec Failure apparaît plus souvent sous la forme suivante :

```
let failwith msg = raise (Failure msg)
let f x = if ... then ... else failwith("f")
let g x = try f(x) with Failure "f" → 0
```

Une analyse d'exceptions précise requiert non seulement le suivi de la chaîne de caractères "f" en tant qu'argument immédiat du constructeur d'exceptions Failure, mais aussi en tant qu'argument passé à la fonction failwith. Ainsi, une telle analyse doit également inclure une part d'analyse de flot non uniquement limitée aux valeurs d'exceptions. En règle générale, nous chercherons donc à conserver trace des valeurs contenues dans tout type de données. Par exemple, nous souhaitons être capables de dire que l'expression if ... then 13 else 42 est de type entier, pouvant contenir les valeurs 13 *ou* 42. De même, Clignotant (Orange) sera analysé comme une valeur de type feu (en accord avec la définition de type donnée précédemment en exemple) dont l'argument est une valeur de type couleur valant Orange.

1.4 Rapidité supérieure à l'analyse de flot

Toutes les caractéristiques que nous avons énoncées jusqu'à présent pour une analyse d'exceptions efficace permettent de penser aux techniques d'analyse de flot de contrôle (*control-flow analysis)* comme celles à base de n-CFA réalisée par Shiver [55] ou la SBA de Heintze [17]. En effet, l'analyse de flot de contrôle fournit une approximation de l'ensemble des valeurs qui peuvent se propager en chaque point du programme. Il est alors relativement simple de l'étendre afin d'approcher également l'ensemble des exceptions pouvant s'échapper en chaque point du programme puisqu'elle fournit une approximation de l'ensemble des valeurs résultats. On peut également envisager que l'analyse d'exceptions fonctionne comme une seconde passe d'analyse de flot de données, exploitant les résultats de l'analyse de flot de contrôle [68], bien que ceci résulte en une perte de précision car ce même flot de contrôle peut être déterminé plus précisément si nous avons des informations sur le flot d'exceptions. En effet, considérons le programme suivant :

```
try raise Foo with
  | Bar → raise Gee
  | _ → 0
```

Si nous déterminons *a priori* le flot de contrôle de cet exemple, alors nous verrons que la première branche du gestionnaire peut lever quelque chose (`Gee` en l'occurrence). Si nous avions su reconnaître que seul `Foo` était susceptible de s'échapper de l'expression entourée par le gestionnaire, alors il nous aurait été possible de déterminer que le premier cas de ce gestionnaire est inutilisé, et donc que l'exception `Gee` ne risque pas de s'échapper.

Notons toutefois que ce style d'analyse fournit néanmoins une relativement bonne approximation des ensembles de valeurs tant que l'on ne cherche pas à traiter les arguments des exceptions.

Comme nous le détaillerons plus précisément dans la section 3, nos premières implémentations d'analyseurs d'exceptions pour Objective Caml étaient basées sur des analyses de flot, initialement la 0-CFA, puis le *polymorphic splitting* de Jagannathan et Wright [22]. Nos résultats pratiques avec de telles techniques ont été mitigés. La précision de l'analyse était satisfaisante (au moins dans le cas du *polymorphic splitting*, tant que l'on ne cherche pas à approximer les arguments des exceptions paramétrées), mais la vitesse laissait à désirer. En particulier, nous avons observé des comportements quadratiques sur plusieurs exemples permettant de supposer de grandes difficultés pour appliquer ces techniques à de très gros programmes. Bien que de nombreuses techniques aient été développées afin d'accélérer les analyses basées sur des contraintes d'inclusion telles que la CFA et la SBA [7, 12, 10, 42], le problème de savoir si elles peuvent être appliquées à des programmes de l'échelle de 100000 lignes reste encore ouvert actuellement.

Chapitre 2

Une présentation de ML

RAISONNER SUR ML TOUT ENTIER (et par extension sur Objective Caml) pour en donner une description formelle est une tâche relativement difficile étant donnée l'étendue des langages. Aussi allons nous, dans un premier temps, nous restreindre à un sous-ensemble significatif de ce style de langage dans la présentation qui suit.

Nous allons plus particulièrement examiner la sémantique dynamique de ce langage, ensuite nous nous intéresserons à son système de types. Cette présentation n'a pas pour but de remplacer les nombreux écrits déjà existants sur le sujet [3, 65], mais plutôt de fournir au lecteur débutant une présentation assez complète des notions de base de tels langages dans le but de mieux appréhender la suite de cette thèse. Nous en profiterons au passage pour introduire les notations et les définitions que nous utiliserons par la suite.

Les extensions nécessaires afin d'adapter notre analyse à un langage complet seront étudiées ultérieurement (cf. chapitre 7) sur la base du mini-langage que nous allons maintenant introduire.

2.1 La syntaxe du langage

Le langage source que nous considérons est donc un sous-ensemble de ML comportant les entiers et les exceptions comme structures de données de base. Le choix d'inclure les exceptions dans cette présentation est directement motivé par la nature de notre analyse.

Par ailleurs, le langage possède, outre les constructions de base (comme les fonctions, l'application et la liaison d'identificateurs), la capacité de lever et d'intercepter des exceptions, ainsi que le filtrage (*pattern matching*). Ces trois dernières constructions sont rendues nécessaires par la nature même de notre analyse, particulièrement concernée par la gestion des exceptions en règle générale et donc par le filtrage pouvant apparaître au sein des gestionnaires servant à les intercepter.

Notre langage se présente donc sous la forme d'expressions (notées a) et de motifs de filtrages (notés p) engendrés par la grammaire suivante :

Expressions :

a	$::= x$	identificateur
	$\mid i$	constante entière
	$\mid \lambda x.a$	abstraction (fonction)
	$\mid a_1(a_2)$	application
	$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	liaison `let`
	$\mid \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3$	filtrage
	$\mid C \mid D(a)$	constructeur d'exceptions
	$\mid \texttt{try}\ a_1\ \texttt{with}\ x \to a_2$	gestionnaire d'exceptions

Motifs :

p	$::= x$	« variable » de motif
	$\mid i \mid C$	constante de motif
	$\mid D(p)$	motif construit

Les constructeurs d'exceptions (similaires à ceux de types concrets comme nous le verrons ultérieurement) se déclinent en deux catégories : les constructeurs constants que l'on peut considérer comme des valeurs de types énumérés à la C ou à la Java, et les constructeurs paramétrés à mettre en parallèle avec les unions discriminées en C . La possibilité de paramétrer les constructeurs autorise ainsi les valeurs d'exceptions à colporter une information complémentaire pouvant servir à préciser ce constructeur.

La construction $\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3$ effectue un filtrage sur la valeur de a_1 ; si cette valeur est filtrée par le motif p alors la branche a_2 est évaluée ; sinon a_3 est évaluée sous l'hypothèse que l'identificateur x est lié à la valeur de a_1.

Considérons les valeurs comme des termes du premier ordre sans variables et les motifs comme des termes, également du premier ordre, mais pouvant comporter des variables. On dira qu'une valeur est *filtrée par un motif* s'il existe une substitution envoyant les variables du motif vers des valeurs tel que cette substitution appliquée au motif rende un terme égal à celui de la valeur. Les liens engendrés par cette substitution sont alors mémorisés dans l'environnement d'évaluation utilisé pour calculer le résultat de l'expression se trouvant à droite du motif. Considérons l'expression suivante munie de quatre cas pour laquelle on suppose que v a la valeur `D (D 1)` :

```
match v with
 | D (D (D x)) → cas₁
 | D (D 1) → cas₂
 | D (D x) → cas₃
 | D x → cas₄
```

Dans un premier temps, considérons chacun des cas séparément. Le premier motif ne filtre pas la valeur v car il est impossible de trouver une substitution affectant x tel qu'on ait l'égalité entre termes. Le second motif filtre trivialement v puisqu'il y a égalité triviale entre les deux termes. La valeur v est également filtrée par le troisième filtre puisque l'on peut trouver la substitution $\varphi = \{\texttt{x} \leftarrow 1\}$ telle que $\varphi(\texttt{D}(\texttt{D x})) = \texttt{D}(\texttt{D 1})$. Quant au dernier motif, la substitution $\varphi = \{\texttt{x} \leftarrow \texttt{D}(1)\}$ nous prouve qu'il filtre aussi v.

Nous avons considéré chacun des cas du filtrage séparément. Dans une construction `match ... with`, nous avons vu que l'ordre d'apparition des branches était significatif : la première branche dont le motif filtre la valeur est sélectionnée pour calculer le résultat de l'expression toute entière par évaluation de l'expression se trouvant à droite de ce motif. Ainsi dans l'exemple ci-dessus, la seconde branche sera choisie à l'exclusion de toutes les autres et lors de l'évaluation de cas_2 nous aurons l'hypothèse supplémentaire que l'identificateur `x` est lié à la valeur 1.

Il est possible de réaliser des filtrages à branches multiples en cascadant plusieurs filtrages de base en série. Ainsi,

```
match ... with
  | 0 → ...
  | 1 → ...
  | i → ...
```

s'écrira sous la forme :

```
match ... with
  | 0 → ...
  | x →
    match x with
      | 1 → ...
      | i → ...
```

La construction try a_1 with $x \to a_2$ évalue l'expression a_1 et retourne la valeur de a_1 si aucune exception n'est levée pendant l'évaluation de a_1. Si par contre, une exception est levée, alors sa valeur est liée à l'identificateur x et l'expression a_2 est évaluée et retournée comme résultat.

Nous remarquerons que notre langage ne comporte pas de forme syntaxique pour la levée d'exceptions. À la place, nous préférons considérer qu'il existe une fonction prédéfinie, raise, qui lève l'exception qui lui est fournie en argument.

Il est également possible d'intercepter seulement certaines exceptions par combinaison d'un try ... with et d'un filtrage. Ainsi le programme suivant intercepte toutes les exceptions de la forme Foo ou Bar(...) :

```
try ... with
  exn →
    match exn with
     | Foo → ...
     | Bar x → ...
     | e → raise e
```

2.2 Sémantique dynamique du langage

Nous allons maintenant donner le sens « exécutable » des constructions du langage, autrement dit la façon de laquelle chacune s'évalue en une valeur lorsqu'un programme s'exécute. À l'instar de [65], nous allons utiliser ici une sémantique dite « à réduction » qui définit la réécriture des expressions du programme jusqu'à l'obtention d'une valeur. D'autres formalismes existent pour présenter la sémantique d'un langage, principalement répartis en deux catégories : les *sémantiques opérationnelles* et les *sémantiques dénotationnelles* .

La première catégorie (à laquelle le formalisme que nous utilisons appartient) comportent également la « sémantique opérationnelle structurée » (SOS) (également appelée « sémantique naturelle ») qui s'appuie sur des jugements d'évaluation et utilise un environnement d'évaluation. Ce formalisme bien que très simple et très proche de l'exécution réelle du programme dans une machine, ne se prête pas aisément à la modélisation des exceptions ce qui explique notre préférence pour une sémantique par « réécriture » .

La seconde catégorie de présentations de la sémantique, dites « dénotationnelles » nécessite en règle générale l'utilisation de principes théoriques plus complexes permettant de donner un sens aux objets du langage en les plongeant dans des structures mathématiques. L'étude des propriétés sémantiques du langage passe alors par l'étude des propriétés de ces structures.

Une présentation opérationnelle de la sémantique d'un langage telle que celle que nous avons choisi de suivre repose sur la notion de *contexte d'évaluation* qui représente intuitivement les « trous » possibles où une réduction d'expression en valeur peut être effectuée à chaque étape de l'évaluation. Un programme s'évaluera ainsi correctement si l'on peut toujours trouver un tel « trou » tant que le programme n'a pas été réécrit sous forme d'une valeur finale. Dualement, s'il est toujours possible de trouver un contexte d'évaluation alors le programme bouclera, et s'il devient impossible d'en trouver un alors que le programme n'a pas été totalement réécrit sous forme d'une valeur alors ce programme sera dit « bloqué » et son exécution se terminera sur un échec. Ce dernier cas correspond intuitivement à un programme incorrect dont l'exécution mène à un « plantage ».

2.2.1 Les éléments sémantiques du langage

Puisque le résultat d'un programme consiste en une *valeur* issue de l'évaluation des expressions de ce programme, nous allons commencer par définir la notion de valeur, de résultat et de contexte d'évaluation en énonçant les grammaires qui les engendrent.

Valeurs :

$$
\begin{array}{lll}
v & ::= i & \text{valeur entière} \\
 & \mid C & \text{valeur d'exception constante} \\
 & \mid D(v) & \text{valeur d'exception paramétrée} \\
 & \mid \lambda x.a & \text{valeur fonctionnelle} \\
 & \mid \texttt{raise} &
\end{array}
$$

Résultats d'évaluation :

$$
\begin{array}{lll}
r & ::= v & \text{terminaison normale (valeur)} \\
 & \mid \texttt{raise}\ v & \text{résultat d'erreur} \\
 & & \text{(exception non rattrapée)}
\end{array}
$$

Contextes d'évaluation :

$$
\begin{array}{lll}
\Gamma & ::= [\,] & \text{contexte racine} \\
 & \mid \Gamma(a) & \text{application - évaluation de la fonction} \\
 & \mid v(\Gamma) & \text{application - évaluation de l'argument} \\
 & \mid \texttt{D}(\Gamma) & \text{contexte de constructeur paramétré} \\
 & \mid \texttt{let}\ x = \Gamma\ \texttt{in}\ a & \text{contexte } \texttt{let} \\
 & \mid \texttt{match}\ \Gamma\ \texttt{with}\ p \to a_2 \mid x \to a_3 & \text{contexte } \texttt{match} \\
 & \mid \texttt{try}\ \Gamma\ \texttt{with}\ x \to a & \text{contexte } \texttt{try}
\end{array}
$$

Les valeurs légales en lesquelles un programme peut s'évaluer sont donc des constantes entières, des constructeurs d'exceptions, des fonctions (puisque ML possède nativement celles-ci comme valeurs de première classe) et enfin la primitive `raise` qui n'est en fait qu'un cas particulier de fonction puisqu'elle se trouve prédéfinie dans l'environnement du langage.

Les résultats possibles pour un programme sont répartis en deux catégories : les résultats « normaux » et les résultats d'exception ou « exceptionnels », qui correspondent justement à la levée d'une exception non interceptée. Intuitivement, le premier cas correspond à une exécution du programme qui a su se terminer correctement (ce qui ne signifie pas pour autant que le résultat rendu soit celui attendu par l'utilisateur qui a pu écrire un algorithme au comportement « exécutable » correct mais ne répondant pas au problème initial). Quant à la seconde alternative, elle correspond à une exécution du programme impossible à poursuivre.

Les contextes d'évaluation permettent de rendre compte de la stratégie d'évaluation du langage. Ainsi, les second et troisième contextes montrent que le langage possède une sémantique d'appel de fonction par valeur qui évalue d'abord la fonction à appliquer, puis l'argument de celle-ci avant de finalement calculer le résultat de l'application. De même, le contexte `let` exprime que l'expression dont la valeur doit être liée à l'identificateur `x` est évaluée préalablement à l'évaluation de l'expression résultat. Un tel comportement implique par exemple que dans $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$, si l'évaluation de l'expression a_1 ne termine pas, alors l'évaluation de l'expression `let` toute entière ne terminera pas, ce qui est en opposition avec une stratégie d'évaluation paresseuse dans laquelle a_1 serait évaluée seulement si sa valeur était nécessaire pour l'évaluation de a_2 (ainsi dans le cas ci-dessus, la valeur de `x` n'étant pas utilisée dans a_2, l'évaluation de l'expression `let` terminerait quand même).

2.2.2 Les règles de réduction

Les définitions ci-dessus étant présentées, nous pouvons désormais énoncer les règles qui régissent la réécriture des expressions en valeurs. Chacune de ces règles est orientée et définit la transformation de son membre gauche en son membre droit.

$$
\begin{array}{rcll}
(\lambda x.a)(v) & \stackrel{*}{\rightarrow} & a\{x \leftarrow v\} & (2.1)\\
\texttt{let } x = v \texttt{ in } a & \stackrel{*}{\rightarrow} & a\{x \leftarrow v\} & (2.2)\\
\texttt{match } v \texttt{ with } p \rightarrow a_2 \mid x \rightarrow a_3 & \stackrel{*}{\rightarrow} & \sigma(a_2) \text{ si } \sigma = M(v,p) \text{ est défini} & (2.3)\\
\texttt{match } v \texttt{ with } p \rightarrow a_2 \mid x \rightarrow a_3 & \stackrel{*}{\rightarrow} & a_3\{x \leftarrow v\} \text{ si } M(v,p) \text{ n'est pas défini} & (2.4)\\
\texttt{try } v \texttt{ with } x \rightarrow a_2 & \stackrel{*}{\rightarrow} & v & (2.5)\\
(\texttt{raise } v)(a) & \stackrel{*}{\rightarrow} & \texttt{raise } v & (2.6)\\
(\lambda x.a)(\texttt{raise } v) & \stackrel{*}{\rightarrow} & \texttt{raise } v & (2.7)\\
D(\texttt{raise } v) & \stackrel{*}{\rightarrow} & \texttt{raise } v & (2.8)\\
\texttt{let } x = \texttt{ raise } v \texttt{ in } a & \stackrel{*}{\rightarrow} & \texttt{raise } v & (2.9)\\
\texttt{match raise } v \texttt{ with } p \rightarrow a_2 \mid x \rightarrow a_3 & \stackrel{*}{\rightarrow} & \texttt{raise } v & (2.10)\\
\texttt{try raise } v \texttt{ with } x \rightarrow a_2 & \stackrel{*}{\rightarrow} & a_2\{x \leftarrow v\} & (2.11)\\
\Gamma[a] & \stackrel{*}{\rightarrow} & \Gamma[a'] \text{ si } a \stackrel{*}{\rightarrow} a' & (2.12)
\end{array}
$$

La fonction de filtrage M est définie de la façon suivante :

$$
M(v,x) = \{x \leftarrow v\} \qquad M(i,i) = id \qquad M(C,C) = id \qquad M(D(v),D(p)) = M(v,p)
$$

Ces règles mettent en évidence la propagation des exceptions levées au cours de l'évaluation. Nous voyons en effet qu'une exception non rattrapée est répercutée dans l'arbre d'évaluation de l'expression sous forme d'un abandon de l'évaluation des sous-expressions. Seule une construction `try ... with` peut permettre l'absorption de cette exception et ainsi la reprise de l'évaluation sous une autre forme que l'erreur.

Quelques exemples Examinons quelques programmes utilisant les exceptions et les réductions qui en découlent afin de bien comprendre ce mécanisme fondamental pour la compréhension de la suite de ce travail. Soit deux constructeurs d'exceptions `C` (constructeur constant) et `D of int` (constructeur paramétré par une valeur entière).

Tout d'abord, un exemple trivial : `raise C`. Le constructeur `C` étant une valeur d'exception et la fonction `raise` une valeur fonctionnelle prédéfinie, ce programme se réduit immédiatement par la règle (2.1). La définition de `raise` étant interne au langage, et sa spécification étant connue comme levant l'exception qui lui est passée en argument, ce programme termine alors son évaluation en l'exception non rattrapée `C`, c'est à dire sous la forme `raise C`.

Considérons maintenant le programme suivant : `let x = raise C in raise D`. Nous sommes alors dans le cas de la règle de réduction (2.9). Nous voyons alors que l'évaluation du membre gauche du `let` lève une exception avant même que l'évaluation du membre droit ne débute. Aussi ce programme se réduit-il également en `raise C`.

Après avoir vu comment lever des exceptions, intéressons-nous maintenant aux gestionnaires d'exceptions, c'est-à-dire la façon de les intercepter. Notre premier exemple à ce propos se contente de positionner un tel gestionnaire autour d'une expression ne levant aucune exception : `try 1 with exn → raise exn`. L'expression 1 étant directement une valeur et levant donc aucune exception, nous nous trouvons dans le cas d'application de la règle (2.5). Il s'ensuit que ce programme se réduit en 1 qui est bien une valeur.

Modifions l'exemple précédent de la façon suivante : `try raise C with exn → raise exn`. L'expression protégée se réduit, comme vu précédemment dans le premier exemple, par la règle (2.1) en `raise C`. Aussi nous trouvons-nous dans le cas d'application de la règle (2.11) et notre expression se réduit donc en l'expression (`raise` exn)$\{exn \leftarrow C\}$. Cette dernière se réduit alors à son tour par la même règle (2.1) en `raise C`, terminant l'exécution du programme par une exception non rattrapée `C`.

Pour terminer l'introduction de ces exemples, nous examinons maintenant un programme plus complexe utilisant le rattrapage sélectif des exceptions. Nous considèrerons pour cet exemple que notre langage est muni d'une conditionnelle `if` avec le comportement habituellement attendu pour une telle construction. Soit le programme :

```
try
  try if raise (D(42)) then raise (D(43)) else 0 with
    a → match a with
          | C → 1
          | b → match b with
                    | D (1) → 2
                    | c → raise c
with
  d → match d with
         | D e → e
         | f → raise f
```

Sans entrer dans le détail de toutes les règles de réduction utilisées afin d'éviter que cet exemple ne prenne plusieurs pages, nous voyons que l'expression `D`(42) se réduit en la valeur `D`(42) en deux coups de règle (2.12). Ainsi, l'expression `raise` (`D`(42)) se réduit-elle par (2.1) en la forme `raise` (`D`(42)) avant même que la suite de la conditionnelle ne soit évaluée. Nous nous trouvons alors dans le cas d'application de la règle (2.11) autorisant une réduction en l'expression :

```
(match a with
          | C → 1
          | b → match b with
                    | D (1) → 2
                    | c → raise c) { a ← D(42) }
```

Nous appliquons alors deux fois la règle (2.4) (la fonction de filtrage M n'étant pas définie sur chacun des premiers motifs) et déduisons que cette expression se réduit en la nouvelle expression `raise` c elle-même réduite par (2.1) en la forme `raise` (`D`(42)).

Ainsi, si nous remontons au gestionnaire extérieur, nous voyons que la règle de réduction s'appliquant est à nouveau la règle (2.11) dans laquelle v est `D`(42). Nous en déduisons que la réduction se poursuit en l'expression :

```
(match d with
         | D e → e
         | f → raise f) { d ← D(42) }
```

Puisque la fonction M est cette fois-ci définie (le motif $\mathtt{D}(e)$ filtre bien la valeur $\mathtt{D}(42)$), nous appliquons alors la règle (2.3) obtenant ainsi de M la substitution $\sigma = \{e \leftarrow 42\}$, substitution que nous appliquons à l'expression e en vertue de notre règle (2.3) pour obtenir l'expression $e\{e \leftarrow 42\}$, c'est-à-dire 42 qui est bien une valeur.

Ainsi le résultat de l'évaluation du programme tout entier que nous venons d'examiner se termine-t'elle sur la valeur 42.

2.3 Typage du langage

Nous venons de voir comment associer une valeur à une expression. Nous allons maintenant chercher à associer un *type* à chacune des expressions de notre langage. Le système de types utilisé dans ML est celui de Milner [4, 35, 58] qui permet à ML de bénéficier de *polymorphisme*, c'est-à-dire qui autorise une expression à posséder plusieurs types.

2.3.1 L'algèbre de types

Soit un ensemble `Btypes` de types de base (par exemple `int` pour le type des valeurs entières, `float` pour celui des nombres flottants, `bool` pour les valeurs de vérité, `exn` pour les exceptions) et un ensemble `Vars` infini de variables de type. Dans la présentation qui suit, nous restreindrons volontairement l'ensemble des types de base aux entiers et aux exceptions.

$$\begin{array}{lll} \alpha, \beta, \gamma & \in \mathtt{Vars} & \text{variables de type} \\ \iota & \in \mathtt{Btypes} = \{\ \mathtt{int}, \mathtt{exn}\ \} & \text{types de base} \end{array}$$

Nous définissons *l'algèbre des types*, notés τ comme étant les termes engendrés par la grammaire suivante :

$$\begin{array}{lll} \tau & ::= \alpha & \text{variable de type} \\ & \mid \mathtt{int} & \text{type des valeurs entières} \\ & \mid \mathtt{exn} & \text{type des exceptions} \\ & \mid \tau_1 \to \tau_2 & \text{type des fonctions d'argument de type } \tau_1 \text{ et de résultat de type } \tau_2 \end{array}$$

On note $\mathcal{L}(\tau)$ l'ensemble des variables de types *libres* (c'est-à-dire apparaissant) dans τ, ensemble défini par :

$$\begin{array}{lll} \mathcal{L}(\iota) & = & \emptyset \\ \mathcal{L}(\alpha) & = & \{\alpha\} \\ \mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \end{array}$$

Nous appellerons *substitution* une application partielle à domaine fini des variables de type dans les types :

$$\varphi \ ::= [\alpha_1 \leftarrow \tau_1; \ldots; \alpha_n \leftarrow \tau_n]$$

On prolonge naturellement une substitution en une application partielle à domaine fini des types dans les types, que l'on note $\overline{\varphi}$, de la façon suivante :

$$\begin{array}{rcl} \overline{\varphi}(\iota) & = & \iota \\ \overline{\varphi}(\alpha) & = & \varphi(\alpha) \text{ si } \alpha \in \mathrm{Dom}\,(\varphi) \\ \overline{\varphi}(\alpha) & = & \alpha \text{ si } \alpha \notin \mathrm{Dom}\,(\varphi) \\ \overline{\varphi}(\tau_1 \to \tau_2) & = & \overline{\varphi}(\tau_1) \to \overline{\varphi}(\tau_2) \end{array}$$

Désormais, nous confondrons la substitution φ et son extension $\overline{\varphi}$, et noterons φ pour les deux.

2.3.2 Schémas de types

Nous définissons maintenant la notion de *schéma de types*. Comme mentionné précédemment, le système de types de ML autorise les expressions à appartenir à plusieurs types : c'est le *polymorphisme*. Chacun de ces types sera alors dérivé d'un « moule » que l'on appelle « schéma ». Plus précisément les schémas de types sont dérivés de la grammaire suivante :

$$\sigma \ ::= \forall \alpha_1 \ldots \alpha_n . \tau$$

dans laquelle l'ordre des variables α_i universellement quantifiées ne présente pas d'importance : il s'agit d'un ensemble de variables que l'on suppose toutes différentes. De plus, on identifie deux schémas de types identiques à alpha-conversion près.

On définit ensuite simplement la notion de *variables libres d'un schéma de types* par :

$$\mathcal{L}(\forall \alpha_1 \ldots \alpha_n . \tau) = \mathcal{L}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}$$

On appelera *schéma trivial* un schéma de la forme $\forall \emptyset . \tau$. Ceci correspond à identifier (transformer) un type τ en schéma de types sans effectuer aucune quantification sur aucune des variables présentes dans le type τ.

2.3.3 Environnements de typage

Certaines expressions peuvent définir des identificateurs. Au cours du processus de typage, un schéma de types sera déterminé pour chacun de ces identificateurs et devra être mémorisé afin de pouvoir être utilisé à chaque fois que cet identificateur apparaîtra dans le reste de l'expression (ou du programme plus généralement).

L'environnement de typage, qui associe un schéma de types à un identificateur, remplit donc cette fonction. Nous pouvons donc le voir soit comme une simple liste d'association (en terme d'implémentation), soit comme une application des identificateurs du programme dans les schémas de types. Nous pouvons d'ores et déjà remarquer qu'associer les identificateurs à des schémas de types plutôt qu'à des types permettra d'utiliser les identificateurs de manière polymorphe.

Un environnement de typage, noté E est donc défini par :

$$E \ ::= [\mathtt{x}_1 \mapsto \sigma_1 ; \ldots ; \mathtt{x}_n \mapsto \sigma_n]$$

On étend la notion de variables libres sur un environnement de typage par :

$$\mathcal{L}(E) = \bigcup_{x \in \mathrm{Dom}\,(E)} \mathcal{L}(E(x))$$

La notion de schéma de types trivial que nous avons vue précédemment permet ainsi d'insérer un type dans un environnement de typage en respectant la structure de ces derniers qui ne doivent contenir que des schémas. Intuitivement, il est possible de considérer ceci comme une encapsulation d'un type dans un schéma, ou encore une injection des types dans les schémas.

2.3.4 Règles de typage

Nous présentons le système de types de ML par l'intermédiaire de *règles d'inférence* de la forme :

$$\frac{P_1 \ P_2 \ldots \ P_n}{P}$$

se lisant de la façon suivante : la proposition P est vraie si chacune des prémisses $P_1, P_2, \ldots, P_n$ est vérifiée. On peut également comprendre cette notation en considérant que $P_1, P_2, \ldots, P_n$ sont les hypothèses requises afin que la propriété P soit vérifiée.

Ces règles permettent de définir des *jugements de typage* de la forme $E \vdash a : \tau$ signifiant que sous les hypothèses de type contenues dans l'environnement de typage E, on peut déduire que l'expression a possède le type τ.

Définition 2.1 (Généralisation) *On définit la généralisation d'un type τ dans un environnement E comme étant le schéma de types tel que :*

$$\mathrm{Gen}\,(\tau, E) = \forall \alpha_1 \ldots \alpha_n . \tau \quad \textit{avec} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{L}(\tau) \setminus \mathcal{L}(E)$$

Définition 2.2 (Instance) *On dira que τ' est une instance du schéma de types $\sigma = \forall \alpha_1 \ldots \alpha_k . \tau$ et l'on notera $\sigma \leq \tau'$, s'il existe une substitution φ telle que* $\mathrm{Dom}\,(\varphi) = \{\alpha_1 \ldots \alpha_k\}$ *et que $\tau' = \varphi(\tau)$.*

On présumera donnée une fonction TypeArg, qui étant donné un constructeur d'exceptions, nous fournit le schéma de types de son paramètre. Il est également possible de considérer que les constructeurs sont plongés dans l'environnement de typage E, mais ceci complique inutilement la présentation du système. De plus, l'environnement initial de typage est supposé contenir une fonction primitive `raise` liée au schéma de types $\forall \alpha.\, \mathtt{exn} \to \alpha$.

Typage des motifs

$$\vdash x : \tau \Rightarrow \{x : \tau\} \qquad \vdash i : \mathtt{int} \Rightarrow \{\} \qquad \vdash \mathtt{C} : \mathtt{exn} \Rightarrow \{\}$$

$$\frac{\tau \leq \mathrm{TypeArg}(\mathtt{D}) \qquad \vdash p : \tau \Rightarrow E}{\vdash \mathtt{D}(p) : \mathtt{exn} \Rightarrow E}$$

Typage des expressions

$$E \vdash i : \mathtt{int} \qquad E \vdash \mathtt{C} : \mathtt{exn}$$

$$\frac{\tau \leq \mathrm{TypeArg}(\mathtt{D}) \qquad E \vdash a : \tau}{E \vdash \mathtt{D}(a) : \mathtt{exn}} \qquad \frac{E(x) \leq \tau}{E \vdash x : \tau'}$$

$$\frac{E \vdash a : \tau \qquad \vdash p : \tau \Rightarrow E' \qquad E \oplus E' \vdash b : \tau' \qquad E \oplus \{x : \tau\} \vdash c : \tau'}{E \vdash \mathtt{match}\ a\ \mathtt{with}\ p \to b \mid x \to c : \tau'}$$

$$\frac{E \vdash a : \tau \qquad E \oplus \{x : \mathtt{exn}\} \vdash b : \tau}{E \vdash \mathtt{try}\ a\ \mathtt{with}\ x \to b : \tau}$$

$$\frac{E \vdash a : \tau_1 \to \tau \qquad E \vdash b : \tau_1}{E \vdash (a\ b) : \tau} \qquad \frac{E \oplus (x : \tau_1) \vdash a : \tau_2}{E \vdash \lambda x.a : \tau_1 \to \tau_2}$$

$$\frac{E \vdash a_1 : \tau_1 \qquad \sigma = \mathrm{Gen}\,(\tau_1, E) \qquad E \oplus (x : \sigma) \vdash a_2 : \tau_2}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2}$$

2.3.5 Propriétés

Il est possible d'établir un certain nombre de propriétés de ce système de types énonçant sa correction vis-à-vis de la sémantique du langage. Nous énonçons ici les deux plus importantes sans toutefois les démontrer. Le lecteur désireux d'en savoir plus pourra trouver les preuves de ces propriétés rédigées en détail dans [28, 4, 35, 58, 65].

Proposition 2.1 (Stabilité du typage par substitution) *Soient a une expression, τ un type, E un environnement de typage, et φ une substitution. Si $E \vdash a : \tau$, alors $\varphi(E) \vdash a : \varphi(\tau)$.*

Proposition 2.2 (Sûreté forte) *Soient* a *une expression,* τ *un type et* E *un environnement de typage tels que* $E \vdash a : \tau$*. Soit :*

- $a \xrightarrow{*} v$ *où* v *est une valeur de type* τ
- $a \xrightarrow{*}$ `raise` v *où* v *est une valeur de type* `exn`
- $a \xrightarrow{*} a'$ *et* $a' : \tau$ *et* a' *peut continuer à se réduire*

Ceci signifie intuitivement que l'on n'est jamais bloqué sur un terme de la forme (1 2).

2.4 Inférence de types

Nous venons de voir au travers des règles de typage qu'il est possible d'attribuer un type à toute expression dont l'évaluation ne résultera pas en une erreur. Nous avons également vu que le polymorphisme autorisait les expressions du langage à posséder plusieurs types (c'est-à-dire que si une expression close a possède le type τ, elle est aussi du type τ' pour tout τ' moins général que τ : autrement dit $\tau' = \varphi(\tau)$ pour une substitution φ). Le calcul du type principal (c'est-à-dire le plus général) d'une expression est appelé processus d'*inférence de types*. Il repose essentiellement sur l'unification entre termes du premier ordre et donne lieu à un algorithme que l'on peut prouver correct et complet vis-à-vis des règles de typage.

Nous ne décrirons pas ici cet algorithme, ni ne ferons les preuves de sa correction et de sa complétude, celles-ci pouvant être également trouvées *in extenso* dans [28, 4]. Notons que des preuves similaires seront effectuées lorsque nous établirons les propriétés de notre analyse d'exceptions, basée elle aussi sur un système de types et munie d'un algorithme d'inférence.

Afin d'introduire les idées que nous développerons ultérieurement, disons seulement que la propriété de correction d'un tel algorithme permet d'établir qu'étant donné un environnement de typage E et une expression a, l'algorithme d'inférence est capable de synthétiser automatiquement un type τ tel que a ait effectivement le type τ en respect des règles de typage du système.

La complétude de cet algorithme établit quant à elle, le fait que le type τ qu'il synthétise pour étiqueter l'expression a est le plus général, c'est-à-dire qu'il pose le minimum de contraintes et ne réduit ainsi pas le spectre d'utilisation de l'expression. Autrement dit, il est impossible de trouver un « meilleur » type pour l'expression.

Chapitre 3

Premières techniques envisagées

Avant de décider d'employer une méthode d'analyse à base de contraintes d'unification, nous avons envisagé et essayé de mettre au point plusieurs analyseurs basés sur d'autres techniques déjà existantes en étendant, si nécessaire, leur principe aux constructions fournies par le langage à analyser (en l'occurrence Objective Caml), par exemple aux objets et aux valeurs mutables. De ces essais sont nés plusieurs prototypes que nous avons pu comparer et examiner en terme de précision et de temps d'analyse. Nous allons ici les étudier plus en détail, sans toutefois faire à chaque fois un exposé formel de la théorie sous-jacente, ce qui sortirait du cadre de cette thèse. Pour une description plus détaillée, il sera toujours possible au lecteur intéressé de se référer aux documents fondateurs que nous citerons au fur et à mesure de ce chapitre.

3.1 $0-$CFA / $n-$CFA

3.1.1 Principe

L'idée de base de la 0-CFA (0^{th} *Order Control Flow Analysis*) consiste à approximer l'ensemble des valeurs prises par chaque identificateur, et plus généralement par chaque point du programme, par un ensemble *explicite* de valeurs d'approximation. Contrairement à la SBA (*Set Based Analysis*) que nous décrirons en section 3.2, en 0-CFA nous désirons obtenir un résultat de la forme : « l'identificateur `x` peut prendre une valeur parmi l'ensemble $\{0; 1; 5\}$ ».

Pour ce faire, nous allons fusionner toutes les valeurs prises par un identificateur au sein d'un seul et même ensemble. Aussi, toute relation entre l'entrée d'une fonction et sa sortie est irrémédiablement perdue. Par exemple, dans ce programme :

```
let f x = x + 1 in f 0 ; f 10
```

la fonction `f` possédera comme ensemble de valeurs d'entrée $\{0; 10\}$ et comme ensemble de valeurs de sortie $\{1; 11\}$. Il ne sera alors plus possible de savoir que l'entrée 0 est liée à la sortie 1 et que l'entrée 10 produit 11 comme valeur de sortie.

La 0-CFA est une analyse itérative qui applique répétitivement un opérateur sur le programme jusqu'à obtention d'un point fixe (c'est-à-dire jusqu'à ce que l'ensemble de solutions obtenu à l'étape n soit identique à celui obtenu à l'étape $n-1$). Les ensembles de solutions sont composés de *valeurs d'approximation* que nous choisissons comme énoncé ci-dessous.

Dans un premier temps, nous nous restreignons à une structure d'approximation simple pour rendre la présentation plus aisée. En particulier nous ne considérons pas encore les approximations de structures complexes comme les n-uplets que nous examinerons ultérieurement. De même, nous ne considérons pas la prise en compte des exceptions et nous verrons qu'il est trivial d'étendre cette analyse pour lui faire intégrer cette notion.

Valeurs d'approximation :

a ::=	$\bot$	« Bottom » : ensemble vide de valeurs
	$\mid \top$	« Top » : ensemble de toutes les valeurs possibles
	$\mid \{\mathtt{f}_1; \ldots; \mathtt{f}_n\}$	Étiquettes de fonction
	$\mid \{0; 1; \ldots\}$	Valeurs entières

$\bot$ représente un ensemble « vide » de valeurs : il dénote le fait que ce point du programme n'intervient dans aucun calcul, donc n'accumule aucune information. L'apparition de tels ensembles permet par ailleurs de détecter les sections de « code mort » dans un programme.

$\top$ représente la perte maximale de précision : un tel ensemble apparaît lorsque l'analyse ne peut plus déterminer de manière précise l'ensemble des valeurs d'un point du programme et décide, de façon conservative, de considérer que n'importe quelle valeur peut potentiellement arriver jusqu'à ce point de programme. Des exemples typiques d'utilisations de $\top$ sont fournis par les fonctions d'entrée-sortie (lecture au clavier), les opérations arithmétiques (pour lesquelles l'analyse ne cherche pas à simuler le comportement) ou encore les fonctions de génération de nombres aléatoires. Ainsi `Random.int` retournerait une valeur entière approximée par $\top$.

Une valeur d'approximation peut également être spécifiée par la donnée en extension d'un ensemble de valeurs entières, ou bien d'étiquettes de fonctions.

Afin de pouvoir énumérer de telles étiquettes, il est indispensable, préalablement à l'analyse elle-même, que toutes les fonctions du programme soient nommées de manière explicite et unique. De plus, chaque identificateur doit également être nommé de façon unique et les arguments des fonctions doivent être explicitement mentionnés. Il s'agit donc de transformations intervenant au niveau de l'arbre de syntaxe abstraite du programme à analyser. Ainsi, la fonction anonyme suivante : `fun x → fun y → fun z → ...` devra être transformée en : $\texttt{<f}_1\texttt{> fun x} \to \texttt{<f}_2\texttt{> fun y} \to \texttt{<f}_3\texttt{> fun z} \to \texttt{...}$ où $\mathtt{f}_1$, $\mathtt{f}_2$ et $\mathtt{f}_3$ représentent le nom des fonctions intermédiaires. Caml fournissant la définition de fonctions par filtrage, il est également nécessaire d'appliquer le même style de transformation afin de garantir que les arguments de fonctions soient explicites. Par exemple :

```
let f = function
  | true → ...
  | false → ...
```

sera transformé en :

```
let <f> = fun arg →
  match arg with
    | true → ...
    | false → ...
```

Le nommage unique de chaque identificateur peut être aisément réalisé par l'adjonction d'une étiquette (en Anglais, *stamp*) représentée, par exemple, par un entier et associée à chaque identificateur : ainsi un identificateur n'est plus seulement une chaîne de caractères, mais un couple (chaîne, étiquette). Nous appellerons ce mécanisme la « détermination de portée des identificateurs » (en Anglais *scoping*).

Quant à l'ajout d'identificateurs explicites, il peut facilement être obtenu sans conflit avec les identificateurs que peut définir le programmeur, par utilisation de caractères interdits par la syntaxe du langage au niveau des identificateurs. À titre anecdotique, un choix sûr est d'utiliser, pour ces noms générés en interne, des séquences de caractères représentant des commentaires dans la syntaxe du langage (par exemple `(*internal_ident_1*)` en Caml). Ainsi, il est impossible d'entrer en conflit avec les identificateurs du programmeur puisqu'étant considérées par le compilateur comme des commentaires, ces portions du programmes sont purement et simplement ignorées.

Nous définissons maintenant l'union de deux ensembles d'approximations de la façon suivante, avec $\perp$ comme élément neutre et $\top$ comme élément absorbant :

$$\begin{array}{ccc}
\{\mathtt{f}_1;\ \ldots;\ \mathtt{f}_k\} \cup \{\mathtt{f'}_1;\ \ldots;\ \mathtt{f'}_k\} & = & \{\mathtt{f}_1;\ \ldots;\ \mathtt{f}_k;\ \mathtt{f'}_1;\ \ldots;\ \mathtt{f'}_k\} \\
\{i;\ \ldots;\ j\} \cup \{i';\ \ldots;\ j'\} & = & \{i;\ \ldots;\ j;\ i';\ \ldots;\ j'\} \\
a \cup \top & = & \top \\
\top \cup a & = & \top \\
a \cup \perp & = & a \\
\perp \cup a & = & a
\end{array}$$

Il nous reste à envisager le cas où nous cherchons à effectuer l'union d'approximations représentant des ensembles hétérogènes : par exemple des entiers et des étiquettes de fonctions. De prime abord, une telle union peut sembler non définie : ML étant typé, il est impossible de mélanger des valeurs de type différent. Néanmoins, ce type d'analyse pouvant s'appliquer à d'autres langages que ceux fortement typés, la question mérite d'être posée.

Qui plus est, rien ne garantit que l'analyse obtenue soit aussi précise que le typage de ML. En particulier, si nous considérons le cas des fonctions polymorphes, leurs arguments et résultats peuvent naturellement être de types différents. L'accumulation des informations provenant des différents sites d'appel de telles fonctions peut mener à de tels ensembles hétérogènes comme dans le programme suivant : `let f x = x in f 1 ; f true`. Aussi la question prend-t'elle tout son sens et nous définissons l'union de tels ensembles par :

$$\{i;\ \ldots;\ j\} \cup \{\mathtt{f'}_1;\ \ldots;\ \mathtt{f'}_k\} \quad = \quad \{i;\ \ldots;\ j;\ \mathtt{f'}_1;\ \ldots;\ \mathtt{f'}_k\}$$

Le processus itératif d'analyse va donc construire deux fonctions d'approximation ν et ρ. La première fait correspondre à chaque *variable* du programme l'approximation des valeurs pouvant être contenues par cette variable. La seconde associe à chaque *nom de fonction* l'approximation du résultat de cette fonction. On calcule l'approximation d'une expression de la façon suivante :

$$\begin{array}{lcl}
\mathcal{A}(\mathtt{x}) & = & \nu(\mathtt{x}) \\
\mathcal{A}(i) & = & \{i\} \\
\mathcal{A}(<\mathtt{f}>\mathtt{fun\ x} \to e) & = & \{\mathtt{f}\} \\
\mathcal{A}(e_1; e_2) & = & \mathtt{calculer}\ \mathcal{A}(e_1)\ \mathtt{et\ renvoyer}\ \mathcal{A}(e_2) \\
\mathcal{A}(\mathtt{let\ x}\ = e_1\ \mathtt{in}\ e_2) & = & \mathtt{faire}\ \nu(\mathtt{x}) \leftarrow \nu(\mathtt{x}) \cup \mathcal{A}(e_1)\ \mathtt{et\ renvoyer}\ \mathcal{A}(e_2) \\
\mathcal{A}(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) & = & \mathtt{calculer}\ \mathcal{A}(e_1)\ \mathtt{et\ renvoyer}\ \mathcal{A}(e_2) \cup \mathcal{A}(e_3) \\
\mathcal{A}(e_1\ e_2) & = & \mathtt{si}\ \mathcal{A}(e_1) = \{\mathtt{f}_1;\ \ldots;\ \mathtt{f}_k\} \\
 & & \mathtt{alors\ renvoyer}\ \bigcup \{\mathcal{B}(\mathtt{f}_i, \mathcal{A}(e_2)) \mid i = 1 \ldots k\} \\
 & & \mathtt{sinon\ calculer}\ \mathcal{A}(e_2)\ \mathtt{et\ renvoyer}\ \top
\end{array}$$

$$\begin{array}{ll}
\mathcal{B}(\mathtt{f}, a) & = \\
 & \quad \mathtt{soit\ x} = \mathcal{P}aram(\mathtt{f}) \\
 & \quad \mathtt{faire}\ \nu(x) \leftarrow \nu(x) \cup a \\
 & \quad \mathtt{renvoyer}\ \rho(\mathtt{f})
\end{array}$$

Nous remarquons l'aspect impératif de la présentation que nous avons donnée de cet algorithme. En particulier, l'approximation d'une expression requiert le calcul de l'approximation de chacune de ses sous-expressions, indépendamment de leur utilisation directe dans l'approximation du résultat. Ceci est précisément dû aux mises à jour effectuées par effet de bord dans les fonctions ν et ρ de chaque variable et chaque fonction. L'algorithme d'approximation d'un programme s'écrit maintenant sous la forme :

```
∀ x, ν(x) ← ⊥
∀ f, ρ(f) ← ⊥
répéter
  𝒜(programme)
  ∀ f, ρ(f) ← ρ(f) ∪ 𝒜( Corps (f))
tant que ν ou ρ a changé
```

Notons que la présentation impérative que nous avons faite de la 0-CFA n'est pas la seule possible, il s'agit seulement de la façon courante de procéder, qui permet une vue intuitive relativement aisée du principe de l'analyse. On pourra trouver des présentations sous forme plus déclaratives dans [23, 37].

Nous voyons maintenant tout l'intérêt que prend le nommage unique des identificateurs de variables ainsi que l'explicitation des arguments de fonction. En effet, le but de ces transformations est de permettre d'identifier les sites où mémoriser les ensembles de valeurs.

Intuitivement, la garantie de terminaison de l'algorithme repose principalement sur deux points. D'une part, les valeurs constantes explicites du programme sont en nombre fini. D'autre part, les solutions de l'analyse d'une expression sont issues d'ensembles composés par union de ces valeurs ainsi que par $\perp$ et $\top$. Aussi les ensembles possibles sont également en nombre fini (notons que ceci nécessite le respect de propriétés de monotonie sur les opérateurs d'approximation des constructions externes au langage comme les primitives). On pourra trouver une présentation plus formelle de ces propriétés dans [54, 55, 38, 52], l'étude de ce point dépassant le cadre de cette thèse.

Notons que l'exemple que nous avons donné au début de cette section peut suggérer que les constantes ne sont pas, contrairement à ce que nous venons de dire, en nombre fini. En effet, dans cet exemple, nous disions que si l'ensemble des valeurs d'entrée de la fonction était $\{0; 10\}$, alors son ensemble de valeurs de sortie serait $\{1; 11\}$ (apparition de deux nouvelles constantes 1 et 11). En réalité, ceci ne se produit pas car lors de l'analyse, nous « n'évaluons pas » les expressions arithmétiques celles-ci étant considérées comme des opérateurs externes au langage. Typiquement, le résultat d'une telle opération sera $\top$ (la formulation de cet exemple n'avait pour but qu'exposer simplement et informellement la fusion des valeurs prises par les identificateurs).

3.1.2 Extensions

Afin d'analyser un langage complet, nous devons envisager la prise en compte de structures de données plus complexes : les n-uplets et les types somme. On se convaincra aisément que des structures comme les enregistrements pourront être considérées comme un cas particulier de n-uplets. Nous étudierons également la construction `match ... with ...` permettant le filtrage (*pattern matching*), puisque celle-ci est similaire au mécanisme mis en œuvre dans les gestionnaires d'exceptions via `try ... with ...`

Les n-uplets Une première approche consiste à étendre les valeurs d'approximation par des ensembles de n-uplets donnés en extension.

Valeurs d'approximation :

$$\begin{aligned} a \ ::=\ & \ldots \\ & \mid \{(a_1, \ldots, a_k); \ldots; (a'_1, \ldots, a'_k)\} \end{aligned}$$

et l'on calcule alors l'approximation d'une expression de n-uplet par : $\mathcal{A}(e_1, \ldots, e_k) = (\mathcal{A}(e_1), \ldots, \mathcal{A}(e_k))$. L'union entre ensembles d'approximations se complète trivialement de la même façon qu'elle opère sur les ensembles d'approximations d'entiers.

Cette stratégie confère une grande précision à l'analyse puisqu'elle est capable de garder exactement trace de tous les n-uplets apparaissant dans le programme, néanmoins elle présente le grave inconvénient de nous exposer à une combinatoire excessive. En effet, considérons le programme :

```
let x = if ... then 1 else 2 in
let y = if ... then 3 else 4 in
(x, y)
```

Son approximation sera composée d'un ensemble de 4 n-uplets comportant chacun 2 éléments. Trivialement, nous pouvons généraliser cet exemple et se rendre compte que sur un exemple similaire comportant 5 n-uplets de 5 éléments chacun, il aura fallu à un *Pentium Pro à 150 MHz* 13 minutes pour calculer l'approximation du programme et 2,562 Mo de texte pour afficher ce résultat ! Il apparaît clairement qu'une telle stratégie est hors de portée dans de telles conditions.

Nous choisissons donc une seconde approche consistant à étendre les approximations en considérant que l'approximation d'un n-uplet est le n-uplet composé des approximations de ses composantes. On obtient donc une structure de valeurs d'approximation « plate » de la forme suivante :

Valeurs d'approximation :

$$\begin{array}{rl} a & ::= \ldots \\ & \mid (a_1, \ldots, a_k) \end{array}$$

On calcule alors l'approximation d'une expression de n-uplet de la même façon que précédemment, mais définissons l'union sur les approximations de la façon suivante :

$$(a_1, \ldots, a_k) \cup (a'_1, \ldots, a'_k) = (a_1 \cup a'_1, \ldots, a_k \cup a'_k)$$

Il s'ensuit une perte de précision au prix d'un gain important en terme de vitesse d'analyse et d'espace mémoire requis pour la mener à bien. Notons toutefois que l'analyse du filtrage sur les n-uplets s'en trouve dégradée comme le verrons un peu plus loin.

Le filtrage Jusqu'à présent, nous n'avons utilisé que des constructions du langages impliquant des unions sur les ensembles d'approximations. Si nous prenons en compte le filtrage, nous devons ajouter à notre CFA la possibilité de supprimer des valeurs dans une approximation afin de rendre compte du mécanisme d'exclusion des différentes branches d'un filtrage. Dans le cas suivant :

```
match { 13 ; 42 } with
  | 0 → 1
  | 13 → 2
  | n → n
  | m  → 0
```

nous aimerions être capables de nous rendre compte que la première branche est inutile (puisque l'ensemble des valeurs confronté à ce filtrage ne contient pas 0), que les seconde et troisième branches sont utilisées, mais que la dernière ne l'est pas (les branches précédentes ayant déjà absorbé toutes les valeurs présentes dans l'ensemble). À cette fin, nous devons définir un mécanisme de *soustraction* d'une valeur d'approximation dans un ensemble que nous pouvons énoncer de la façon suivante :

$$\begin{array}{ccc} \{\mathtt{f}_1; \ldots \mathtt{f}_n\} \setminus \mathtt{f} & = & \{\mathtt{f}_1; \ldots; \mathtt{f}_{i-1}; \mathtt{f}_{i+1} \ldots; \mathtt{f}_n\} \text{ si } \mathtt{f} = \mathtt{f}_i \\ & = & \{\mathtt{f}_1; \ldots \mathtt{f}_n\} \text{ si } \mathtt{f} \neq \mathtt{f}_i \text{ pour tout } i \\ \{\} \setminus a & = & \{\} \\ \top \setminus a & = & \top \\ \bot \setminus a & = & \bot \end{array}$$

en nous appuyant bien évidemment sur une forme normale des ensembles interdisant les doublons d'éléments. Ainsi dans l'exemple précédent, l'approximation des valeurs pouvant parvenir jusqu'à `n` ne contiendra ni 0 ni 13, ce qui le réduit bien à $\{42\}$.

Comme mentionné précédemment, la stratégie d'approximation choisie pour les n-uplets (et les enregistrements) se prête mal à une analyse fine des filtrages. Pour preuve l'exemple suivant :

```
let v = if ... then (0, 1) else (2, 3) in
match v with
  | (0, 3) → ...
```

dans lequel l'approximation de `v` est $(\{0; 2\}, \{1; 3\})$, forçant l'oubli du lien qu'il existe entre chaque composante deux à deux des couples d'origine. Ainsi, l'analyse déduira à tort (pour rester correcte) que la branche du filtrage ci-dessus est utilisée.

Néanmoins, afin de rester correct vis-à-vis de la sémantique dynamique du langage, il ne faudra pas considérer que les valeurs d'approximation considérées comme filtrées par le motif sont définitivement absorbées ! En effet, considérons l'exemple suivant :

```
let v = if ... then (0, 2) else (0, 3) in
match v with
  | (0, 2) → ...
  | (0, 3) → ...
```

dans lequel l'approximation liée à `v` est $(\{0\}, \{2; 3\})$. Considérer que 0 doit être soustrait de cet ensemble après être passé par le premier filtre, résulterait en un ensemble vide pour la première composante de l'approximation restante, autorisant ainsi à tort la déduction que le second motif du filtrage est inutilisé et donc que sa partie droite ne participe pas au résultat de l'expression. Notons que nous perdons ainsi par la même occasion la possibilité de détecter l'exhaustivité des motifs dans les filtrages.

On extrapole aisément ce comportement aux gestionnaires d'exceptions (qui sont une construction très similaire au `match ... with ...`), mettant en évidence que ce mécanisme pourra causer des notifications abusives d'exceptions non rattrapées.

Les types somme (valeurs construites) Caml permettant de déclarer des types somme munis de constructeurs constants ainsi que paramétrés, il semble naturel d'étendre nos approximations de la façon suivante :

Valeurs d'approximation :

$$
\begin{array}{ll}
a & ::= \dots \\
 & \mid \{C_1; \dots; C_n; D_1(a_1); \dots D_m(a_m)\}
\end{array}
$$

où les $\mathtt{C}_i$ sont des constructeurs constants et les $\mathtt{D}_i$ des constructeurs paramétrés portant les approximations a_i de leur argument.

Ainsi, pour la définition de type `type t = A | B of int`, l'expression `if ... then A else B(5)` serait approximée par l'ensemble $\{\mathtt{A}; \mathtt{B}(\{5\})\}$. Une telle structure nous permet ainsi d'approcher les constructeurs ainsi que leurs arguments, conférant ainsi à l'analyse une précision satisfaisante. On étend alors l'union d'approximations trivialement par une union factorisant les constructeurs afin d'éviter tout comportement combinatoire (comme dans le cas des n-uplets), à savoir :

$$
\begin{array}{rcl}
\{\mathtt{C}\} \cup \{\mathtt{C}_1; \dots; \mathtt{C}_n; \mathtt{D}_m(a_m)\} & = & \{\mathtt{C}; \mathtt{C}_1; \dots \mathtt{C}_n; \mathtt{D}_m(a_m)\} \\
 & & \quad \text{si } \ \mathtt{C} \neq \mathtt{C}_i \quad \text{pour tout } i \in [1 \dots n] \\
 & = & \{\mathtt{C}_1; \dots \mathtt{C}_n; \mathtt{D}_m(a_m)\} \quad \text{sinon} \\
\{\mathtt{D}(a)\} \cup \{\mathtt{D}(a'); \dots\} & = & \{\mathtt{D}(a \cup a'); \dots\} \\
\{\mathtt{D}(a)\} \cup \{\mathtt{C}_i; \mathtt{D}_1(a_1); \dots \mathtt{D}_n(a_n)\} & = & \{\mathtt{C}_i; \mathtt{D}(a); \mathtt{D}_1(a_1); \dots; \mathtt{D}_n(a_n)\} \\
 & & \quad \text{si } \ \mathtt{D} \neq \mathtt{D}_i \text{ pour tout } i \in [1 \dots n]
\end{array}
$$

Considérons maintenant les entiers de Peano, définis par le type ML suivant : `type nat = O | S of nat` et la fonction identité sur ces entiers :

```
let rec id = function
  | O → O
  | S n → S (id n) in
id (S O)
```

Malheureusement, appliquer l'analyse avec la structure d'approximations que nous avons ainsi définie remet en question la terminaison de l'algorithme. Tentons tout d'abord de comprendre intuitivement le comportement de cet exemple. Au début de l'analyse, nous avons $\rho(\texttt{id}) = \perp$. La première itération mémorise que $\nu(\texttt{n}) = \{\texttt{O}\}$ puis déduit en résultat que `id` renvoie $\texttt{S}(\perp)$, c'est-à-dire $\perp$. En même temps, on s'est rendu compte que `id` pouvait être appelée avec `O` comme argument (par l'intermédiaire de son appel récursif) puisque l'analyse avait mémorisé que $\nu(\texttt{n}) = \{\texttt{O}\}$. Ainsi à l'itération suivante, nous déduirons que `id` peut retourner $\{\texttt{O}\}$ puisque son argument contient `O` et permet d'utiliser la première branche du filtrage. Ainsi maintenant, $\rho(\texttt{id}) = \{\texttt{O}\}$. Les ensembles ν et ρ n'ayant pas atteint de point fixe, une nouvelle itération commence alors au cours de laquelle nous verrons que `id` peut (par l'intermédiaire de son appel récursif) retourner également « `S` de ce qu'elle pouvait retourner précédemment », c'est-à-dire `S(O)`. Ainsi on a $\rho(\texttt{id}) = \{O; S(O)\}$. Nous devons alors continuer à itérer et l'on s'aperçoit alors aisément que l'itération suivante nous mènera à $\rho(\texttt{id}) = \{O; S(O); S(S(O))\}$ et ainsi de suite…

La raison de cette non-terminaison réside dans le fait que les valeurs d'approximation n'existent plus en nombre fini avec la structure que nous avons choisie, en raison de la récursivité apparaissant dans cette structure. Aussi est-il possible que les ensembles solutions de notre analyse s'étendent toujours plus à chaque itération. La seule possibilité de remédier à ce comportement consiste à briser la récursivité qu'il existe au niveau de nos approximations dans le cas des constructeurs paramétrés par des valeurs. Une première alternative consiste à ignorer péremptoirement les arguments des constructeurs en les approchant par $\top$, au prix d'une perte importante de précision de l'analyse. Une seconde solution pour réduire la dégradation de la précision de l'analyse consiste à classer les constructeurs paramétrés en deux catégories. Tout d'abord, celle de ceux donc l'argument est une valeur « simple » (comme les entiers, les booléens, les chaînes de caractères, etc…), autrement dit une approximation n'introduisant pas de récursion. Et la seconde regroupant les constructeurs dont l'argument, plus complexe, introduit un comportement récursif, que l'on approche par $\top$.

Ainsi, nos approximations doivent être stratifiées plus finement que la présentation que nous en avons faite tout au début de cette section. Nous devons donc définir une notion d'approximation « pouvant paramétrer un constructeur » et une notion d'approximation « ne pouvant pas le faire ».

Nous pouvons à juste titre nous demander si l'ajout de n-uplets, que nous avons étudié précédemment, ne pose pas le même problème (en effet, les n-uplets sont un cas particulier de constructeur paramétré). Pratiquement, en s'appuyant sur l'hypothèse que les programmes que nous désirons analyser sont typés par ML, nous avons l'invariant que les n-uplets ne peuvent pas être récursifs. Ainsi, il sera impossible d'écrire en ML le programme `let rec f x = f (x, x) in f ...`, le typage l'aura refusé. Bien entendu, si nous relaxons cette hypothèse, alors l'analyse des n-uplets pourra aboutir à la non terminaison de l'algorithme de 0-CFA.

3.1.3 Résultats pratiques

Nous avons réalisé un prototype d'analyseur basé sur la 0-CFA pour un sous-ensemble de Caml, c'est-à-dire un langage comportant les types de base (`int, bool, string, float`), les types sommes, les n-uplets et les enregistrements, ainsi que toutes les constructions de base du langage (application, fonction, séquence, conditionnelle, filtrage, liaisons `let` et `let rec`, boucles (`for` et `while`) et la levée d'exceptions (`raise`)).

Les performances de temps obtenues restent loin derrière celles que nous cherchons à obtenir comme énoncé précédemment. Bien que notre implémentation de cette analyse ne saurait en aucun cas être considérée comme optimale, force est de constater que la complexité de la 0-CFA en elle-même est $O(n^3)$ où n représente la taille du programme, ne permettant ainsi pas d'espérer des gains de vitesse d'un facteur significatif. Néanmoins, nous n'avons pas observé un tel comportement cubique au cours de nos tests.

Toutefois, un comportement quadratique ($O(n^2)$) est observable dans des cas relativement courants. Considérons qu'un groupe de fonctions de taille $k = O(n)$ récurse sur une liste de $m = O(n)$ éléments donnés en extension dans le source du programme. L'analyse requiert alors au minimum m itérations avant l'atteinte du point fixe sur l'ensemble des paramètres et des résultats des fonctions. Puisque chaque itération est proportionnelle à k, le temps de l'analyse est $O(n^2)$.

3.2 Set Based Analysis

Contrairement à la 0-CFA, en SBA (*Set Based Analysis*) l'approximation des ensembles de valeurs prises par chaque identificateur (et par extension, chaque point du programme), est réalisée par la construction d'un ensemble *implicite*, par opposition aux ensembles donnés en extension de la 0-CFA. Cet ensemble peut alors être vu comme une *grammaire* capable de décrire des ensembles potentiellement infinis. Cette grammaire est décrite par un ensemble de contraintes d'inclusion et d'égalité qu'il conviendra alors de simplifier afin d'en extraire des ensembles *explicites* de valeurs.

Une présentation détaillée de la *Set Based Analysis* ne saurait trouver sa place dans cette thèse. Le lecteur intéressé pourra trouver de plus amples informations dans les documents fondateurs [15, 17, 14, 16, 18].

En terme de résultats pratiques, nous avons implémenté une telle analyse sur un sous-ensemble très restreint de ML. Plusieurs raisons ont motivé notre abandon dans une plus ample exploration de cette voie. À l'instar de celle de la 0-CFA, la complexité algorithmique de cette analyse reste en $O(n^3)$, pour n représentant la taille du programme. De plus, l'obtention d'une grammaire décrivant les ensembles (potentiellement infinis) de valeurs en chaque point du programme n'est pas suffisante pour notre analyse d'exceptions : il reste nécessaire d'expliciter ces ensembles pour les exploiter et présenter des informations à l'utilisateur. Plusieurs algorithmes travaillant sur de telles grammaires ont été étudiés [2, 25, 69], mais leur complexité reste relativement élevée, nous éloignant encore d'avantage des performances que nous attendons pour notre analyseur. Finalement, la prise en compte du filtrage dans le cadre de la SBA reste encore un problème ouvert, nécessitant entre autre l'ajout de contraintes négatives. Or, par nature, une analyse d'exceptions devra être capable de gérer précisément cette notion, le concept de gestionnaire d'exceptions (c'est-à-dire la construction `try ... with ...`) s'appuyant fortement dessus.

3.3 Les analyses polyvariantes

Un des principaux inconvénients de la 0-CFA provient de ce que les différents appels à une fonction `f` sont « fusionnés » en un même point. Autrement dit, soit une fonction `f` définie par `let f x = x`, si l'on applique deux fois `f` dans le programme, une fois à l'entier 1 et une fois au booléen `true`, alors l'analyse mémorisera que $\nu(\mathtt{x}) = \{1; \mathtt{true}\}$. Dès lors deux problèmes majeurs surviennent.

Tout d'abord, dans le cas où la fonction `f` est polymorphe, les ensembles approximant, d'une part les valeurs des arguments, et d'autres part les résultats de cette fonction deviennent hétérogènes. Autrement dit, ils peuvent contenir des valeurs d'approximation de différents types. Ce regroupement a également pour effet d'augmenter la taille des ensembles manipulés, augmentant ainsi le temps nécessité par les opérations de base (union et soustraction) sur ces ensembles. Or nous avons vu précédemment que le coût des opérations ensemblistes n'était pas linéaire en la taille des arguments, d'où une perte d'efficacité importante.

En second lieu, considérons la fonction `let f x = raise x` et deux applications à des exceptions différentes (f $\mathtt{E}_1$) et (f $\mathtt{E}_2$). La 0-CFA nous garantit que $\nu(\mathtt{x}) = \{\mathtt{E}_1; \mathtt{E}_2\}$. Ainsi, chacune des deux applications de cette fonction sera considérée à tort comme levant les deux exceptions $\mathtt{E}_1$ et $\mathtt{E}_2$. Il s'ensuit une importante imprécision de l'analyse obtenue, incompatible avec les critères que nous avons exigés précédemment.

Ce problème se pose plus précisément pour toutes les analyses non *polyvariantes*. La solution consiste alors à différencier les sites d'appel des fonctions et ainsi de mémoriser les valeurs de leurs arguments et leurs valeurs de retour à des endroits séparés. Ainsi dans l'exemple ci-dessus, ayant deux instances d'application de `f`, nous aurions deux instances différentes de son argument `x`, chacune mémorisant une seule valeur.

Plusieurs analyses proposent une telle polyvariance. Citons entre autres : les *k-CFA* (où k signifie que l'analyse s'autorise à distinguer et mémoriser les k plus récents sites d'appel des fonctions) [21], l'*Infinitary Control Flow Analysis* [38], la *Iterative Flow Analysis* [41], le *Polymorphic Splitting* [22] (que nous détaillerons dans la section suivante).

3.3.1 Principe des analyses polyvariantes

Détaillons intuitivement et de façon simplifiée le processus qui se déroule lors de l'analyse d'un exemple simple : `let f x = x in f 100 ; ...f 200`. Étiquetons le premier appel de `f` (c'est-à-dire `f 100`) par α et le second par β. L'analyse mémorise alors que l'identificateur `x` peut contenir 100 au « site α » (c'est-à-dire, avec le formalisme vu précédemment pour la 0-CFA : $\nu(\mathtt{x}) = \{100\}$). De même, au « site β », $\nu(\mathtt{x}) = \{200\}$. Il faut désormais calculer les ensembles de valeurs retournés par `f`. Recherchons tous les « sites » auxquels l'argument de `f` est utilisé : nous trouvons ainsi α et β. Pour chacun de ces « sites », nous approximons de la même façon qu'en 0-CFA le corps de la fonction en utilisant comme approximation de son argument celle de ce « site », puis nous mémorisons l'ensemble résultat à ce même site. Ainsi, l'ensemble des valeurs retourné par `f` au « site α » est $\{100\}$ et celui du « site β » est bien $\{200\}$.

En terme d'implémentation, nous pouvons voir cette notion de « site » comme un index dans une table mémorisant tous les ensembles de valeurs pouvant être associés à un identificateur.

Autorisons-nous une comparaison entre évaluateur, 0-CFA et une analyse polyvariante en général, afin de conforter la compréhension intuitive du mécanisme : en 0-CFA la structure qui remplace *l'environnement de valeurs* (association identificateur-valeur) d'un évaluateur était un *environnement d'approximations* (association identificateur-approximation). Dans les analyses polyvariantes, cette structure est désormais remplacée par un environnement qui met en correspondance un identificateur avec une table d'approximations indexée par le « site ». On peut également comprendre cette nouvelle structure comme une structure d'environnement à deux clefs, qui envoie un couple (identificateur, « site ») sur une approximation.

3.3.2 Le polymorphic splitting

Le mécanisme proposé par [22] au travers du *polymorphic splitting* consiste à considérer les fonctions `f` liées par une expression `let` ainsi que toutes leurs occurrences $\mathtt{f}_1 \ldots \mathtt{f}_n$ syntaxiquement différentes dans le texte du programme. L'analyse va alors différencier chacun des ces sites d'appel en leur associant chacun un ensemble où accumuler les informations, distinct des autres. Nous voyons que la clef de cette analyse réside dans le choix des sites de différenciation au niveau des `let`, un peu comme ML introduit le polymorphisme au niveau de cette même construction.

Nous allons maintenant examiner plus en détail le fonctionnement de cette analyse. Dans un premier temps, nous devons étiqueter toutes les sous-expressions du programme de façon unique. Ensuite, afin de différencier les *sites d'appel* des fonctions, chaque identificateur de fonction lié par `let (rec)` doit également être étiqueté de façon unique. Différencier uniquement les sites d'appel de ces fonctions est suffisant puisqu'elles seules peuvent être utilisées en de multiples endroits (les fonctions anonymes comme (`fun x` $\rightarrow$ `...`), ne pouvant être utilisées qu'au moment de leur définition). Nous remarquerons que cette idée se place dans le même esprit que le typage de ML qui introduit du polymorphisme uniquement au niveau des fonctions liées par `let (rec)` en dupliquant le schéma de types y étant lié à chaque utilisation de ces fonctions.

Ainsi, l'expression `let f x = x in f 100 ; f 200` sera annotée sous la forme :

$\texttt{(let (f x = x}^{10}\texttt{)}^{2}\texttt{ in ((f}^{6}\texttt{ 100}^{7}\texttt{)}^{4}\texttt{ ; (f}^{8}\texttt{ 200}^{9}\texttt{)}^{5}\texttt{)}^{3}\texttt{)}^{1}$

On définit également une notion de *contour* (une version plus élaborée de nos « sites ») comme une suite d'étiquettes. Le but étant de construire pour chaque point du programme l'ensemble des valeurs possibles en ce point, un tel point de programme peut être alors référencé de deux façons différentes :

- par une étiquette et un contour, afin de pouvoir parler des valeurs possibles en une sous-expression
- ou par un nom d'identificateur et un contour, afin de considérer cette fois-ci les valeurs possibles pour un identificateur.

Intuitivement, un contour définit la trace statique permettant d'identifier un composant d'une expression à partir de son imbrication dans la structure de cette expression.

À l'instar de la 0-CFA l'analyse va se comporter somme un processus itératif visant à mémoriser l'ensemble des valeurs pouvant parvenir en tout point du programme, et ce jusqu'à obtention d'un point fixe. La fonction d'approximation d'une expression est maintenant paramétrée par un contour κ représentant le contour courant et un environnement ρ envoyant des identificateurs sur des contours, permettant ainsi de retrouver pour un identificateur le contour dans lequel considérer sa valeur (autrement dit, cet environnement va nous permettre de retrouver, pour un identificateur, quelle est l'approximation qui nous « intéresse », étant donné que nous avons vu que chaque identificateur est associé à plusieurs ensembles d'approximations, un par contour). L'utilité de ces deux nouveaux arguments sera plus apparente lorsque nous examinerons la fonction d'approximation d'une expression dans le cas de l'application.

La structure des approximations reste sensiblement identique à celle utilisée en 0-CFA à ceci près que les fonctions ne sont plus approximées uniquement par leur étiquette, mais par la donnée de trois informations : leur nom étiqueté, un contour et un environnement, ces deux derniers éléments étant le contour et l'environnement courants au moment de la création de la fermeture de la fonction. Ainsi, nous noterons une telle approximation de fonction sous la forme $\langle \mathtt{f}, \rho, \kappa \rangle$.

Dans ce qui suit, nous noterons les points de programme entre chevrons. Soit le langage suivant :

e^l ::=	
$\mid c^l$	constante
$\mid \mathtt{x}^l$	identificateur non lié par `let`
$\mid x^l_{l'}$	identificateur lié par `let` où l' représente l'étiquette de l'expression définissant cet identificateur
$\mid (e_1^{l_1}\ e_2^{l_2})^l$	application
$\mid (\texttt{function}\ x \rightarrow e_b^{l_b})^l$	fonction
$\mid (\texttt{let}\ x_l = e_1^{l_1}\ \texttt{in}\ e_2^{l_2})^l$	liaison d'identificateur

On définit la fonction d'approximation d'une expression e par $[\![e]\!]_{\rho\kappa}$ de la façon suivante :

• $[\![c^l]\!]_{\rho\,\kappa}$. Il faut mémoriser que le point de programme (correspondant à l'étiquette l) au contour courant peut dorénavant contenir la valeur constante c :

$$< l, \kappa > \stackrel{\cup}{\leftarrow} c$$

• $[\![x^l]\!]_{\rho\,\kappa}$. Dans le cas d'un identificateur non lié par `let`, il convient de retrouver l'approximation des valeurs pouvant être contenues dans cet identificateur au contour courant pour dire que le point de programme courant peut dorénavant contenir cette approximation lui aussi :

$$< l, \kappa > \stackrel{\cup}{\leftarrow} < x, \rho(x) >$$

• $[\![x^l_{l_0}]\!]_{\rho\,\kappa}$. Dans le cas où l'identificateur lié par `let` n'est pas une fonction, nous nous ramenons au cas précédent. Dans le cas contraire, pour toutes les fermetures $\langle l', \rho', \kappa' \rangle$ appartenant à $< \mathtt{x}, \rho(\mathtt{x}) >$, l'analyse va dire que le point de programme courant peut contenir cette fermeture « dans laquelle la valeur intéressante de `x` » se situe au contour courant étendu par l'étiquette courante (puisque les contours varient avec les sites d'utilisation des variables liées par `let`). On notera `.` l'extension des contours, ce qui peut être intuitivement vu comme une concaténation d'étiquettes.

$$< l, \kappa > \stackrel{\cup}{\leftarrow} \langle l', \rho', \kappa'[l_0 \leftarrow \kappa.l] \rangle$$

• $[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]_{\rho\,\kappa}$. Nous allons calculer e_1 et e_2 dans le contour qui nous « intéresse actuellement », (c'est-à-dire le contour courant) et avec l'environnement ρ courant. Pour chacune des valeurs se trouvant dans $[\![e_1^{l_1}]\!]_{\rho\,\kappa}$ qui sont des fermetures $\langle l', \rho', \kappa' \rangle$, nous recherchons l'expression qui l'a engendrée (c'est à dire l'expression d'étiquette l'). Cette expression est forcément de la forme : $(\mathtt{function}\ x \rightarrow e_b^{l_b})^{l'}$. On mémorise alors la nouvelle valeur de l'argument de cette fonction au même contour que celui de la fonction, de façon à dire que « pour cette instance de la fonction, l'approximation de son argument est ... ». Ainsi, l'argument d'une instance de fonction est lié au même contour que la fermeture représentant cette instance (puisque les fermetures enferment leur contour de création). Nous pouvons maintenant approximer le corps de la fonction, tout en précisant que l'approximation, « qui nous intéresse », de l'argument est celle de contour égal à celui où l'on vient de le mémoriser, à savoir κ'. Ainsi, nous obtenons l'algorithme suivant :

- Soit $a_1 = [\![e_1^{l_1}]\!]_{\rho\,\kappa}$
- Soit $a_2 = [\![e_2^{l_2}]\!]_{\rho\,\kappa}$
- $\forall \langle l', \rho', \kappa' \rangle \in a_1$
  - Soit $(\lambda \mathtt{x}.\, e_b^{lb})^{l'}$ l'expression d'étiquette l',
  - $< x, \kappa' > \stackrel{\cup}{\leftarrow} a_2$
  - $< l, \kappa > \stackrel{\cup}{\leftarrow} [\![e_b^{lb}]\!]_{(\rho' \oplus \{\mathtt{x} \mapsto \kappa'\})\,\kappa'}$

• $[\![(\mathtt{function}\ \mathtt{x} \rightarrow e^{l'})^l]\!]_{\rho\,\kappa}$. Il suffit de mémoriser que le point de programme courant peut contenir une fermeture qui enfermera son contour et son environnement de définition restreint aux identificateurs libres dans le corps de la fonction.

$$< l, \kappa > \stackrel{\cup}{\leftarrow} \langle l, \rho_{|\mathtt{FV}(e)}, \kappa \rangle$$

• $[\![(\mathtt{let}\ \mathtt{f} = e_1^{l_1}\ \mathtt{in}\ e_2^{l_2})^l]\!]_{\rho\,\kappa}$. L'analyse commence par inférer l'approximation de l'identificateur défini en prenant soin d'étendre le contour courant, puis elle calcule celle de l'expression résultat dans un environnement liant cet identificateur au contour courant. Rappelons que l'on note `.` l'extension des contours.

$$< \mathtt{f}, \kappa > \stackrel{\cup}{\leftarrow} [\![e_1^{l_1}]\!]_{\rho\,\kappa.l}$$

$$< l, \kappa > \stackrel{\cup}{\leftarrow} [\![e_2^{l_2}]\!]_{(\rho \oplus \{\mathtt{f} \mapsto \kappa\})\,\kappa}$$

Notons que cette présentation dérivée du formalisme originel présente l'analyse sous forme purement fonctionnelle et ne peut être utilisée directement sous peine de faire boucler l'analyseur dans le cas de fonctions récursives. Par exemple dans le cas de l'application, plutôt que de chercher à approximer directement le corps des fonctions appliquées, il convient comme en 0-CFA de retourner l'approximation du corps trouvée à l'étape précédente et d'effectuer une seconde phase au sein de chaque itération afin de mettre à jour les approximations de retour des fonctions. Néanmoins, il est relativement aisé d'adapter cette formulation afin de retrouver le processus itératif en deux passes de la 0-CFA.

3.3.3 Résultats pratiques

Cette analyse présente l'avantage de résoudre le problème des fusions de sites d'appels. Nous l'avons tout d'abord implémentée en tant que prototype sur un sous-ensemble d'Objective Caml traitant tout le langage de base. Le gain de vitesse engendré par la diminution de la taille des ensembles manipulés s'est avéré insuffisant pour satisfaire nos critères de rapidité initiaux. Afin d'accélérer l'analyse, nous avons tenté de dégrader les approximations de manière à approximer toutes les valeurs de base par $\perp$. La perte de précision s'ensuivant est notable pour un gain de vitesse mineur. En effet, ceci nous interdit tout espoir d'approximer précisément les arguments d'exceptions.

Une extension aux objets d'Objective Caml a également été réalisée dans ce prototype et semble montrer que malgré sa complexité, une telle technique d'analyse peut être étendue au-delà des constructions de base des langages fonctionnels.

Chapitre 4

Un système de types pour analyser les exceptions

DANS CE CHAPITRE, nous abordons le principe de notre analyse d'exceptions non rattrapées en elle-même. Pour commencer, nous introduisons progressivement les idées de base qui ont motivé la détermination de l'architecture de notre analyse. Nous verrons en particulier de quelle manière cette architecture a été pensée afin de satisfaire les différentes exigences que nous avons énoncées au chapitre 1. Ce n'est qu'ensuite que nous exposerons formellement (algèbre, règles, etc...) le modèle de l'analyse afin de conserver un caractère pédagogique et progressif à cette thèse.

Sans déflorer le sujet ni gâcher le suspense, disons d'ores et déjà que notre analyse se présente comme un algorithme d'inférence de types non standards, ou *système d'effets*, dans le même ordre d'esprit que les travaux de [34, 56, 63]. Le système de types que nous allons étudier utilise un mécanisme d'unification sur des *rangées* [61, 43, 47], à la fois pour garder trace des effets (ensembles d'exceptions pouvant s'échapper) des expressions, et pour raffiner les types habituellement utilisés en ML en leur adjoignant une information plus précise quant aux valeurs en lesquelles les expressions qu'ils étiquettent peuvent s'évaluer.

Le langage sur lequel nous désirons travailler ayant été introduit dans le chapitre 2, nous allons directement nous intéresser aux problèmes de typage de ce langage dans notre nouveau système. Aussi commençons-nous par présenter le système de types lui-même (c'est-à-dire les spécifications de l'analyse).

4.1 L'élaboration du mécanisme

4.1.1 La base de tout : les effets

Nous avons désormais bien en tête que le but de notre analyse est donc de détecter les exceptions pouvant survenir lors de l'évaluation d'un programme, lui-même composé comme il se doit d'une suite d'expressions du langage.

Chacune de ces expressions est alors susceptible, lors de son évaluation, de lever un certain nombre d'exceptions. La première idée est donc d'attacher à chaque expression **un effet** représentant une approximation de l'ensemble des exceptions pouvant s'en échapper, un peu comme chaque expression possède un type en ML.

Ainsi, l'effet associé à l'expression `raise Not_found` sera un ensemble de la forme : {`Not_found`}, alors que son type sera, comme d'habitude en ML, α.

Nous venons de voir l'effet direct d'une expression. Il est maintenant nécessaire d'être capable de déterminer l'ensemble des exceptions pouvant s'échapper lors d'une application de fonction. En effet, la définition d'une fonction ne lève pas d'exception puisqu'il s'agit simplement de la création d'une fermeture. Toutefois le corps de cette fonction peut, lorsqu'il est évalué au cours d'une application de cette fonction, provoquer la levée d'exceptions. C'est la notion **d'effet latent** des fonctions.

Par exemple, la définition de la fonction `let f = fun x → 5 / x` ne lève clairement aucune exception. Par contre, l'application (`f` 0), elle, déclenchera manifestement une exception `Division_by_zero`. Nous voyons ici que les exceptions pouvant être levées dans le corps de la fonction sont « en attente », d'où leur qualification d'effet latent.

Afin de rendre compte de ce comportement, à l'instar des autres systèmes d'effets, nous allons étiqueter le type « flèche » des fonctions par leur effet latent, c'est-à-dire par l'ensemble des exceptions qui pourront éventuellement s'échapper lors de l'utilisation de ces fonctions.

Ainsi la fonction `f` ci-dessus aura pour type $\texttt{int} \xrightarrow{\{\texttt{Division\_by\_zero}\}} \texttt{int}$ et l'expression correspondant à sa définition aura pour effet $\emptyset$, l'ensemble vide d'exceptions.

Munie de cette nouvelle façon de représenter le type des fonctions, notre algèbre de types devient donc :

τ ::=	α	variable de type
	$\mid$ `int`	type de valeurs entières
	$\mid$ `exn`	type d'exceptions
	$\mid \tau_1 \overset{E}{\to} \tau_2$	type des fonctions d'argument de type τ_1, de résultat de type τ_2 et pouvant lever l'ensemble d'exceptions E lors de leur application

Nous laissons volontairement la notion « d'ensemble E » non formellement définie ; celle-ci sera raffinée au fur et à mesure de cette présentation du support de notre analyse.

4.1.2 L'union d'effets

Considérons une construction conditionnelle de la forme `if` $expr_1$ `then` $expr_2$ `else` $expr_3$. Le comportement, en terme d'exceptions levées, que l'on attend pour une telle expression, est que l'ensemble des exceptions potentiellement levé soit **l'union** des effets de chacune des sous-expressions $expr_1$, $expr_2$ et $expr_3$.

De même, nous avons vu que les fonctions étaient annotées par leur effet latent lors de leur définition. Il convient maintenant de comprendre comment gérer l'application de ces fonctions en terme de détection d'exceptions non rattrapées. L'intuition nous tente à penser que lors d'une application de fonction, l'ensemble des exceptions pouvant être levé est composé de l'effet latent de la fonction, de l'effet de l'argument appliqué à cette fonction, ainsi que de l'effet de l'expression fonctionnelle elle-même. En effet dans le programme suivant :

```
let f x = if ... then raise Exn1 else x in
f (if ... then raise Exn2 else 0)
```

l'ensemble des exceptions pouvant s'échapper est $\{\texttt{Exn}_1; \texttt{Exn}_2\}$, la première provenant du corps de `f` et la seconde de l'expression `if` passée en argument à `f`. Ici encore, nous avons besoin d'un mécanisme d'union entre effets, c'est-à-dire entre ensembles d'exceptions.

Jusqu'à présent un mécanisme d'effets et une représentation d'ensembles simple comme il en existe de nombreuses (listes triées, B-arbres, etc...) semblent se montrer satisfaisants.

Insistons sur le fait que les ensembles d'exceptions que nous manipulons représentent une *approximation* des ensembles réels d'exceptions pouvant s'échapper lors de l'évaluation d'une expression, et non pas une description exacte de ces ensembles. En effet, notre analyse ne « calcule » pas le programme qui lui est soumis. Comme nous l'avons vu avec l'exemple de la construction conditionnelle, l'analyse considère que chacune des sous-expressions peut lever ses exceptions, indépendamment de l'ordre d'évaluation du langage et indépendamment de la valeur de vérité de la condition.

Ainsi si nous avions une expression permettant la construction de couples, l'effet de celle-ci serait l'union des effets de chacune des deux expressions membres du couple, indépendamment de l'ordre dans lequel elles sont évaluées.

4.1.3 Les fonctions d'ordre supérieur : polymorphisme d'effet

Comme nous l'avons présenté dans le chapitre 1, ML autorise la définition de fonctions d'ordre supérieur. Considérons la fonction `f` définie par `let f g = g 1`, prenant en argument une fonction `g`.

Lors d'une application de `f`, l'ensemble des exceptions pouvant s'échapper est entre autre composé de l'ensemble des exceptions pouvant s'échapper de son corps, c'est-à-dire de l'application de l'argument `g` à 1. Autrement dit, l'effet latent de `f` est l'effet latent de son argument `g`. Or ce dernier est inconnu *a priori* : c'est une forme de « variable d'effet » ou de « variable d'ensemble » . Nous avons donc besoin d'une forme de **polymorphisme** sur les ensembles d'exceptions.

Ainsi le type de la fonction `f`, devant refléter le fait que son effet latent provient de `g` doit-il avoir une forme : $\forall \alpha, \rho.\, (\mathtt{int} \xrightarrow{\rho} \alpha) \xrightarrow{\rho} \alpha$, dans laquelle ρ est une variable pouvant être instanciée en n'importe quel ensemble d'exceptions.

Considérons une fonction `f'` définie par `let f' y = raise Not_found`. Son effet latent est alors l'ensemble $\{\mathtt{Not\_found}\}$, d'où son type : $\forall \alpha, \beta.\, \alpha \xrightarrow{\{\mathtt{Not\_found}\}} \beta$. Appliquons la fonction `f` à `f'` : la variable d'effet ρ se retrouve alors instanciée par l'ensemble $\{\mathtt{Not\_found}\}$ via sa présence en tant qu'effet latent de l'argument fonctionnel de `f`. Cette variable apparaissant également en tant qu'effet latent de `f`, l'effet de notre application sera égal à cet ensemble, dénotant le fait que l'exception `Not_found` peut effectivement être levée.

Donc, en plus d'une représentation simple d'ensembles comme nous l'avons évoqué précédemment, nous avons désormais besoin d'une notion de variable d'ensembles d'exceptions.

4.1.4 Les exceptions valeurs de première classe

Les exceptions sont levées grâce à la primitive `raise` dont le type ML est $\forall \alpha.\, \mathtt{exn} \rightarrow \alpha$. Nous cherchons maintenant à comprendre quel doit être le type de cette primitive dans notre système. La sémantique de `raise` est de « lever les exceptions qu'elle reçoit en argument ». Comment faire alors le lien entre l'argument d'exception passé à `raise` et son effet latent ?

L'idée consiste alors à établir un lien entre le type `exn` des exceptions et l'effet latent de la fonction. Autrement dit, nous voudrions que `exn` puisse encapsuler des ensembles tels qu'ils apparaissent dans les effets. Nous allons donc raffiner le type `exn` des exceptions en lui ajoutant une information d'ensemble, information représentant les constructeurs d'exceptions pouvant apparaître dans les valeurs de ce type (similairement à [9]). Dans la suite de cette thèse, nous emploierons souvent le terme **annotation** pour désigner cette information servant à raffiner le type.

Ainsi, en ML, la déclaration d'une exception `Error` par la construction `exception Error` permettait de générer un nouveau constructeur (`Error` en l'occurrence) de type `exn`. Désormais ce constructeur aura le type `exn[{Error}]`.

Ce raffinement du type exception nous permet maintenant de manipuler les exceptions comme des valeurs de première classe, place qu'elles occupent en ML. Ainsi si nous définissons deux exceptions `Foo` et `Bar`, l'expression `if ... then Foo else Bar` aura un type exception dont l'ensemble des valeurs possibles sera composé de l'union des ensembles de valeurs d'exceptions possibles pour chaque branche de la conditionnelle. C'est-à-dire : $\texttt{exn}[\{\texttt{Foo};\, \texttt{Bar}\}]$.

Nous pouvons désormais donner le type de la primitive `raise`, qui combine à la fois le polymorphisme sur les ensembles et l'annotation du type `exn`. Ainsi `raise` sera de type : $\forall \alpha, \rho.\, \texttt{exn}[\rho] \xrightarrow{\rho} \alpha$.

4.1.5 Une première synthèse : les rangées

Nous venons de voir que la structure des annotations dont nous avions besoin pour représenter les ensembles d'exceptions nécessitait deux caractéristiques particulières : la possibilité de faire l'union d'ensembles et la possibilité de bénéficier d'une forme de polymorphisme sur ces ensembles.

De plus, l'intégration de ces ensembles au sein de types suggère la mise en place d'un système de types modifié par rapport à celui dont bénéficie ordinairement ML. Ce dernier permet l'inférence automatique des types du programme à partir d'un algorithme qui, comme nous le verrons ultérieurement, repose fortement sur un mécanisme à base d'unification entre termes du premier ordre. Aussi, voudrions-nous que notre système modifié autorise lui aussi une synthèse automatique des types des programmes.

L'idée est d'utiliser un mécanisme de **rangées** [61, 43, 47, 57] (similaires à celles utilisées pour le typage des enregistrements extensibles) pour représenter nos ensembles d'exceptions. L'intérêt principal d'un tel cadre de travail provient du fait qu'il fournit, entre autre, les trois propriétés que nous venons d'énoncer (union, polymorphisme et existence d'un algorithme d'unification et d'inférence de rangées).

Nous verrons plus en détail la forme des rangées que nous utilisons lorsque nous aborderons ultérieurement la présentation formelle du système de types sur lequel est basée notre analyse. Toutefois, nous jugeons bon de donner ici un aperçu récapitulatif de leur structure ainsi qu'une vue globale de notre nouveau système de types (incluant l'annotation du constructeur de type `exn`) afin d'aider à la compréhension intuitive du mécanisme global. Ainsi, notre algèbre de types et nos rangées sont désormais engendrées par la grammaire suivante :

Expressions de type :

τ	$::= \alpha$	variable de type
	$\mid \texttt{int}$	type de valeurs entières
	$\mid \texttt{exn}[\varphi]$	type d'exceptions
	$\mid \tau_1 \xrightarrow{\varphi} \tau_2$	type fonctionnel

Rangées :

φ	$::= \rho$	variable de rangée
	$\mid \epsilon;\, \varphi$	l'élément ϵ plus ce qui est dans la rangée φ

Éléments de rangée :

ϵ	$::= C$	exception constante

Il est de plus nécessaire d'ajouter une théorie équationnelle sur ces rangées de façon à exprimer que l'ordre d'apparition des éléments ne présente pas d'importance :

$$\epsilon_1;\, \epsilon_2;\, \varphi = \epsilon_2;\, \epsilon_1;\, \varphi$$

Ainsi la rangée $C_1;\, C_2;\, C_3;\, \rho$ est-elle égale à la rangée $C_2;\, C_3;\, C_1;\, \rho$.

Nous voyons donc qu'une *rangée* est une séquence $\epsilon_1;\, \epsilon_2;\, \rho$ *d'éléments de rangée* et se trouve toujours terminée par une *variable de rangée*. Pour le moment, les éléments de rangée sont limités à des noms de constructeurs d'exceptions constantes. Ainsi, le type $\texttt{exn}[C_1;\, C_2;\, C_3;\, \rho]$, annoté par une rangée, représente l'ensemble des exceptions $\{C_1;\, C_2;\, C_3\}$ possiblement suivi de ce en quoi la variable finale, ρ, peut être instanciée au besoin. En l'absence d'instanciation particulière, celle-ci restera *non contrainte*, laissant cette rangée dénoter l'ensemble uniquement composé des trois constructeurs d'exceptions que nous venons d'énumérer.

Les rangées et les variables de rangées fournissent à la fois du polymorphisme sur les ensembles et une forme d'union d'ensembles par l'intermédiaire d'un mécanisme d'unification. Par exemple, deux rangées $(\epsilon_1;\, \rho_1)$ et $(\epsilon_2;\, \rho_2)$, qui représentent informellement les ensembles $\{\epsilon_1\}$ et $\{\epsilon_2\}$ respectivement, s'unifient en la rangée $(\epsilon_1;\, \epsilon_2;\, \rho)$ représentant l'ensemble $\{\epsilon_1;\, \epsilon_2\}$ via la substitution $\{\rho_1 \leftarrow (\epsilon_2;\, \rho);\, \rho_2 \leftarrow (\epsilon_1;\, \rho)\}$.

Une présentation complète du mécanisme originel des rangées ainsi que de ses propriétés pourra être trouvée dans [47, 61]. Bien que notre algèbre de rangées diffère par rapport à ces travaux, l'idée sous-jacente reste la même. Nous réaliserons une présentation formelle de ce mécanisme dans le cadre de notre algèbre et établirons ses propriétés ultérieurement, dans les chapitres qui vont suivre.

De la même façon que nous avons défini les schémas de types dans le système de ML conventionnel, nous les définissons sur notre algèbre en étendant l'ensemble des variables universellement quantifiées aux variables de rangées.

Schémas de types :

$$\sigma \ ::= \forall \alpha_i, \rho_j.\, \tau$$

Il semblerait que la représentation de nos ensembles par des rangées nous prive de la notion d'ensemble vide. En effet, avant de déterminer la façon dont nous allions encoder les ensembles d'approximation, nous utilisions simplement la notation d'ensemble vide $\emptyset$ qui manifestement n'existe pas dans notre algèbre de rangées. Pourtant notre fonction d'addition ne lève aucune exception lors de son application.

Pour la même raison que le schéma de types $\forall \alpha.\, \alpha$ dénote un ensemble vide de valeurs ([50, 60]), nous pouvons dire que $\forall \rho.\, \rho$ dénote un ensemble vide d'éléments de rangée. Lorsque cette rangée étiquette une flèche de type fonction, ceci revient bien à exprimer que la fonction ne lève aucune exception. Plus exactement, les variables universellement quantifiées (c'est-à-dire généralisées) n'apparaissant qu'en position positive dans les schémas de types, représentent des ensembles vides de valeurs.

Cette notion de variable dénotant un ensemble vide de valeurs nous permet désormais de mieux comprendre en quoi le schéma de types $\forall \rho.\, \texttt{exn}[C_1;\, C_2;\, C_3;\, \rho]$ représente l'ensemble <u>uniquement</u> composé des exceptions C_1, C_2 et C_3. En effet, la variable ρ étant universellement quantifiée en position uniquement positive, elle ne pourra désormais plus subir aucune instanciation.

Ainsi, elle restera non contrainte (c'est-à-dire qu'elle ne pourra rien « rajouter » à l'ensemble) et nous pouvons donc « l'oublier ».

4.1.6 Les exceptions paramétrées : annotations généralisées

Nous avons vu que ML permettait non seulement la définition de constructeurs d'exceptions constants, mais aussi de constructeurs d'exceptions paramétrés. Ainsi les déclarations suivantes :

```
exception Foo of int
exception Bar of string
exception Gee of exn
```

définissent-elles trois nouveaux constructeurs d'exceptions, le premier permettant de colporter une valeur entière, le second une valeur de chaîne de caractères et le dernier une valeur d'exception. Il est ainsi possible de construire des valeurs d'exceptions de la forme : $\texttt{Foo}(42)$, $\texttt{Bar}(\text{"}\textit{coucou}\text{"})$, ou $\texttt{Gee}(\texttt{Foo}(13))$.

Comme expliqué en section 1, dans le cadre d'une exception paramétrée, nous souhaiterions que notre analyse soit capable, non seulement de mémoriser le constructeur de tête représentant la valeur d'exception, mais également les valeurs des arguments du constructeur de cette exception.

Nous voyons ici que les constructeurs d'exceptions pouvant *a priori* être paramétrés par n'importe quelle valeur du langage, la seule alternative consiste à étendre notre mécanisme d'annotation du type `exn` à tous les constructeurs de types. Ainsi le type `int` va-t'il se voir lui aussi raffiné par une annotation de rangée.

Puisque d'une part nous généralisons les annotations à tous les constructeurs de types (c'est-à-dire, dans notre langage de travail, au type `int`) et que d'autre part nous introduisons des constructeurs d'exceptions paramétrés, il nous faut désormais étendre nos rangées afin de prendre en compte ces deux modifications.

Nous étendons donc les éléments de rangée par des valeurs entières, et par des noms de constructeurs d'exceptions eux-mêmes annotés par le type de leur argument. Nos rangées et nos types prennent donc la forme suivante :

Expressions de type :

τ	$::= \ldots$	
	$\mid \texttt{int}[\varphi]$	type de valeurs entières

Rangées :

φ	$::= \ldots$	

Éléments de rangée :

ϵ	$::= i$	constante entière
	$\mid C$	exception constante
	$\mid D(\tau)$	exception paramétrée

Munis de cette nouvelle extension de notre système de types, nous pouvons maintenant dire que le schéma de types $\forall\rho,\rho'.\,\texttt{exn}[D(\texttt{int}[3;\ 4;\ \rho]);\ \rho']$ représente l'ensemble d'exceptions $\{D(3);\ D(4)\}$.

4.1.7 L'approximation des primitives : $\top$

Maintenant que nous avons décidé d'approximer également les valeurs du langage autres que seulement les exceptions, nous souhaiterions donner un type à la fonction d'addition entre entiers. À moins que celle-ci n'ait été redéfinie par l'utilisateur, cette fonction fait partie des primitives du langage et donc sa définition n'est pas accessible à l'analyseur. Ceci signifie en particulier que nous ne pouvons pas déterminer automatiquement l'approximation des valeurs qui en ressortent en fonction de celles qui lui sont fournies en argument.

Autrement dit, nous devons être capables de dire que la fonction « + » prend en arguments deux entiers et retourne un entier dont nous ne connaissons pas la valeur. Plus précisément la valeur entière retournée peut être **n'importe quelle valeur entière**.

Un autre exemple, peut-être encore plus parlant intuitivement, est celui des fonctions de génération de nombres aléatoires (*random*). Par définition, il est impossible de prévoir statiquement et à l'avance le résultat d'une telle fonction : n'importe quel nombre peut être effectivement obtenu.

Actuellement, notre algèbre de rangées ne nous permet pas de décrire un tel ensemble de valeurs. En fait, ne pas être capable d'approximer la valeur du résultat de l'addition ou de la fonction de génération de nombre aléatoire revient à dire que le type de leur résultat est `int`, annoté par *l'ensemble de toutes les valeurs entières possibles.*

Nous étendons donc la structure de nos rangées par l'approximation $\top$ (*top* qui représente alors l'ensemble de toutes les valeurs possibles du type qu'il annote. Les rangées prennent désormais la forme suivante :

Rangées :

φ	$::= \rho$	variable de rangée
	$\mid \top$	tous les éléments possibles
	$\mid \epsilon;\ \varphi$	l'élément ϵ plus ce qui est dans φ

À ce point de développement du système, deux remarques sont nécessaires. D'une part $\top$ est l'élément absorbant de l'opération d'union entre ensembles (donc entre rangées). Ceci signifie que nous devons rajouter, dans la théorie équationnelle gérant les rangées, une règle de la forme :

$$i;\ \top = \top$$

D'autre part, la présence de $\top$, donc l'équation ci-dessus, n'est nécessaire que pour les types de signature infinie (ou au moins extrêmement vaste). Pour les types de signature finie (par exemple `bool`, le type des valeurs de vérité, comportant les deux constructeurs `true` et `false`), l'ensemble de toutes les valeurs possibles peut alors être obtenu par la rangée énumérant simplement tous les constructeurs de ce type. Ainsi dans le langage que nous utilisons pour notre présentation, l'utilisation $\top$ est-elle restreinte aux valeurs représentant des constantes entières. Comme nous le verrons plus en détail dans la section suivante (cf. 4.1.8), cette restriction est justifiée par la correction de notre analyse.

Pour en terminer avec l'utilisation de $\top$ dans les rangées, il est intéressant de noter que celui-ci n'apparaîtra ainsi pas dans les rangées annotant le type `exn`. En effet, les exceptions appartiennent à un type dont la signature est finie et peut être déterminée en regardant toutes les déclarations de constructeurs effectuées dans le programme à analyser. Ceci présente le grand intérêt de nous prémunir contre l'apparition d'expressions dont l'effet serait $\top$, c'est-à-dire dont nous serions incapables d'exprimer précisément l'ensemble des exceptions pouvant s'en échapper (perte d'information maximale).

Munis de notre nouvelle algèbre de rangée, nous voyons que `int`$[\top]$ représente le type de toutes les valeurs entières, et nous pouvons désormais donner le type de notre fonction d'addition comme nous le cherchions initialement :

$$+ : \forall \rho_1, \rho_2, \rho_3, \rho_4 . \, \mathtt{int}[\rho_1] \stackrel{\rho_2}{\to} \mathtt{int}[\rho_3] \stackrel{\rho_4}{\to} \mathtt{int}[\top]$$

L'examen du schéma de types de cette fonction nous permet en outre de nous apercevoir qu'elle ne peut lever aucune exception au cours d'une de ses applications. En effet, les variables ρ_2 et ρ_4 apparaissant dans le schéma de types de celle-ci sont généralisées et apparaissent uniquement en position positive. Comme nous l'avons expliqué précédemment (cf. section 4.1.5), ceci dénote bien le fait que ces variables représentent des ensembles vides d'exceptions, donc qu'une application de « + » ne peut en aucun cas générer d'exception *en provenance de son propre code* (en revanche des exceptions peuvent survenir en provenance des arguments auxquels elle est appliquée, mais ceci ne concerne pas le comportement de la fonction proprement dite).

4.1.8 Le sortage des rangées : l'algèbre de types finale

La description que nous avons donnée de nos rangées doit être maintenant légèrement modifiée. En effet, afin d'assurer l'existence d'unificateurs principaux et de types principaux, il est nécessaire que les rangées satisfassent un certain nombre d'invariants structurels. Nous n'exposerons pas ici les justifications théoriques de cette nécessité, de nombreux travaux antérieurs traitant de ce sujet. Pour en savoir plus, le lecteur pourra se reporter à [61, 43, 47].

Ces invariants définissent une notion de *bonne formation* des rangées et peuvent se résumer par les quatre points suivants :

1. Un élément de rangée (constante entière ou constructeur d'exceptions) ne peut apparaître au plus qu'une seule fois au sein de cette rangée. Ainsi, $(D(\tau);\ D(\tau');\ \varphi)$ n'est pas une rangée bien formée.
2. Une variable de rangée ρ est toujours précédée du même ensemble d'éléments de rangée (encore une fois, constantes entières ou constructeurs d'exceptions) dans toutes les rangées où elle apparaît. Par exemple, nous ne pouvons avoir à la fois $(1;\ \rho)$ et $(2;\ \rho)$ dans une même dérivation.
3. Une rangée φ annotant un type entier $\mathtt{int}[\varphi]$ ne peut contenir que des éléments de rangée i étant des constantes entières.
4. Une rangée φ annotant un type exception $\mathtt{exn}[\varphi]$ ou un type fonctionnel $\tau_1 \stackrel{\varphi}{\to} \tau_2$ ne peut contenir que des éléments de rangée C, D étant des constructeurs d'exceptions constants ou paramétrés. De plus une telle rangée ne peut se terminer par $\top$.

Les invariants (1) et (2) sont déjà bien connus de travaux antérieurs sur les types d'enregistrements extensibles [47]. Les invariants (3) et (4) sont moins courants. Ils assurent une distinction claire entre les annotations du type entier `int` (composées de constantes entières et éventuellement de $\top$) et les annotations du type exception (`exn`) (composées de constructeurs constants ou paramétrés et jamais de $\top$).

Puisque $\top$ absorbe uniquement les éléments de rangée de type entier (équation 2), nous ne voulons clairement pas le voir apparaître dans des rangées contenant des constructeurs d'exceptions C, D.

Rappelons que la garantie que les rangées annotant les types `exn` ne comporteront jamais $\top$ nous permet d'assurer que notre analyse ne déclarera jamais que l'ensemble des exceptions pouvant s'échapper lors d'un calcul est un ensemble totalement « inconnu », ce qui est relativement satisfaisant quant à la précision de notre analyse.

Mais la raison fondamentale de cette interdiction de $\top$ dans de telles rangées provient de l'impossibilité de lui associer une équation cohérente dans notre théorie. En effet, l'équation $D(\tau);\ \top = \top$ à laquelle on pourrait penser de prime abord est incorrecte car elle permet des déductions de la forme $D(\alpha);\ \top = \top = D(\beta);\ \top$, ce qui mène à des dérivations de typage incohérentes.

Plus précisément, considérons la définition de type `type` α `t = A of` α et autorisons les rangées étiquetant le type `t` à contenir $\top$. Le schéma de types que nous attribuons au constructeur `A` est : $\mathtt{A} : \forall \alpha, \rho_0, \rho_1 .\ \alpha \xrightarrow{\rho_0} \alpha\ \mathtt{t}[\mathtt{A}(\alpha);\ \rho_1]$. Soit une valeur v de type $\mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\top]$. Unifions le type de v avec une instance du schéma de `A` (ce qui se produit au cours du typage d'une expression de la forme `match` v `with A x` $\rightarrow$ `...` par exemple).

$$\begin{array}{cccc} \mathtt{int}[1;\ \rho_2] & \mathtt{t} & [\top] & (1) \\ & \cup & & \\ \alpha & \mathtt{t} & [\mathtt{A}(\alpha);\ \rho_1] & (2) \end{array}$$

Selon que nous unifions d'abord les parties droites ou les parties gauches de ces deux types, nous obtenons les types finals suivants :

- *À gauche d'abord :* nous obtenons les substitutions $\alpha \leftarrow \mathtt{int}[1;\ \rho_2]$ puis $\rho_1, \rho_2 \leftarrow \top$. Appliquons-les au type (1), nous obtenons :
$$\begin{array}{lcll} \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\top] & \rightsquigarrow & \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\top] & (\text{par } \alpha \leftarrow \mathtt{int}[1;\ \rho_2]) \\ & \rightsquigarrow & \mathtt{int}[1;\ \top]\ \mathtt{t}[\top] & (\text{par } \rho_1, \rho_2 \leftarrow \top) \\ & = & \mathtt{int}[\top]\ \mathtt{t}[\top] & (\text{par l'équation } i;\ \top = \top) \end{array}$$
Appliquons maintenant ces substitutions au type (2) et nous obtenons :
$$\begin{array}{lcll} \alpha\ \mathtt{t}[\mathtt{A}(\alpha);\ \rho_1] & \rightsquigarrow & \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\mathtt{A}(\mathtt{int}[1;\ \rho_2]);\ \rho_1] & (\text{par } \alpha \leftarrow \mathtt{int}[1;\ \rho_2]) \\ & \rightsquigarrow & \mathtt{int}[1;\ \top]\ \mathtt{t}[\mathtt{A}(\mathtt{int}[1;\ \top]);\ \top] & (\text{par } \rho_1, \rho_2 \leftarrow \top) \\ & = & \mathtt{int}[\top]\ \mathtt{t}[\mathtt{A}(\mathtt{int}[\top]);\ \top] & (\text{par l'équation } i;\ \top = \top) \\ & = & \mathtt{int}[\top]\ \mathtt{t}[\top] & (\text{par l'équation erronnée}) \end{array}$$
- *À droite d'abord :* les substitutions sont alors $\rho_1 \leftarrow \top$ puis $\alpha \leftarrow \mathtt{int}[1;\ \rho_2]$. Appliquons-les au type (1), nous obtenons alors :
$$\begin{array}{lcll} \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\top] & \rightsquigarrow & \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\top] & (\text{par } \rho_1 \leftarrow \top) \\ & \rightsquigarrow & \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\top] & (\text{par } \alpha \leftarrow \mathtt{int}[1;\ \rho_2]) \end{array}$$
Appliquons maintenant ces substitutions au type (2) et nous obtenons :
$$\begin{array}{lcll} \alpha\ \mathtt{t}[\mathtt{A}(\alpha);\ \rho_1] & \rightsquigarrow & \alpha\ \mathtt{t}[\mathtt{A}(\alpha);\ \top] & (\text{par } \rho_1 \leftarrow \top) \\ & \rightsquigarrow & \mathtt{int}[1;\ \rho_2]\ \mathtt{t}[\mathtt{A}(\mathtt{int}[1;\ \rho_2]);\ \top] & (\text{par } \alpha \leftarrow \mathtt{int}[1;\ \rho_2]) \end{array}$$

En effet dans le second cas, la variable de type α ne portant pas d'annotation, nous ne pouvons pas lui faire refléter le fait que l'on ne saura plus rien de précis (perte d'information maximale) quant aux valeurs du type par lequel elle sera instanciée. Autrement dit, au moment où cette variable sera instanciée par un type, nous aurons oublié le fait que l'ensemble des valeurs présentes de ce type, est l'ensemble de *toutes ses valeurs possibles*. Dans un tel cas, nous perdons ainsi la principalité du typage.

Si ML possédait une notion de sous-typage ainsi qu'un supertype $\top$ de tous les types, alors l'équation correcte serait $D(\top);\ \top = \top$. Cette équation permet à $\top$ d'absorber tous les $D(\tau)$ (puisque $D(\tau);\ \top <: D(\top);\ \top = \top$), mais n'autorise l'expansion de $\top$ que dans $D(\top);\ \top$, spécifiant ainsi correctement qu'aucune information n'est disponible quant à l'argument du constructeur D.

Les drapeaux de présence

Considérons maintenant la fonction suivante : `f = fun 0` $\rightarrow$ `1 | x` $\rightarrow$ `x`. Intuitivement, nous nous attendons à ce qu'elle possède le schéma de types : $\forall \rho_0, \rho_1 .\ \mathtt{int}[0;\ \rho_0] \xrightarrow{\rho_1} \mathtt{int}[1;\ \rho_0]$, exprimant le fait que cette dernière ne lève aucune exception et retourne en résultat une valeur entière pouvant être soit 1, soit l'argument qui lui a été fourni *excepté* 0.

Ce dernier point est dénoté par le fait que la rangée du résultat partage la même variable ρ_0 que la rangée de l'argument, dans laquelle cette variable a été explicitement « séparée » de la valeur de 0. Autrement dit, dans la rangée de l'argument de `f`, apparaît clairement le fait que 0 existe séparément de ρ_0, et comme les rangées ne doivent pas contenir plusieurs fois les mêmes éléments, ceci implique que ce par quoi la variable ρ_0 peut être instanciée ne contient pas à nouveau l'élément 0.

Toutefois un tel type ne satisfait pas les invariants nécessaires à la bonne formation des rangées, et plus particulièrement, il viole celui demandant à ce qu'une variable de rangée soit toujours précédée du même ensemble d'éléments de rangée dans toutes les rangées où elle apparaît (invariant 1). En effet, dans notre exemple, la variable ρ_0 est une fois précédée de 0 et l'autre fois de 1.

Nous voudrions que les éléments 0 et 1 apparaissent dans toutes les rangées faisant intervenir la variable ρ_0. Pourtant, nous voudrions également dire que l'élément 0 ne peut pas apparaître dans la rangée décrivant le résultat de `f`. La solution consiste à toujours laisser les éléments de rangée exister dans ces rangées, mais de les affubler d'un **drapeau** disant s'ils sont réellement présents ou s'ils ne sont là « que pour des questions de conservation des invariants nécessaires à la correction du système ».

Ces drapeaux seront au nombre de deux. Tout d'abord `Pre`, spécifiant que l'élément fait effectivement partie de l'ensemble que la rangée où il apparaît décrit. En second lieu, nous introduisons des **variables de présence**, δ pouvant être instanciées, si nécessaire, soit par `Pre`, soit par une autre variable de présence.

Que signifient alors ces variables ? Intuitivement, elles permettent de décrire que la présence de l'élément dans la rangée n'est pas encore déterminée à un moment donné. En particulier, si une telle variable devient universellement quantifiée et n'apparaît qu'en position positive, tout comme nous l'avons déjà vu pour les variables de type ML et les variables de rangées, elle dénotera *un ensemble vide de présence*, c'est-à-dire *l'absence* de l'élément de rangée qu'elle annote.

Nommons τ le type de l'argument de `f`. L'idée est donc de forcer τ à s'unifier avec $\mathtt{int}[0:\mathtt{Pre};\, \varphi]$ afin d'exposer *explicitement* dans φ l'ensemble des valeurs possibles de τ qui sont différentes de 0.

Pour synthétiser le type du résultat de `f`, nous pouvons alors prendre $\tau' = \mathtt{int}[0:\pi;\, \varphi]$ pour une annotation de présence π nous satisfaisant. En particulier, si cette annotation reste non contrainte dans la suite de la dérivation de typage, alors nous pouvons la considérer comme étant égale à une variable de présence fraîche δ, reflétant le fait que 0 ne fait pas partie des valeurs possibles de τ'.

Ainsi la variable de rangée ρ_0 de notre exemple initial restera-t'elle bien toujours précédée du même ensemble d'éléments de rangée, dont seul le drapeau de présence aura changé. Nous obtenons alors le schéma de types suivant pour notre fonction `f` :

$$f : \forall \rho_0, \rho_1, \delta.\, \mathtt{int}[0:\mathtt{Pre};\, 1:\mathtt{Pre};\, \rho_0] \xrightarrow{\rho_1} \mathtt{int}[0:\delta;\, 1:\mathtt{Pre};\, \rho_0]$$

L'algèbre de types

Il est maintenant temps de donner la structure finale de notre algèbre de types et de rangées :

Expressions de types :

τ	$::= \alpha$	variable de type
	$\mid \mathtt{int}[\varphi]$	type de valeurs entières
	$\mid \mathtt{exn}[\varphi]$	type d'exceptions
	$\mid \tau_1 \xrightarrow{\varphi} \tau_2$	type fonctionnel

Rangées :

φ	$::= \rho$	variable de rangée
	$\mid \top$	tous les éléments possibles
	$\mid \epsilon;\ \varphi$	l'élément ϵ plus ce qui est dans φ

Éléments de rangée :

ϵ	$::= i : \pi$	constante entière
	$\mid C : \pi$	exception constante
	$\mid D(\tau)$	exception paramétrée

Annotations de présence :

π	$::= \mathtt{Pre}$	l'élément est présent
	$\mid \delta$	variable de présence

Schémas de types :

$\sigma \ ::= \forall \alpha_i, \rho_j, \delta_k . \tau$

Nous remarquerons que les drapeaux de présence que nous avons introduits précédemment n'étiquettent que les éléments de rangée constants. En effet, il n'est pas nécessaire de faire de même pour les éléments paramétrés, en partant du principe que si l'argument d'un tel élément est absent, alors l'élément lui-même, dans sa globalité, l'est (on peut ainsi dire que les éléments paramétrés sont « stricts en terme d'absence »).

Théorie équationnelle sur les rangées

Nous récapitulons maintenant la théorie équationnelle nécessaire à nos rangées, dont les équations sont modifiées afin de s'adapter à la nouvelle forme des éléments composant ces rangées :

$$\epsilon_1;\ \epsilon_2;\ \varphi \quad = \quad \epsilon_2;\ \epsilon_1;\ \varphi \tag{4.1}$$
$$i:\mathtt{Pre};\ \top \quad = \quad \top \tag{4.2}$$

Rappelons que cette théorie équationnelle exprime que l'ordre des éléments d'une rangée n'a pas d'importance (équation 4.1) et que $\top$ est élément absorbant (équation 4.2).

Règles de bonne formation des rangées

Afin de raisonner formellement sur les rangées vérifiant des invariants comme ceux que nous avons énoncés précédemment, nous définissons maintenant formellement les règles qui garantissent la bonne formation de nos rangées.

Dans la continuité de [47, 39], nous utilisons des *sortes* pour assurer les invariants énoncés ci-dessus. Nos sortes, κ, sont composées d'un marqueur (soit `INT` soit `EXN`) et d'un ensemble de constantes entières ou de constructeurs :

$\kappa \ ::= \mathtt{INT}(\{i_1, \ldots, i_n\}) \mid \mathtt{EXN}(\{C_1, \ldots, C_p, D_1, \ldots, D_q\})$

Les constantes et les constructeurs apparaissant dans la partie « ensemble » d'une sorte représentent les constantes et les constructeurs *qui ne doivent pas apparaître* dans les rangées de cette sorte (parce qu'ils apparaissent déjà parmi les éléments concaténés avant ces rangées).

$$\vdash \rho :: K(\rho)\ (1) \qquad \vdash \top :: \mathtt{INT}(S)\ (2) \qquad \frac{i \notin S \qquad \vdash \varphi :: \mathtt{INT}(S \cup \{i\})}{\vdash (i : \pi;\, \varphi) :: \mathtt{INT}(S)}\ (3)$$

$$\frac{C \notin S \qquad \vdash \varphi :: \mathtt{EXN}(S \cup \{C\})}{\vdash (C : \pi;\, \varphi) :: \mathtt{EXN}(S)}\ (4) \qquad \frac{D \notin S \qquad \vdash \varphi :: \mathtt{EXN}(S \cup \{D\}) \qquad \vdash \tau \text{ bf}}{\vdash (D(\tau);\, \varphi) :: \mathtt{EXN}(S)}\ (5)$$

$$\vdash \alpha \text{ bf}\ (6) \qquad \frac{\vdash \varphi :: \mathtt{INT}(\emptyset)}{\vdash \mathtt{int}[\varphi] \text{ bf}}\ (7) \qquad \frac{\vdash \varphi :: \mathtt{EXN}(\emptyset)}{\vdash \mathtt{exn}[\varphi] \text{ bf}}\ (8)$$

$$\frac{\vdash \tau_1 \text{ bf} \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset) \qquad \vdash \tau_2 \text{ bf}}{\vdash \tau_1 \xrightarrow{\varphi} \tau_2 \text{ bf}}\ (9)$$

FIG. 4.1 – Règles de sortage

Nous considérons donnée une fonction globale K assignant des sortes aux variables de rangées, et telle que pour chaque sorte κ, il existe une infinité de variables de cette sorte (c'est-à-dire $K^{-1}(\kappa)$ est infini). Les règles de *sortage* (ou bonne formation) sont données dans la figure ci-dessous (4.1). Elles définissent deux jugements $\vdash \varphi :: \kappa$ (la rangée φ est de sorte κ) et $\vdash \tau$ bf (le type τ est bien formé).

4.1.9 Les gestionnaires d'exceptions : soustraction

Nous savons désormais comment approximer des ensembles de valeurs au travers du type donné à une expression. Nous savons également comment représenter les ensembles d'exceptions pouvant s'échapper d'une expression ou se trouvant en attente dans une fonction (par l'intermédiaire de son effet latent).

ML permet de lever des exceptions par l'intermédiaire de la primitive `raise`, mais il permet également de les rattraper grâce à la construction `try` $expr_1$ `with x` $\rightarrow$ $expr_2$ que nous avons appelée un *gestionnaire d'exceptions*. Nous avons vu que cette construction permettait « d'absorber » toute exception levée pendant l'évaluation de $expr_1$ et d'exécuter le cas échéant $expr_2$ dans laquelle la variable x était liée à l'exception ayant interrompu le calcul de $expr_1$.

Cascader un filtrage (`match ... with ...`) en aval d'un tel gestionnaire permet alors de rattraper sélectivement certaines exceptions en relançant celles que l'on ne désire explicitement pas gérer. Ainsi, dans l'exemple suivant :

```
try raise C with
  x →
    match x with
      | C → 1
      | y → raise y
```

nous souhaiterions être capables de nous rendre compte qu'aucune exception ne peut s'échapper lors de l'évaluation de cette expression. L'effet de l'expression `raise C` est représenté par la rangée $(C : \mathtt{Pre};\, \rho)$. Ainsi, la variable x est-elle liée au type $\mathtt{exn}[C : \mathtt{Pre};\, \rho]$. Le premier motif du filtrage permet justement d'intercepter les exceptions de la forme C. Nous voudrions donc refléter le fait que l'ensemble des valeurs d'exceptions pouvant parvenir jusqu'au second motif du filtrage ne peuvent plus contenir le constructeur C. Ceci revient à refléter le fait que la seconde alternative du filtrage n'est sélectionnée que lorsque la première alternative ne filtre pas la valeur lui étant soumise.

L'idée est que, contrairement à ce qui se passe en ML, le type de y n'est pas exactement le même que celui de x. Nous voulons associer à y un type `exn` dans lequel le constructeur

C a été **soustrait** de la rangée étiquetant ce type. Pour ce faire, nous allons donc mettre en place une opération de **soustraction** permettant d'effectuer cette différenciation des types.

Nous avons déjà entr'aperçu cette notion, certes à demi-mots, lorsque nous avons exposé la nécessité de rajouter des drapeaux de présence aux éléments de rangée. En effet, le but était déjà de refléter l'absence d'une valeur au sein d'une rangée.

Nous ne donnons pas immédiatement la définition de cette opération, car il nous semble plus judicieux de la présenter en même temps que les règles de typage associées à notre système de types, règles que nous allons étudier en détail dans la section suivante. Toutefois, l'explication fournie lors de l'introduction des drapeaux permet d'ores et déjà de se douter du mécanisme que nous utiliserons pour effectuer la soustraction d'un élément de rangée d'une rangée : « l'introduction de variables de présence »...

4.2 Les règles de typage

Nous avons désormais présenté tout le matériel nécessaire à la réalisation de notre analyse. Nous connaissons la structure de l'algèbre de types sur laquelle elle repose, et nous avons découvert la nécessité d'une nouvelle opération de soustraction.

Nous allons maintenant définir formellement les règles qui gouvernent le typage des expressions du langage. Ces règles définissent le jugement $E \vdash a : \tau/\varphi$, dans lequel E est l'environnement de typage, a le terme à typer, τ le type des valeurs en lesquelles a peut s'évaluer, et φ l'ensemble des exceptions pouvant s'échapper durant l'évaluation de a. Tous les types apparaissant dans ces formules sont supposés bien formés en accord avec les règles de sortage énoncées ci-dessus. De plus nous supposons donné l'environnement de typage initial suivant : $E_0 = \{\mathtt{raise} : \forall \alpha, \rho.\, \mathtt{exn}[\rho] \xrightarrow{\rho} \alpha\}$.

Typage des expressions :

$$\frac{\tau \leq E(x) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash x : \tau / \varphi}\ (10) \qquad \frac{\vdash \varphi' :: \mathtt{INT}(\{i\}) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash i : \mathtt{int}[i : \mathtt{Pre};\ \varphi'] / \varphi}\ (11)$$

$$\frac{\vdash \tau_1 \text{ bf} \qquad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}\ (12)$$

$$\frac{E \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \qquad E \vdash a_2 : \tau'/\varphi}{E \vdash a_1(a_2) : \tau/\varphi}\ (13)$$

$$\frac{E \vdash a_1 : \tau_1/\varphi \qquad E \oplus \{x : \mathrm{Gen}\,(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi}\ (14)$$

$$\frac{\begin{array}{c} E \vdash a_1 : \tau_1/\varphi \qquad \vdash p : \tau_1 \Rightarrow E' \qquad \vdash \tau_1 - p \rightsquigarrow \tau_2 \\ E \oplus E' \vdash a_2 : \tau/\varphi \qquad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E \vdash \mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi}\ (15)$$

$$\frac{\vdash \varphi' :: \mathtt{EXN}(\{C\}) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash C : \mathtt{exn}[C : \mathtt{Pre};\ \varphi']/\varphi}\ (16)$$

$$\frac{\tau \leq \mathrm{TypeArg}(D) \qquad E \vdash a : \tau/\varphi \qquad \vdash \varphi' :: \mathtt{EXN}(\{D\}) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash D(a) : \mathtt{exn}[D(\tau);\ \varphi']/\varphi}\ (17)$$

$$\frac{E \vdash a_1 : \tau/\varphi' \qquad E \oplus \{x : \mathtt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi}{E \vdash \mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2 : \tau/\varphi}\ (18)$$

Typage des motifs :

$$\vdash x : \tau \Rightarrow \{x : \tau\}\ (19) \qquad \vdash i : \mathtt{int}[i : \mathtt{Pre};\ \varphi] \Rightarrow \{\}\ (20)$$

$$\vdash C : \mathtt{exn}[C : \mathtt{Pre};\ \varphi] \Rightarrow \{\}\ (21) \qquad \frac{\tau \leq \mathrm{TypeArg}(D) \qquad \vdash p : \tau \Rightarrow E}{\vdash D(p) : \mathtt{exn}[D(\tau);\ \varphi] \Rightarrow E}\ (22)$$

Soustraction de motif :

$$\vdash \mathtt{int}[i : \mathtt{Pre}; \varphi] - i \rightsquigarrow \mathtt{int}[i : \pi;\ \varphi]\ (23)$$

$$\vdash \mathtt{exn}[C : \mathtt{Pre}; \varphi] - C \rightsquigarrow \mathtt{exn}[C : \pi;\ \varphi]\ (24) \qquad \frac{\vdash \tau' \text{ bf}}{\vdash \tau - x \rightsquigarrow \tau'}\ (25)$$

$$\frac{\vdash \tau - p \rightsquigarrow \tau'}{\vdash \mathtt{exn}[D(\tau);\ \varphi] - D(p) \rightsquigarrow \mathtt{exn}[D(\tau');\ \varphi]}\ (26)$$

Instanciation et généralisation :

$\tau' \leq \forall \alpha_i \rho_j \delta_k.\,\tau$ si et seulement si il existe τ_i, φ_j, π_k tel que $\tau' = \tau\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\}$ et $\vdash \tau_i$ bf et $\vdash \varphi_j :: K(\rho_j)$.

$\mathrm{Gen}\,(\tau, E, \varphi)$ est $\forall \alpha_i \rho_j \delta_k.\,\tau$ où $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$.

Les règles pour les identificateurs et la liaison par `let` (règles 10 et 14) sont tout à fait standards, excepté que nous généralisons maintenant trois sortes de variables (variables de type, variables de rangée et variables de présence). Pour les identificateurs, tout comme pour les autres expressions du langage qui ne lèvent jamais d'exceptions, (règles 10, 11, 12, 16), l'effet (la rangée décrivant l'ensemble d'exceptions levé) φ du résultat reste non contrainte et peut être choisie comme l'on désire afin de satisfaire les contraintes d'égalité dans le reste de la dérivation de typage.

Les règles pour les fonctions et l'application (règles 12 et 13) ressemblent aux règles usuelles des systèmes d'effets. Pour l'abstraction, l'effet du corps de la fonction devient l'effet latent étiquetant le type de cette fonction. Pour ce qui est de l'application $a_1(a_2)$, nous demandons à ce que le même ensemble φ d'exceptions apparaisse comme effet de a_1, comme effet latent de la fonction dénotée par a_1, et comme effet de a_2. Ceci correspond, dans notre contexte basé sur l'unification, à prendre l'union de ces trois effets.

Pour les constantes entières et les constructeurs d'exceptions (règles 11, 16 et 17), nous mémorisons la valeur effective de l'expression dans la partie approximation annotant le type `int` ou `exn`. Par exemple, le type de i doit être de la forme $\mathtt{int}[i:\mathtt{Pre};\varphi]$, forçant i à apparaître dans le type de l'expression. Dans les règles 17 et 22, nous écrivons $\mathrm{TypeArg}(D)$ pour le schéma de types de l'argument du constructeur D, c'est-à-dire que $\mathrm{TypeArg}(D) = \forall\rho.\, \mathtt{int}[\rho]$ pour une exception D paramétrée par une valeur entière.

Pour un gestionnaire d'exceptions $\mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2$ (règle 18), l'effet φ_1 de a_1 est injecté dans le type $\mathtt{exn}[\varphi_1]$ supposé pour x dans a_2.

Nous arrivons maintenant à la règle la plus intéressante de notre système de types : la règle 15 pour la construction `match ... with ...` Cette règle étant cruciale pour la précision de notre analyse d'exceptions, elle mérite que nous nous y attardons plus longuement. Cette règle est chargée de refléter le comportement séquentiel et exclusif des filtrages.

Comme nous l'avons succinctement présenté dans la section précédente, lorsque nous typons $\mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3$, nous voulons refléter le fait que la seconde alternative $(x \to a_3)$ est sélectionnée seulement lorsque la première alternative $(p \to a_2)$ ne filtre pas la valeur de a_1. En d'autres termes, le type des valeurs pouvant se propager jusqu'à x dans le second cas de filtrage n'est pas le type de la valeur filtrée, mais le type de a_1 duquel nous avons exclu toutes les valeurs ayant été filtrées par le motif p dans la première alternative.

Ce comportement s'avère de première importance pour notre analyse puisque les gestionnaires sélectifs d'exceptions (c'est-à-dire ceux ne rattrapant que certaines exceptions) sont construits à partir d'une expression `try ... with ...` couplée avec une série d'expressions `match ... with ...` mises en cascade. Par exemple, en utilisant la syntaxe décrite pour notre langage, nous pouvons écrire le gestionnaire d'exceptions sélectif suivant :

```
try ... with x →
  match x with
    | C → 0
    | y →
       match y with
         | D 42 → 1
         | z → raise z
```

dont le rôle est d'intercepter, parmi les exceptions levées lors de l'évaluation de l'expression dénotée par les points de suspension, celles et uniquement celles de la forme C et $D(42)$. Les autres exceptions sont alors ignorées et relevées par la dernière expression `raise`. Notons qu'il est possible d'écrire un tel programme plus concisément en Caml sous la forme `try ... with C → 0 | D 42 → 1`, les exceptions non filtrées étant automatiquement propagées dans l'arbre d'appel de l'évaluation.

Pour rendre compte de ce comportement, les règles 23–26 définissent le prédicat de *soustraction de motif* $\vdash \tau - p \leadsto \tau'$, signifiant que τ' est un type correct pour les valeurs de type τ *qui ne sont pas filtrées* par le motif p. Pour un motif « variable » (c'est-à-dire de la forme d'un identificateur, non pas que le motif soit changeant) $p = x$ (règle 25), toutes les valeurs sont filtrées par ce motif. Ainsi, il est correct de considérer que n'importe quel τ' convient pour type des valeurs non filtrées (puisqu'en réalité il n'y en a plus...). Pour un motif $p = i$ représentant un entier (règle 23), nous supprimons dans τ tous les éléments de rangée égaux à i, mettant ainsi en évidence que cette constante a été absorbée par le motif. Les règles pour les motifs d'exceptions (règles 24 et 26) sont similaires à celles que nous venons d'exposer. Si l'exception possède un argument, nous soustrayons récursivement dans le type de l'argument du constructeur de cette exception.

4.3 Quelques exemples de types

Nous allons donner ici quelques exemples de types associés à des expressions ML afin de renforcer l'intuition sous-jacente à cette algèbre et à ce système.

À la lumière des règles de typage et de soustraction, nous pouvons maintenant revoir l'exemple d'introduction de la soustraction (cf. section 4.1.9) qui était :

```
try raise C with
  x →
    match x with
      | C → 1
      | y → raise y
```

L'effet de `raise(C)` est donc $\mathtt{C} : \mathtt{Pre}; \rho$. Ainsi, le type de `x` est $\mathtt{exn}[\mathtt{C} : \mathtt{Pre}; \rho]$. Après soustraction du motif `C`, nous obtenons le type $\mathtt{exn}[\mathtt{C} : \delta; \rho]$ pour `y`. Nous en déduisons que l'effet de l'expression `match` (et donc également de `try`) devient $\mathtt{C} : \delta; \rho$ et son type $\mathtt{int}[1 : \mathtt{Pre}; \rho']$. Maintenant, nous avons trois variables généralisables n'apparaissant qu'en position positive : ρ, δ et ρ' et dénotant ainsi des « ensembles vides ». En particulier δ exprime bien qu'en réalité C n'est pas présent dans la rangée, et nous en déduisons à nouveau qu'aucune exception ne peut s'échapper de l'évaluation de cette expression.

Étendons maintenant l'exemple ci-dessus dans l'idée de la fonction `failwith` que nous avons présentée à propos des exigences de notre analyse (cf. section 1.3).

```
let failwith n = raise(D(n)) in
try failwith(42)
with x → match x with D(42) → 0 | y → raise y
```

Nous obtenons alors les typages intermédiaires suivants :

$$\begin{array}{rcl}
\mathtt{failwith} & : & \forall\alpha,\rho_1,\rho_2.\, \mathtt{int}[\rho_1] \xrightarrow{D(\mathtt{int}[\rho_1]);\, \rho_2} \alpha \\
\mathtt{x} & : & \mathtt{exn}[D(\mathtt{int}[42 : \mathtt{Pre};\, \rho_3]);\, \rho_4] \\
\mathtt{y} & : & \mathtt{exn}[D(\mathtt{int}[42 : \delta;\, \rho_3]);\, \rho_4]
\end{array}$$

Ainsi, nous concluons comme précédemment qu'aucune exception ne peut s'échapper de cette expression.

Afin d'exposer un cas représentatif de fonction d'ordre supérieur, considérons la composition de fonctions :

```
let compose f g x = f(g(x)) in
compose (fun y → 0) (fun z → raise(C)) 1
```

Le schéma de types de `compose` est donc $\forall\alpha,\beta,\gamma,\rho,\rho_1,\rho_2.(\alpha \xrightarrow{\rho} \beta) \xrightarrow{\rho_1} (\gamma \xrightarrow{\rho} \alpha) \xrightarrow{\rho_2} \gamma \xrightarrow{\rho} \beta$. Les trois occurrences de ρ expriment l'union des effets de `f` et `g`. L'application de `compose` ci-dessus a ainsi l'effet $\mathtt{C} : \mathtt{Pre}; \rho_3$.

En ce qui concerne les exceptions considérées comme valeurs de première classe, le premier exemple de la section 1.2 devient :

```
let test exn =
  try raise(exn) with
    | x →
      match x with
        | C → 1
        | y → raise(y)
in test(C)
```

Le schéma de types alors associé à `test` est $\forall\rho,\rho',\delta.\, \mathtt{exn}[\mathtt{C}:\mathtt{Pre};\ \rho] \xrightarrow{\mathtt{C}:\delta;\ \rho} \mathtt{int}[1:\mathtt{Pre};\ \rho']$, exprimant le fait que la fonction lève n'importe quelle exception reçue en argument à l'exception de `C`. L'application `test(C)` a ainsi le type $\mathtt{int}[1:\mathtt{Pre};\ \rho_1]$ et l'effet $\mathtt{C}:\delta_2;\ \rho_2$. Ainsi, aucune exception ne peut effectivement s'échapper. L'application `test C'` où `C'` est une autre exception distincte de `C` aurait l'effet $\mathtt{C}:\delta_3;\ \mathtt{C}':\mathtt{Pre};\ \rho_3$, montrant ainsi que `C'` peut échapper.

Nous avons vu que notre algèbre de types était établie modulo une théorie équationnelle comportant deux équations. Nous allons voir ici deux exemples simples de typages exposant l'utilité de ces équations.

En effet, dans les règles de typages énoncées précédemment, la première de ces équations (4.1) est utilisée mais implicitement ce qui peut dérouter le lecteur en lui donnant l'impression qu'elle ne sert pas. Cette utilisation implicite provient de ce que dans toutes les règles, nous explicitons les rangées en mettant directement en tête le constructeur qui nous intéresse. Or, dans une dérivation de typage, ce n'est pas toujours forcément le cas, ce qui n'empêche pas pour autant notre système de typer l'expression en question. Plus précisément, considérons qu'au cours d'une dérivation de typage, nous ayions affaire à un séquent de la forme :

$$\frac{}{E \vdash \mathtt{3} : \mathtt{int}[1:\mathtt{Pre};\ 2:\mathtt{Pre};\ 3:\mathtt{Pre};\ \rho]}$$

En l'absence de notre première équation (4.1), il nous est impossible d'appliquer la règle de typage 11 puisque le constructeur `3` ne se trouve pas en tête de la rangée étiquetant le type `int`. Réorganisons cette rangée sous la forme $[3:\mathtt{Pre};\ 1:\mathtt{Pre};\ 2:\mathtt{Pre};\ \rho]$ en appliquant deux fois l'équation 4.1 et la règle 11 devient maintenant utilisable.

Ainsi, pragmatiquement, l'expression :

```
let sign x = if x < 0 then -1 else if x > 0 then 1 else 0 in
  sign 42
```

aura le type $\mathtt{int}[13:\mathtt{Pre};\ 4:\delta;\ 0:\mathtt{Pre};\ 1:\mathtt{Pre};\ -1:\mathtt{Pre}]$, même si le type plus précis (principal) suivant : $\mathtt{int}[0:\mathtt{Pre};\ 1:\mathtt{Pre};\ -1:\mathtt{Pre}]$ serait préférable dans le cadre où ces types sont inférés par le système !

La seconde équation de notre théorie porte sur le comportement de $\top$. Celui-ci n'apparaît en effet pas dans les règles du fait que nous avons formalisé un système sans primitives externes au langage (par exemple les opérateurs arithmétiques). Il est toutefois aisé de donner un tel exemple sous la forme de l'expression suivante : `3 + 4` qui aura simplement le type $\mathtt{int}[\top]$. En effet, puisqu'il nous est impossible d'analyser le corps de `+`, nous ne pouvons que considérer que son schéma de types est $\forall\rho,\rho',\rho'',\rho'''.\, \mathtt{int}[\rho] \xrightarrow{\rho'} \mathtt{int}[\rho''] \xrightarrow{\rho'''} \mathtt{int}[\top]$.

Jusqu'à présent, nous n'avons montré que des exemples où notre système donnait des types relativement précis, apportant une « bonne information » quant aux exceptions pouvant s'échapper des expressions. Voyons maintenant un cas contraire. Nous nous appuyons sur la fonction `test` donnée ci-dessus, dont nous rappelons le texte au passage.

```
exception C

let test exn =
  try raise(exn) with
```

```
  | C → 1
  | y → raise(y)

let test2 exn2 =
 let foo = test exn2 in
 raise exn2
```

Comme précédemment, la fonction `test` sera considérée comme ne laissant pas s'échapper l'exception C ainsi que l'illustre son schéma de types : $\forall \rho, \rho', \delta.\, \mathtt{exn}[C:\mathtt{Pre};\ \rho] \xrightarrow{C:\delta;\ \rho} \mathtt{int}[1:\mathtt{Pre};\ \rho']$. Examinons maintenant le typage de la fonction `test2`. Nous savons que les arguments des fonctions ne peuvent pas être généralisés. Ainsi le type de `exn2` ne l'est pas et a simplement la forme : $\mathtt{exn}[\rho_2]$. Typons l'application `test exn2`. Nous commençons par prendre une instance du schéma de types de `test` c'est-à-dire un type de la forme : $\mathtt{exn}[C:\mathtt{Pre};\ \rho_0] \xrightarrow{C:\delta_0;\ \rho_0} \mathtt{int}[1:\mathtt{Pre};\ \rho_1]$. Nous unifions alors le type de `exn2` (à savoir $\mathtt{exn}[\rho_2]$) avec la partie gauche cette instance (à savoir $\mathtt{exn}[C:\mathtt{Pre};\ \rho_0]$). Ceci provoque l'apparition de $C:\mathtt{Pre}$ dans la rangée annotant le type de `exn2`. Puisque le reste de la fonction `test2` consiste à lever l'exception `exn2` son effet latent contient donc l'exception C, disant ainsi à tort que notre fonction `test2` peut laisser s'échapper l'exception `E`.

Finalement, nous examinons ici un exemple (pathologique, certes) qui est mal typé en ML, mais bien typé dans notre système grâce à notre typage fin des filtrages par l'opération de soustraction de motifs.

```
match 1 with x → x | e → raise e
```

Puisque le premier cas du filtrage est « totalement absorbant », la règle 15 nous autorise à assigner le type $\mathtt{exn}[\rho']$ (dans lequel ρ' est une variable fraîche) à l'identificateur `e` lié par le second cas, même si la valeur filtrée est en fait un entier.

Ainsi l'expression est bien typée et nous déduisons qu'elle a pour type $\mathtt{int}[1:\mathtt{Pre};\ \rho]$ et ne lève aucune exception (son effet est ρ' pour tout ρ').

4.4 Conservativité du typage par rapport à ML

Notre analyse doit être appliquée à des programmes ML déjà existant. Il est donc naturel de se demander si tout programme typable avec le système de ML conventionnel sera typable avec notre système étendu par les effets et les rangées. Autrement dit, y-a-t'il *conservativité* du typage entre ML et notre système ?

4.4.1 Une première interrogation

Dans le cadre où nous avons décrit notre analyse, c'est-à-dire en présence de termes de types finis (non cycliques), la réponse est **non**. En effet, considérons l'exemple suivant :

```
exception E of int → int

function f →
 if ... then f
 else (function x → raise (E f))
```

En ML, cette fonction est de type `(int → int) → int → int`. Examinons quel est son type dans notre système. Plus particulièrement intéressons-nous au typage des deux branches de la conditionnelle. En ML conventionnel, supposons que `f` (première branche de la conditionnelle) soit d'un type $\tau_1 \to \tau_2$, la seconde branche sera de type $\alpha \to \beta$ et unifier ces deux types ne posera aucune difficulté. Dans notre système, `f` aura plutôt un type de la forme $\tau_1 \xrightarrow{\rho} \tau_2$. La seconde branche, quant à elle, sera de type $\tau_1 \xrightarrow{E(\tau_1 \xrightarrow{\rho} \tau_2);\ \rho'} \tau_2$.

Unifier ces deux types dans le cadre de termes finis est un problème qui n'a pas de solution puisque cela revient à rendre égaux deux termes dont l'un est un sous-terme de l'autre (pratiquement, dans l'algorithme d'unification que nous verrons ultérieurement, ceci correspond à un test d'occurrence, ou *occur-check* en Anglais, qui échoue).

La solution à ce problème est d'utiliser des termes de types cycliques (termes rationnels). Dans un tel cadre, l'algorithme d'unification entre termes n'effectue plus de test d'occurrence et autorise la génération de types comme $\mu\,\beta.\,\alpha \to \beta$ signifiant que β représente le type infini $\alpha \to \alpha \to \alpha \to \ldots$.

L'implantation réelle de notre analyseur utilise effectivement une telle algèbre de types cycliques afin de contourner ce problème. Ainsi, notre fonction va avoir le type :

$$(\mu\,\alpha.\,\mathtt{int}[\rho_1] \xrightarrow{E(\alpha);\,\rho_2} \mathtt{int}[\rho_3]) \xrightarrow{\rho_0} (\mu\,\alpha.\,\mathtt{int}[\rho_1] \xrightarrow{E(\alpha);\,\rho_2} \mathtt{int}[\rho_3])$$

ce que l'on peut écrire, de manière peut-être plus intuitive sous la forme :

$$\tau_0 \xrightarrow{\rho_0} \tau_0 \quad \text{où} \quad \tau_0 = \mathtt{int}[\rho_1] \xrightarrow{E(\tau_0);\,\rho_2} \mathtt{int}[\rho_3]$$

Toutefois, afin de conserver une présentation simple de notre système et de ses propriétés, dans la suite de cette thèse, nous allons nous restreindre au cadre des termes de types finis. Les preuves que nous effectuerons peuvent s'étendre au cas des termes de types rationnels, au prix d'une augmentation de leur complexité non négligeable que nous préférons éviter car ceci apporte peu à la compréhension du mécanisme même qui gouverne notre analyse.

L'extension de notre algèbre par les types cycliques implique principalement le remplacement de l'unification entre termes par une unification entre graphes au sein de l'algorithme d'inférence, mais ne pose pas de réels problèmes techniques.

4.4.2 Un problème ouvert

Nous avons vu que l'utilisation d'une algèbre de types cycliques nous permettait de régler notre premier problème de conservativité du typage par rapport à ML. Reste désormais un dernier point délicat : celui de la *généralisation*.

En effet, la question qui nous préoccupe maintenant est de savoir si l'ajout d'effets dans notre algèbre, ne pourrait pas empêcher des généralisations de types qui l'auraient été avec le système de types de ML habituel.

Imaginons qu'en ML nous ayons une dérivation de typage de la forme :

$$E \oplus (x : \mathtt{int} \to \mathtt{int}) \vdash e : \alpha$$

La variable de type α n'apparaissant pas dans l'environnement, elle est généralisable. Est-il maintenant possible que la dérivation équivalente dans notre système de types soit de la forme :

$$E \oplus (x : \mathtt{int} \xrightarrow{\ldots\alpha\ldots} \mathtt{int}) \vdash e : \alpha$$

auquel cas, la variable α apparaissant désormais dans l'environnement, celle-ci *n'est plus* généralisable !

Nous laissons ce problème en suspend pour le moment et nous en re-parlerons ultérieurement dans le chapitre 10.

Conclusion Nous avons donc abouti à une algèbre de types permettant de refléter les deux aspects importants de notre analyse, à savoir d'une part sa raison d'être (qui est de détecter les exceptions levées et non rattrapées) et d'autre part sa capacité à approximer le flot des valeurs pouvant se propager dans le programme.

Ainsi, notre algèbre présente deux nouvelles caractéristiques majeures par rapport à l'algèbre de base de ML : une notion *d'effet* pour chaque expression (c'est-à-dire l'ensemble des exceptions qu'elle peut provoquer) en plus de son type, ainsi qu'une notion *d'approximation de valeurs* pour chaque type.

Chapitre 5

Correction de l'analyse

Nous avons maintenant terminé la présentation du système de types sur lequel repose notre analyse d'exceptions. Nous devons maintenant établir la correction de cette analyse, c'est-à-dire montrer que toutes les exceptions non rattrapées pouvant survenir durant l'exécution d'un programme sont effectivement prédites par notre analyse.

5.1 Prélude

Nous allons maintenant nous concentrer sur les preuves de correction de notre système de types proprement dit. Les preuves rédigées dans cette section seront volontairement très détaillées, suivant pour chaque cas le même schéma de raisonnement. Ceci peut certes rendre leur lecture monotone, mais nous pensons que c'est également un moyen de bien les comprendre et de les exposer de façon régulière et rigoureuse. Aussi privilégions-nous sciemment l'aspect systématique à l'aspect rhétorique.

5.2 Propriétés des jugements de typage

Lemme 5.1 (Le typage des motifs rend des environnements bien formés) *Soit un motif p et un type τ bien formé. Alors $\vdash p : \tau \Rightarrow E$ implique que E est bien formé (c'est-à-dire que pour tout $x \in \mathrm{Dom}\,(E)$, $E(x)$ est de la forme $\forall\vec{\alpha},\vec{\rho},\vec{\delta}.\tau$ avec $\vdash \tau$ bf).*

Preuve : La preuve s'effectue très simplement par cas sur les jugements de typage des motifs.

Cas p est x.

Par hypothèse τ est bien formé. L'environnement de typage E résultat étant constitué de $\{x : \tau\}$ (cf. règle 19), il est par conséquent bien formé.

Cas p est i ou C.

Dans ces deux cas, l'environnement résultat E est vide et donc trivialement bien formé.

Cas p est $D(p)$.

Ce dernier cas se traite simplement en appliquant l'hypothèse de récurrence sur partie droite de la règle 22, ce qui nous fournit le résultat attendu.

□

Lemme 5.2 (La soustraction rend des types bien formés) *Soit un motif p et un type τ bien formé. Soit τ' tel que $\vdash \tau - p \rightsquigarrow \tau'$, alors $\vdash \tau'$ bf.*

Preuve : La preuve est menée par induction sur les règles de soustraction entre type et motif.

Cas $\mathtt{int}[i : \mathtt{Pre}; \varphi] - i$ (règle 23).

Dans ce cas le type du résultat de la soustraction est $\mathtt{int}[i : \pi; \varphi]$. Puisque φ était par hypothèse bien sortée, elle le demeure dans le type résultat (elle est toujours précédée des mêmes éléments de rangée), établissant ainsi la bonne formation de ce dernier.

Cas $\mathtt{exn}[C : \mathtt{Pre}; \varphi] - C$ (règle 24).

On se convaincra aisément que ce cas est similaire au précédent.

Cas $\mathtt{exn}[D(\tau); \varphi] - D(p)$ (règle 26).

Le type $\mathtt{exn}[D(\tau); \varphi]$ étant par hypothèse bien formé, son sous-terme τ l'est également. Ainsi nous pouvons appliquer l'hypothèse de récurrence sur la prémisse de la règle de typage et en déduire que τ' est bien formé. Le type résultat de la soustraction étant $\mathtt{exn}[D(\tau'); \varphi]$, il est ainsi également bien formé (puisque la sorte de φ n'a pas changé et que τ' est bien formé).

□

Lemme 5.3 (La généralisation rend des schémas de types bien formés) *Soit un type τ, un environnement de typage E et un effet φ tous bien formés. Alors* $\mathrm{Gen}\,(\tau, E, \varphi)$ *est un schéma de types bien formé (c'est-à-dire de la forme $\forall \alpha_i \rho_j \delta_k . \tau$ où τ* bf*).*

Preuve : Ceci est trivialement vrai par la définition de la généralisation Gen . □

Lemme 5.4 (Une instance de schéma bien formé est un type bien formé) *Soit un schéma de types σ bien formé et un type τ. Si $\tau \leq \sigma$ alors* $\vdash \tau$ bf.

Preuve : Par définition de l'instanciation celle-ci remplace, dans le corps de σ, des variables de types généralisées par des types τ_i bien formés, ainsi que des variables de rangée par des rangées des mêmes sortes ($\varphi_j :: K(\rho_j)$). Ainsi τ est-il lui-même bien formé. □

Lemme 5.5 (Les règles de typage des expressions sont bien sortées) *Soit E un environnement de typage bien formé (c'est-à-dire tel que, pour tout $x \in \mathrm{Dom}\,(E)$, $E(x)$ est de la forme $\forall \vec{\alpha}, \vec{\rho}, \vec{\delta} . \tau$ avec $\vdash \tau$* bf*). Supposons de plus que* $\mathrm{TypeArg}(D_i)$ *est bien formé pour tout constructeur d'exceptions D_i paramétré. Alors, $E \vdash a : \tau/\varphi$ implique $\vdash \tau$* bf *et* $\vdash \varphi :: \mathtt{EXN}(\emptyset)$.

Ce lemme permet de s'assurer que le mécanisme de sortage que nous avons mis en place afin de garantir les invariants nécessaires à nos rangées sera conservé tout au long d'une dérivation de typage.

Preuve : La preuve s'effectue par induction sur la dérivation de $E \vdash a : \tau/\varphi$ en utilisant les quatre lemmes précédents (5.1, 5.2, 5.3 et 5.4).

Cas a est x.

Hypothèse :

$$E \vdash x : \tau/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \tau \text{ bf} \tag{5.2}$$

$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 10 appliquée à 5.1, nous déduisons que :

$$\tau \leq E(x) \tag{5.4}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.5}$$

Par hypothèse, E ne contient que des schémas de types bien formés. τ étant une instance d'un schéma de E (déduction 5.4), il est également bien formé par application du lemme 5.4.

De plus, φ étant par hypothèse bien formé, de sorte $\mathtt{EXN}(\emptyset)$ (déduction 5.5), ceci établit la seconde partie de la preuve de ce cas.

Ainsi toutes les conditions nécessaires à la vérification de notre lemme dans le cas présent sont vérifiées.

Cas a est i.
Hypothèse :

$$E \vdash i : \mathtt{int}[i : \mathtt{Pre};\ \varphi']/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \mathtt{int}[i : \mathtt{Pre};\ \varphi'] \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 11 appliquée à 5.1, nous déduisons que :

$$\vdash \varphi' :: \mathtt{INT}(\{i\}) \tag{5.4}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.5}$$

Le type $\mathtt{int}[i : \mathtt{Pre};\ \varphi']$ de l'expression est bien formé si sa rangée est de sorte $\mathtt{INT}(\emptyset)$. Par hypothèse, $\varphi' :: \mathtt{INT}(\{i\}$ (déduction 5.4), ce que nous pouvons écrire $\varphi' :: \mathtt{INT}(S \cup \{i\})$ avec $S = \emptyset$. Puisque $i \notin \emptyset$, nous pouvons alors appliquer la règle de sortage 3 suivante :

$$\frac{i \notin S \qquad \vdash \varphi :: \mathtt{INT}(S \cup \{i\})}{\vdash (i : \pi;\ \varphi) :: \mathtt{INT}(S)}$$

et en déduire que $[i : \mathtt{Pre};\ \varphi'] :: \mathtt{INT}(\emptyset)$ ce qui établit la correction de notre rangée étiquetant le type `int` résultat et donc celle de ce type lui-même comme attendu en 5.2.

Pour ce qui est de l'effet φ de l'expression, par hypothèse 5.5 celui-ci satisfait nos exigences (but 5.3) puisqu'il est trivialement choisi de sorte $\mathtt{EXN}(\emptyset)$.

Ainsi, cette dérivation de typage conserve-t'elle la bonne formation des types.

Cas a est $\lambda x.a_1$.
Hypothèse :

$$E \vdash \lambda x.a_1 : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \tau_1 \xrightarrow{\varphi'} \tau_2 \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 12 appliquée à 5.1, nous déduisons que :

$$\vdash \tau_1 \text{ bf} \tag{5.4}$$
$$E \oplus \{x : \tau_1\} \vdash a_1 : \tau_2/\varphi' \tag{5.5}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.6}$$

Par l'hypothèse 5.4, τ_1 est bien formé. Puisque E est également bien formé par hypothèse et a_1 est un terme strictement plus petit que l'expression complète, nous pouvons appliquer l'hypothèse de récurrence et déduire que τ_2 est bien formé et que $\varphi' :: \mathtt{EXN}(\emptyset)$. Ainsi le type du résultat, $\tau_1 \overset{\varphi'}{\to} \tau_2$ est bien formé (but 5.2).

De plus, par l'hypothèse 5.6, $\varphi :: \mathtt{EXN}(\emptyset)$ (but 5.3). Ainsi toutes les conditions sont réunies pour clore la preuve de notre lemme dans ce cas.

Cas a est $a_1(a_2)$.

Hypothèse :

$$E \vdash a_1(a_2) : \tau/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \tau \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 13 appliquée à 5.1, nous déduisons que :

$$E \vdash a_1 : (\tau' \overset{\varphi}{\to} \tau)/\varphi \tag{5.4}$$
$$E \vdash a_2 : \tau'/\varphi \tag{5.5}$$

Les sous-expressions a_1 et a_2 étant strictement plus petites que $a_1(a_2)$ et E ne contenant par hypothèse que des types bien formés, nous pouvons appliquer deux fois l'hypothèse de récurrence (à 5.4 et 5.5) et déduire que $(\tau' \overset{\varphi}{\to} \tau)$ est bien formé (donc que τ l'est) et que $\varphi :: \mathtt{EXN}(\emptyset)$. Ceci clos ainsi la preuve de ce cas.

Cas a est $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$.

Hypothèse :

$$E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \tau \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 14 appliquée à 5.1, nous déduisons que :

$$E \vdash a_1 : \tau_1/\varphi \tag{5.4}$$
$$E \oplus \{x : \mathrm{Gen}\,(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi \tag{5.5}$$

Par application de l'hypothèse de récurrence sur 5.4, nous déduisons que τ_1 est bien formé et que $\varphi :: \mathtt{EXN}(\emptyset)$. Nous voulons maintenant montrer que l'environnement utilisé dans 5.5 est lui aussi bien formé afin de pouvoir appliquer l'hypothèse de récurrence. Cet environnement, construit à partir de E (par hypothèse bien formé) et de la généralisation de τ_1 (également bien formé comme montré juste précédemment) est bien formé comme nous l'énonce l'application du lemme 5.3.

Ainsi, nous pouvons appliquer l'hypothèse de récurrence sur 5.5 (puisque a_2 est strictement plus petite que l'expression `let` toute entière) et déduire que τ est bien formé (but 5.2), et une fois encore que $\varphi :: \mathtt{EXN}(\emptyset)$ (but 5.3). Ceci est donc le résultat attendu.

Cas a est $\mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3$.

Hypothèse :

$$E \vdash \mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \tau \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 15 appliquée à 5.1, nous déduisons que :

$$E \vdash a_1 : \tau_1/\varphi \tag{5.4}$$
$$\vdash p : \tau_1 \Rightarrow E' \tag{5.5}$$
$$\vdash \tau_1 - p \rightsquigarrow \tau_2 \tag{5.6}$$
$$E \oplus E' \vdash a_2 : \tau/\varphi \tag{5.7}$$
$$E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \tag{5.8}$$

Nous pouvons d'ores et déjà appliquer l'hypothèse de récurrence sur 5.4 et en déduire que τ_1 est bien formé et que $\varphi :: \mathtt{EXN}(\emptyset)$.

Nous savons (lemme 5.1) que le typage des motifs retourne des environnements bien formés. Donc $E \oplus E'$ est bien formé et nous pouvons appliquer la récurrence sur la déduction 5.7 pour en déduire que τ est bien formé (et à nouveau que $\varphi :: \mathtt{EXN}(\emptyset)$).

Nous savons également (lemme 5.2) que la soustraction entre un type bien formé et un motif rend un type bien formé. Puisque τ est bien formé, l'environnement de typage $E \oplus \{x : \tau_2\}$ l'est également. Nous appliquons donc la récurrence sur 5.8 et finalement déduisons à nouveau que τ est toujours bien formé (but 5.2) et que le sortage de φ est toujours tel que $\varphi :: \mathtt{EXN}(\emptyset)$ (but 5.3).

Toutes les conditions requises étant réunies, nous concluons ce cas de notre preuve.

Cas a est C. La règle utilisée dans ce cas de la preuve est la règle 16. On se convaincra aisément que ce cas est totalement similaire au cas de l'expression « constante entière » démontré ci-dessus. Nous nous autorisons à ne pas le traiter explicitement afin de ne pas rallonger inutilement cette démonstration.

Cas a est $D(a_1)$.

Hypothèse :

$$E \vdash D(a) : \mathtt{exn}[D(\tau);\ \varphi']/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \mathtt{exn}[D(\tau);\ \varphi'] \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 17 appliquée à 5.1, nous déduisons que :

$$\tau \leq \text{TypeArg}(D) \tag{5.4}$$
$$E \vdash a_1 : \tau/\varphi \tag{5.5}$$
$$\vdash \varphi' :: \mathtt{EXN}(\{D\}) \tag{5.6}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.7}$$

Ce cas ne pose pas de problème particulier et s'ensuit directement de l'application de l'hypothèse de récurrence sur a_1, sachant que par hypothèses TypeArg(D) est bien formé, $\vdash \varphi' :: \mathtt{EXN}(\{D\})$ (déduction 5.6) et $\vdash \varphi :: \mathtt{EXN}(\emptyset)$ (déduction 5.7).

Cas a est $\mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2$.

Hypothèse :

$$E \vdash \mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2 : \tau/\varphi \tag{5.1}$$

On veut prouver que :

$$\vdash \tau \text{ bf} \tag{5.2}$$
$$\vdash \varphi :: \texttt{EXN}(\emptyset) \tag{5.3}$$

Par existence d'une dérivation 18 appliquée à 5.1, nous déduisons que :

$$E \vdash a_1 : \tau/\varphi' \tag{5.4}$$
$$E \oplus \{x : \texttt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi \tag{5.5}$$

Nous appliquons la récurrence sur 5.4 et déduisons que τ bf et φ' :: $\texttt{EXN}(\emptyset)$. Ainsi le type $\texttt{exn}[\varphi']$ est-il bien formé, d'où l'environnement de typage $E \oplus \{x : \texttt{exn}[\varphi']\}$ aussi. Nous pouvons alors appliquer à nouveau l'hypothèse de récurrence sur 5.5 et finalement déduire le résultat attendu : $\vdash \tau$ bf et $\vdash \varphi :: \texttt{EXN}(\emptyset)$.

□

Dans la définition qui suit, nous notons $\text{Rng}(\theta)$ pour le codomaine de la substitution θ, c'est-à-dire, $\bigcup\{FV(\theta(v)) \mid v \in \text{Dom}(\theta)\}$.

Définition 5.1 (Variable hors de portée d'une substitution) *Nous dirons qu'une variable de type v est hors de portée d'une substitution θ si $v \notin \text{Dom}(v) \cup \text{Rng}(v)$.*

En d'autres termes, $\theta(v) = v$, et pour toutes les autres variables $v' \neq v$, v n'est pas libre dans $\theta(v')$.

Nous pouvons encore exprimer cette propriété en disant qu'une variable v est dite hors de portée d'une substitution θ si elle n'appartient au domaine de θ, et n'est pas libre dans un des types de l'image de θ.

Nous énonçons maintenant trois lemmes ayant trait aux variables hors de portée de substitutions et faisant partie des résultats connus sur les substitutions. Les deux premiers de ces lemmes sont principalement techniques (et leur démonstration n'est pas donnée ici afin de ne pas rallonger ce chapitre outre mesure) et servent dans la preuve du troisième d'entre eux.

Lemme 5.6 () *Si α est hors de portée de θ et $\alpha \in FV(\theta(\beta))$ alors $\alpha = \beta$.*

Preuve : $\beta \notin \text{Dom}(\theta)$ car dans le cas contraire on aurait $\alpha \in \text{Rng}(\theta)$, c'est-à-dire α non hors de portée de θ.

Donc $\theta(\beta) = \beta$ et $\alpha \in FV(\theta(\beta)) = FV(\beta) = \{\beta\}$ entraîne $\alpha = \beta$. □

Lemme 5.7 () *Si $\alpha \in FV(\varphi(\tau))$ alors il existe une variable γ telle que $\alpha \in FV(\varphi(\gamma))$ et $\gamma \in FV(\tau))$.*

Preuve : Le preuve de ce lemme est simplement effectuée par réccurrence structurelle sur τ. □

Lemme 5.8 (Variable hors de portée d'une composition de substitutions) *Soient une variable α et deux substitutions θ et ψ. Si α est hors de portée de θ et hors de portée de ψ, alors α est hors de portée de $\theta \circ \psi$.*

Preuve : Soit α hors de portée de θ et de ψ. Soit une variable β.

Supposons que $\alpha \in FV(\theta \circ \psi(\beta))$ et $\beta \in \mathrm{Dom}\,(\theta \circ \psi)$ (c'est-à-dire que α n'est pas hors de portée de $\theta \circ \psi$).

Par le lemme 5.7, on sait qu'il existe une variable γ telle que :

$$\alpha \in FV(\theta(\gamma)) \tag{5.1}$$
$$\gamma \in FV(\psi(\beta)) \tag{5.2}$$

Par le lemme 5.6, de la déduction 5.1, nous déduisons alors que $\alpha = \gamma$. Donc, la déduction 5.2 devient :

$$\alpha \in FV(\psi(\beta)) \tag{5.3}$$

Appliquons à nouveau le lemme 5.6, mais sur la déduction 5.3. Il nous vient alors que $\alpha = \beta$, ce qui entre en contradiction avec notre hypothèse initiale disant que $\beta \in \mathrm{Dom}\,(\theta \circ \psi)$.
□

Lemme 5.9 (Commutation entre instanciation et substitution) *Si $\tau \leq \sigma$, alors $\theta(\tau) \leq \theta(\sigma)$ pour toute substitution θ.*

Preuve : Notons le schéma de types σ sous la forme : $\forall \alpha_i \rho_j \delta_k . \tau'$. Après renommage si nécessaire, on peut supposer que les α_i, ρ_j et δ_k sont hors de portée de θ.
Hypothèse :

$$\tau \leq \sigma \tag{5.1}$$

c'est-à-dire, par définition de l'instanciation :

$$\text{il existe une substitution } \psi \text{ des } \alpha_i, \rho_j, \delta_k \text{ telle que } \tau = \psi(\tau') \tag{5.2}$$

On veut prouver que :

$$\theta(\tau) \leq \theta(\sigma) \tag{5.3}$$

c'est-à-dire, par définition de l'instanciation :

$$\text{qu'il existe une substitution } \psi' \text{ telle que } \theta(\tau) = \psi'(\theta(\tau')) \tag{5.4}$$

On définit une substitution ψ' de domaine $\alpha_i, \rho_j, \delta_k$ par $\psi'(\alpha_i) = \theta(\psi(\alpha_i)$, et $\psi'(\rho_j) = \theta(\psi(\rho_j)$, et $\psi'(\delta_k) = \theta(\psi(\delta_k)$. Nous avons alors :

$$\begin{array}{rclcll}
\psi'(\theta(\alpha_i)) & = & \psi'(\alpha_i) & = & \theta(\psi(\alpha_i)) & \text{pour tout } i \\
\psi'(\theta(\alpha'_i)) & = & \theta(\alpha'_i) & = & \theta(\psi(\alpha'_i)) & \text{pour tout } \alpha'_i \text{ différent des } \alpha_i \\
\hline
\psi'(\theta(\rho_j)) & = & \psi'(\rho_j) & = & \theta(\psi(\rho_j)) & \text{pour tout } j \\
\psi'(\theta(\rho'_j)) & = & \theta(\rho'_j) & = & \theta(\psi(\rho'_j)) & \text{pour tout } \rho'_j \text{ différent des } \rho_j \\
\hline
\psi'(\theta(\delta_k)) & = & \psi'(\delta_k) & = & \theta(\psi(\delta_k)) & \text{pour tout } k \\
\psi'(\theta(\delta'_k)) & = & \theta(\delta'_k) & = & \theta(\psi(\delta'_k)) & \text{pour tout } \delta'_k \text{ différent des } \delta_k
\end{array}$$

D'où $\psi'(\theta(\tau')) = \theta(\psi(\tau')) = \theta(\tau)$, comme attendu en 5.4.
□

Lemme 5.10 (Commutation entre généralisation et substitution) *Si toutes les variables présentes dans $FV(\tau) \setminus (FV(E) \cup FV(\varphi))$ sont hors de portée de la substitution θ, alors $\mathrm{Gen}\,(\theta(\tau), \theta(E), \theta(\varphi)) = \theta(\mathrm{Gen}\,(\tau, E, \varphi))$.*

Preuve : Il est facile de voir qu'une variable α hors de portée de θ est libre dans un type τ si et seulement si elle est libre dans $\theta(\tau)$. Ainsi, $FV(\theta(\tau)) \setminus (FV(\theta(E)) \cup FV(\theta(\varphi))) = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$, et le résultat attendu s'ensuit. □

Lemme 5.11 (Préservation des sortes sur des rangées et des types arbitraires)

Soit θ une substitution préservant les sortes, c'est-à-dire telle que pour toute variable de rangée ρ, on ait $\theta(\rho) :: K(\rho)$, alors $\vdash \varphi :: \kappa$, implique $\vdash \theta(\varphi) :: \kappa$. Et $\vdash \tau$ bf implique $\vdash \theta(\tau)$ bf.

Intuitivement ce lemme montre qu'une substitution qui préserve les sortes des variables de rangées préserve également la sorte des rangées contenant ces variables. Autrement dit, on étend la notion de « préservation de sorte » des variables de rangée aux rangées arbitraires.

Preuve : La preuve est simplement effectuée par récurrence sur la structure des rangées.

Cas $\varphi = \rho$.
Puisque φ est égale à ρ, nous avons $\kappa = K(\rho)$. Puisque θ préserve les sortes, $\theta(\rho) :: K(\rho)$. Donc $\theta(\rho) :: \kappa$, c'est-à-dire $\theta(\varphi) :: \kappa$.

Cas $\varphi = \top$.
La rangée $\top$ ne comportant aucune variable de rangée, le théorème est trivialement vérifié.

Cas $\varphi = \epsilon; \varphi'$.
Nous devons raisonner par cas sur la forme de l'élément de rangée ϵ.

- Cas ϵ est $C : \pi$. Nous avons alors $\kappa = \texttt{EXN}(S)$.
  Hypothèse :

$$\vdash (C : \pi;\ \varphi') :: \texttt{EXN}(S) \tag{5.1}$$

  On veut prouver que :

$$\vdash \theta(C : \pi;\ \varphi') :: \texttt{EXN}(S) \tag{5.2}$$

  Pour y parvenir, nous voudrions utiliser une règle 4 de la forme :

$$\frac{\cdots \qquad \cdots}{\vdash \theta(C : \pi;\ \varphi') :: \texttt{EXN}(S)}$$

  Nous devons donc en établir les prémisses, c'est-à-dire :

$$\vdash \theta(C) \notin S \tag{5.3}$$
$$\vdash \theta(\varphi') :: \texttt{EXN}(S \cup \{C\}) \tag{5.4}$$

  De 5.1, en utilisant une règle 4, nous déduisons que :

$$C \notin S \tag{5.5}$$
$$\vdash \varphi' :: \texttt{EXN}(S \cup \{C\}) \tag{5.6}$$

  La substitution θ n'ayant pas d'effet sur C, nous déduisons notre but 5.3 trivialement de 5.5. La rangée φ' étant plus petite que φ, nous pouvons appliquer l'hypothèse de récurrence sur 5.6 et nous déduisons directement notre but 5.4. Ainsi, munis de ces deux prémisses, nous pouvons reconstituer une dérivation 4 comme attendu et conclure le sous-cas présent.

- Cas ϵ est $i:\pi$. Nous avons alors $\kappa = \mathtt{INT}(S)$. On se convaincra aisément que ce cas est traité similairement au précédent.
- Cas ϵ est $D(\tau)$. Nous avons alors $\kappa = \mathtt{EXN}(S)$.
  Hypothèse :

$$\vdash (D(\tau);\, \varphi') :: \mathtt{EXN}(S) \tag{5.1}$$

  On veut prouver que :

$$\vdash \theta(D(\tau);\, \varphi') :: \mathtt{EXN}(S) \tag{5.2}$$

  Pour y parvenir, nous voudrions utiliser une règle (5) de la forme :

$$\frac{\ldots \quad \ldots \quad \ldots}{\vdash \theta(D(\tau);\, \varphi') :: \mathtt{EXN}(S)}$$

  Nous devons donc en établir les prémisses, c'est-à-dire :

$$\vdash \theta(D) \notin S \tag{5.3}$$
$$\vdash \theta(\varphi') :: \mathtt{EXN}(S \cup \{D\}) \tag{5.4}$$
$$\vdash \theta(\tau)\ \mathrm{bf} \tag{5.5}$$

  De (5.1), en utilisant une règle (5), nous déduisons que :

$$D \notin S \tag{5.6}$$
$$\vdash \varphi' :: \mathtt{EXN}(S \cup \{D\}) \tag{5.7}$$
$$\vdash \tau\ \mathrm{bf} \tag{5.8}$$

  La substitution θ n'ayant pas d'effet sur D, nous déduisons notre but 5.3 trivialement de 5.6. La rangée φ' étant plus petite que φ, nous pouvons appliquer l'hypothèse de récurrence sur 5.7 et nous déduisons directement notre but 5.4. De même, en appliquant l'hypothèse de récurrence sur 5.8, nous déduisons notre but 5.5. Ainsi, munis de ces trois prémisses, nous pouvons reconstituer une dérivation comme attendu et clore ca cas de notre démonstration.

Note : afin d'appliquer l'hypothèse de récurrence sur les types, comme nous l'avons fait dans le dernier sous-cas ci-dessus, nous devons montrer que $\vdash \tau$ bf implique $\vdash \theta(\tau)$ bf. Cette preuve est traitée par cas sur les règles de sortage des rangées (6–9) et ne présente aucune difficulté. L'application de l'hypothèse de récurrence, aussi bien au niveau des types que des rangées suffit à montrer chacun de ces cas, et nous nous autorisons à ne pas donner cette démonstration *in extenso* afin de ne pas rallonger inutilement la présente démonstration.

□

Lemme 5.12 (Le typage est stable par substitution) *Soit θ une substitution préservant les sortes.*

1. *Si $\vdash p : \tau \Rightarrow E$ alors $\vdash p : \theta(\tau) \Rightarrow \theta(E)$.*
2. *Si $\vdash \tau - p \rightsquigarrow \tau'$ alors $\vdash \theta(\tau) - p \rightsquigarrow \theta(\tau')$.*
3. *Si $E \vdash a : \tau/\varphi$ alors $\theta(E) \vdash a : \theta(\tau)/\theta(\varphi)$.*

Preuve : Commençons par démontrer que le typage des motifs est stable par substitution (point 1). La preuve s'effectue par induction sur la forme des motifs.

Cas p est x.

La règle 19 nous dit que $\vdash x : \tau \Rightarrow \{x : \tau\}$, donc trivialement $\vdash x : \theta(\tau) \Rightarrow \{x : \theta(\tau)\}$ ce qui est le résultat attendu.

Cas p est i ou C.
On se convaincra que ces deux cas sont similaires. Les règles 20 et 21 nous montrent que l'environnement de typage résultat E est vide, et donc satisfait trivialement notre proposition

Cas p est $D(p)$.
La dérivation (22) impliquée dans ce cas est de la forme :

$$\frac{\tau \leq \text{TypeArg}(D) \qquad \vdash p : \tau \Rightarrow E}{\vdash D(p) : \texttt{exn}[D(\tau);\, \varphi] \Rightarrow E}$$

Ce cas se traite donc par utilisation de l'hypothèse de récurence sur le sous-motif p (celui-ci étant strictement plus petit que $D(p)$) ce qui nous fournit directement le résultat escompté. □

Preuve : Prouvons maintenant le point 2 de notre lemme, à savoir que la soustraction est stable par substitution. La preuve s'effectue par cas sur la définition de la soustraction.

Cas $\texttt{int}[i : \texttt{Pre};\, \varphi] - i$.
Dans le cas présent, à partir d'une règle 23, nous avons que :

$$\vdash \texttt{int}[i : \texttt{Pre}; \varphi] - i \rightsquigarrow \texttt{int}[i : \pi;\, \varphi]$$

Nous voulons montrer que :

$$\vdash \theta(\texttt{int}[i : \texttt{Pre};\, \varphi]) - i \rightsquigarrow \theta(\texttt{int}[i : \pi;\, \varphi]) \tag{5.9}$$

Nous avons que :

$$\theta(\texttt{int}[i : \texttt{Pre};\, \varphi]) = \texttt{int}[i : \texttt{Pre};\, \theta(\varphi)] \tag{5.10}$$

Appliquons alors la règle 23 sur le membre droit de 5.10. Il vient que :

$$\texttt{int}[i : \texttt{Pre};\, \theta(\varphi)] - i \rightsquigarrow \texttt{int}[i : \pi';\, \theta(\varphi)] \tag{5.11}$$

Choisissons $\pi' = \theta(\pi)$. Nous avons donc que :

$$\texttt{int}[i : \texttt{Pre};\, \theta(\varphi)] - i \rightsquigarrow \texttt{int}[i : \theta(\pi);\, \theta(\varphi)]$$

c'est-à-dire que :

$$\texttt{int}[i : \texttt{Pre};\, \theta(\varphi)] - i \rightsquigarrow \theta(\texttt{int}[i : \pi;\, \varphi])$$

ce qui revient, par l'égalité (5.10), à :

$$\theta(\texttt{int}[i : \texttt{Pre};\, \varphi]) - i \rightsquigarrow \theta(\texttt{int}[i : \pi;\, \varphi])$$

comme attendu en 5.9.

Cas $\texttt{exn}[C : \texttt{Pre};\, \varphi] - C$ (règle 24).
On se convaincra aisément que ce cas est traité de la même manière que le précédent et ne sera pas donné formellement afin de ne pas rallonger inutilement cette démonstration.

Cas $\tau - x$.
Dans le cas présent, à partir d'une règle 25, nous avons que : $\vdash \tau - x \rightsquigarrow \tau'$ avec $\vdash \tau'$ bf. Nous voulons montrer que : $\vdash \theta(\tau) - x \rightsquigarrow \theta(\tau')$.

Par la règle 25, nous avons : $\vdash \theta(\tau) - x \leadsto \tau''$ et $\vdash \tau''$ bf pour un certain τ'' non contraint. Choisissons alors $\tau'' = \theta(\tau')$. La substitution θ préservant les sortes et τ' étant bien formé, par le lemme 5.11, nous déduisons que $\theta(\tau')$, donc τ'' est bien formé. De plus nous avons bien alors que $\vdash \theta(\tau) - x \leadsto \theta(\tau')$ comme attendu.

Cas $\mathtt{exn}[D(\tau);\ \varphi] - D(p)$.

Dans le cas présent, à partir d'une règle 26, nous avons que :

$$\vdash \mathtt{exn}[D(\tau);\ \varphi] - D(p) \leadsto \mathtt{exn}[D(\tau');\ \varphi] \tag{5.1}$$

$$\vdash \tau - p \leadsto \tau' \tag{5.2}$$

Nous voulons montrer que :

$$\vdash \theta(\mathtt{exn}[D(\tau);\ \varphi]) - D(p) \leadsto \theta(\mathtt{exn}[D(\tau');\ \varphi])$$

Par application de l'hypothèse de récurence à 5.2, il nous vient que :

$$\vdash \theta(\tau) - p \leadsto \theta(\tau') \tag{5.3}$$

Appliquons une règle 26 sur 5.3, et il vient que :

$$\vdash \mathtt{exn}[D(\theta(\tau));\ \theta(\varphi)] - D(p) \leadsto \mathtt{exn}[D(\theta(\tau'));\ \theta(\varphi)]$$

c'est-à-dire :

$$\vdash \theta(\mathtt{exn}[D(\tau);\ \varphi]) - D(p) \leadsto \theta(\mathtt{exn}[D(\tau');\ \varphi])$$

ce qui est le résultat attendu.

□

Preuve : Pour terminer, nous prouver le dernier point de notre lemme (point 3), c'est-à-dire que les règles de typage des expressions sont stables par substitution. On raisonnera par récurrence structurelle sur la forme des expressions.

Cas a est x.

On a comme hypothèse que $E \vdash x : \tau$. On veut prouver que $\theta(E) \vdash x : \theta(\tau)$.

Pour obtenir notre hypothèse de départ, nous avons forcément utilisé une dérivation 10 de la forme :

$$\frac{\tau \leq E(x) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash x : \tau / \varphi}$$

dont les prémisses font ainsi également partie de nos hypothèses.

Pour prouver notre lemme dans le cas présent, on souhaite reconstituer une telle dérivation, mais portant pour conclusion :

$$\frac{\ldots}{\theta(E) \vdash x : \theta(\tau)/\theta(\varphi)}$$

Il faut reconstituer les prémisses d'une telle règle, autrement dit prouver que $\theta(\tau) \leq \theta(E(x))$ et que $\theta(\varphi) :: \mathtt{EXN}(\emptyset)$.

Puisque nous savons que $\tau \leq E(x)$, la première exigence découle directement du lemme de commutation entre instanciation et substitution démontré précédemment (cf. lemme 5.9).

La seconde est directement obtenue en appliquant le lemme de préservation des sortes sur une rangée (lemme 5.11) à la prémisse de droite de notre règle 10 de départ.

Ainsi nos deux prémisses étant réunies, nous pouvons bien reconstituer une dérivation telle que nous le voulions, ce qui nous démontre bien notre but dans ce cas.

Cas a est i.

Nous sommes dans le cas où $E \vdash i : \mathtt{int}[i : \mathtt{Pre};\ \varphi']/\varphi$. Nous voulons prouver que :

$$\theta(E) \vdash i : \theta(\mathtt{int}[i : \mathtt{Pre};\ \varphi'])/\theta(\varphi)$$

Pour aboutir au cas présent, nous avons forcément utilisé une dérivation 11 de la forme :

$$\frac{\vdash \varphi' :: \mathtt{INT}(\{i\}) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash i : \mathtt{int}[i : \mathtt{Pre};\ \varphi']/\varphi}$$

dont les prémisses sont donc également nos hypothèses de départ. Nous voulons donc reconstituer une telle dérivation, terminant par :

$$\frac{\dots}{\theta(E) \vdash i : \theta(\mathtt{int}[i : \mathtt{Pre};\ \varphi'])/\theta(\varphi)}$$

Il nous faut donc prouver les prémisses de cette règle, autrement dit $\vdash \theta(\varphi') :: \mathtt{INT}(\{i\})$ et $\vdash \theta(\varphi) :: \mathtt{EXN}(\emptyset)$.

Celles-ci sont directement obtenues en appliquant le lemme de préservation des sortes sur une rangée (lemme 5.11) à la prémisse de droite de notre règle 11 de départ.

Cas a est $\lambda x.a$

Nous savons que $E \vdash \lambda x.a : \tau_1 \xrightarrow{\varphi'} \tau_2/\varphi$. Nous voulons prouver que

$$\theta(E) \vdash \lambda x.a : \theta(\tau_1 \xrightarrow{\varphi'} \tau_2)/\theta(\varphi)$$

Dans le cas présent, nous avons forcément fait appel à une dérivation 12 de la forme :

$$\frac{\vdash \tau_1 \text{ bf} \qquad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash \lambda x.a : \tau_1 \xrightarrow{\varphi'} \tau_2/\varphi}$$

Nous voulons donc reconstituer cette dérivation avec notre but comme conclusion. Ainsi nous devons en prouver les prémisses, c'est-à-dire $\vdash \theta(\tau_1)$ bf, $\theta(E \oplus \{x : \tau_1\}) \vdash a : \theta(\tau_2)/\theta(\varphi')$ et enfin $\vdash \theta(\varphi) :: \mathtt{EXN}(\emptyset)$.

La substitution θ préservant les sortes et τ_1 étant bien formé, nous déduisons que $\vdash \theta(\tau_1)$ bf comme attendu en première prémisse.

D'autre part nous pouvons appliquer l'hypothèse de récurrence sur la seconde prémisse, avec l'environnement $E \oplus \{x : \tau_1\}$ ce qui nous fournit la seconde prémisse que nous attendions.

Pour obtenir la dernière prémisse, φ étant par hypothèse de sorte $\mathtt{EXN}(\emptyset)$, et la substitution θ préservant les sortes, il en est de même pour $\theta(\varphi)$.

Munis de ces trois prémisses, nous pouvons ainsi reconstituer une dérivation 12 comme attendu et clore ce cas de la preuve.

Cas a est $a_1(a_2)$.

Nous savons que $E \vdash a_1(a_2) : \tau/\varphi$. Nous voulons prouver que

$$\theta(E) \vdash a_1(a_2) : \theta(\tau)/\theta(\varphi)$$

Nous avons forcément utilisé une dérivation 13 de la forme :

$$\frac{E \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \qquad E \vdash a_2 : \tau'/\varphi}{E \vdash a_1(a_2) : \tau/\varphi}$$

Nous voulons reconstituer une dérivation similaire, avec comme conclusion :

$$\frac{\dots}{\theta(E) \vdash a_1(a_2) : \theta(\tau)/\theta(\varphi)}$$

Nous devons donc en démontrer les prémisses, c'est-à-dire $\theta(E) \vdash a_1 : \theta(\tau' \xrightarrow{\varphi} \tau)/\theta(\varphi)$ et $\theta(E) \vdash a_2 : \theta(\tau')/\theta(\varphi)$.

Les expressions a_1 et a_2 étant plus petites que l'expression originale, nous pouvons appliquer deux fois l'hypothèse de récurrence (une fois à droite et une fois à gauche) pour en déduire exactement les deux prémisses que nous attendions. Nous pouvons de ce fait reconstituer notre dérivation et clore ce cas.

Cas a est `let` $x = a_1$ `in` a_2.

Notre hypothèse de départ est : $E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau/\varphi$.

Nous voulons prouver que :

$$\theta(E) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \theta(\tau)/\theta(\varphi)$$

De notre hypothèse de départ, nous déduisons que nous avons forcément utilisé une dérivation 14 de la forme :

$$\frac{E \vdash a_1 : \tau_1/\varphi \qquad E \oplus \{x : \mathrm{Gen}\,(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau/\varphi}$$

Nous voulons donc reconstituer une dérivation de cette forme dont la conclusion est notre but, c'est-à-dire :

$$\frac{\dots \qquad \dots}{\theta(E) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \theta(\tau)/\theta(\varphi)}$$

Établissons donc les prémisses de cette règle. Par définition de la généralisation, nous savons que $\mathrm{Gen}\,(\tau_1, E, \varphi)$ est égal à $\forall \alpha_i, \rho_j, \delta_k.\, \tau_1$ avec $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau_1) \backslash (FV(E) \cup FV(\varphi))$.

Soient α'_i, ρ'_j et δ'_k des variables hors de portée de θ et non libres (n'apparaissant pas) dans E ni φ. On note θ' la substitution $\{\alpha_i \leftarrow \alpha'_i;\ \rho_j \leftarrow \rho'_j;\ \delta_k \leftarrow \delta'_k\}$ et $\theta'' = \theta \circ \theta'$.

Appliquons l'hypothèse de récurence sur la prémisse de gauche avec θ''. Nous déduisons que :

$$\theta''(E) \vdash a_1 : \theta''(\tau_1)/\theta''(\varphi) \tag{5.1}$$

Appliquons maintenant l'hypothèse de récurence avec θ sur la prémisse de droite dans laquelle nous renommons les variables α_i, ρ_j et δ_k de τ_1 par la substitution θ'. Nous en déduisons que :

$$\theta(E \oplus \{x : \mathrm{Gen}\,(\theta'(\tau_1), E, \varphi)\}) \vdash a_2 : \theta(\tau)/\theta(\varphi) \tag{5.2}$$

c'est-à-dire que :

$$\theta(E) \oplus \theta(\{x : \mathrm{Gen}\,(\theta'(\tau_1), E, \varphi)\}) \vdash a_2 : \theta(\tau)/\theta(\varphi) \tag{5.3}$$

Les variables de $\theta'(\tau_1)$ étant hors de portée de θ, nous pouvons appliquer lemme de commutation entre généralisation et substitution sur 5.3. Donc nous avons que :

$$\theta(E) \oplus \{x : \mathrm{Gen}\,(\theta \circ \theta'(\tau_1), \theta(E), \theta(\varphi))\} \vdash a_2 : \theta(\tau)/\theta(\varphi) \tag{5.4}$$

c'est-à-dire :

$$\theta(E) \oplus \{x : \mathrm{Gen}\,(\theta''(\tau_1), \theta(E), \theta(\varphi))\} \vdash a_2 : \theta(\tau)/\theta(\varphi) \tag{5.5}$$

Or nous avons que $\theta''(E) = \theta(E)$ et $\theta''(\varphi) = \theta(\varphi)$ puisque les variables α_i, ρ_j et δ_k affectées par θ' ne sont pas libres dans E ni φ et que par choix, $\theta'' = \theta' \circ \theta$. Donc de 5.1 nous déduisons que nous avons :

$$\theta(E) \vdash a_1 : \theta''(\tau_1)/\theta(\varphi) \tag{5.6}$$

Munis des deux prémisses 5.5 et 5.6, nous pouvons appliquer une règle 14, qui nous permet de déduire que :

$$\theta(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \theta(\tau)/\theta(\varphi) \tag{5.7}$$

comme attendu au début.

Cas a est $\texttt{match } a_1 \texttt{ with } p \to a_2 \mid x \to a_3$.

Nous savons que :

$$E \vdash \texttt{match } a_1 \texttt{ with } p \to a_2 \mid x \to a_3 : \tau/\varphi$$

et nous voulons prouver que :

$$\theta(E) \vdash \texttt{match } a_1 \texttt{ with } p \to a_2 \mid x \to a_3 : \theta(\tau)/\theta(\varphi)$$

Nous avons forcément utilisé une dérivation 15 de la forme :

$$\frac{\begin{array}{c} E \vdash a_1 : \tau_1/\varphi \qquad \vdash p : \tau_1 \Rightarrow E' \qquad \vdash \tau_1 - p \rightsquigarrow \tau_2 \\ E \oplus E' \vdash a_2 : \tau/\varphi \qquad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E \vdash \texttt{match } a_1 \texttt{ with } p \to a_2 \mid x \to a_3 : \tau/\varphi}$$

Nous voulons donc construire une dérivation identique terminant par notre but, c'est-à-dire :

$$\frac{\cdots}{E \vdash \texttt{match } a_1 \texttt{ with } p \to a_2 \mid x \to a_3 : \tau/\varphi}$$

Comme dans les cas précédents, nous essayons d'établir les prémisses d'une telle règle. L'application de l'hypothèse de récurrence à $E \vdash a_1 : \tau_1/\varphi$ nous permet de déduire notre première prémisse : $\theta(E) \vdash a_1 : \theta(\tau_1)/\theta(\varphi)$.

Nous avons montré en tant que premier point du présent lemme, que le typage des motifs était stable par substitution. Ainsi ce résultat appliqué à la seconde prémisse de la règle de départ nous fournit directement notre seconde prémisse : $\vdash p : \theta(\tau_1) \Rightarrow \theta(E')$ pour la règle que nous voulons reconstituer.

De plus, en tant que second point de ce lemme, nous avons également prouvé que la soustraction de motif était stable par substitution. Appliqué à la troisième prémisse, (c'est-à-dire $\vdash \tau_1 - p \rightsquigarrow \tau_2$) nous en déduisons que $\vdash \theta(\tau_1) - p \rightsquigarrow \theta(\tau_2)$.

Les deux dernières prémisses dont nous avons besoin sont simplement déduites par application de l'hypothèse de récurrence sur les deux dernières prémisses de la règle de départ.

Munis de toutes nos prémisses, nous reconstituons bien la dérivation attendue, terminant ainsi ce cas de la preuve.

Cas a est C.

Ce cas est similaire au cas a est i démontré précédemment.

Cas a est $D(a)$.

Nous savons que $E \vdash D(a) : \texttt{exn}[D(\tau);\ \varphi']/\varphi$ et nous voulons démontrer que :

$$\theta(E) \vdash D(a) : \theta(\texttt{exn}[D(\tau);\ \varphi'])/\theta(\varphi)$$

De notre hypothèse départ, nous déduisons que nous avons forcément utilisé une règle 17 de la forme :

$$\frac{\tau \leq \text{TypeArg}(D) \qquad E \vdash a : \tau/\varphi \qquad \vdash \varphi' :: \texttt{EXN}(\{D\}) \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E \vdash D(a) : \texttt{exn}[D(\tau);\ \varphi']/\varphi}$$

Nous voulons donc reconstituer une telle dérivation terminant par notre but, à savoir :

$$\frac{\dots}{\theta(E) \vdash D(a) : \theta(\mathtt{exn}[D(\tau);\ \varphi'])/\theta(\varphi)}$$

Appliquons l'hypothèse de récurence sur $E \vdash a : \tau/\varphi$. Nous déduisons que $\theta(E) \vdash a : \theta(\tau)/\theta(\varphi)$ ce qui est la première prémisse que nous recherchons.

Les deux autres prémisses de droite sont directement obtenues en appliquant le lemme de préservation des sortes sur une rangée (lemme 5.11) aux deux prémisses respectives de droite de notre règle 17 de départ.

Munis de toutes nos prémisses, nous reconstituons bien la dérivation attendue, closant ainsi ce cas de la preuve.

Cas a est $\mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2$.

Nous sommes dans le cas où $E \vdash \mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2 : \tau/\varphi$ et nous voulons prouver que :

$$\theta(E) \vdash \mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2 : \theta(\tau)/\theta(\varphi)$$

De notre hypothèse de départ, nous déduisons que nous avons forcément une dérivation 18 de la forme :

$$\frac{E \vdash a_1 : \tau/\varphi' \qquad E \oplus \{x : \mathtt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi}{E \vdash \mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2 : \tau/\varphi}$$

Recherchons les prémisses nécessaires à la construction d'une telle règle terminant par notre but.

La première d'entre elles est simplement obtenue par application de l'hypothèse de récurence à la prémisse de gauche de notre règle 18 de départ et nous permet donc de déduire que $\theta(E) \vdash a_1 : \theta(\tau)/\theta(\varphi')$.

La seconde est de la forme $\theta(E) \oplus \{x : \theta(\mathtt{exn}[\varphi'])\} \vdash a_2 : \ \theta(\tau)/\theta(\varphi)$. Appliquons à nouveau l'hypothèse de récurence, mais cette fois sur la prémisse de droite de notre règle de départ. Il nous vient alors que $\theta(E \oplus \{x : \mathtt{exn}[\varphi']\}) \vdash a_2 : \theta(\tau)/\theta(\varphi)$, ce qui n'est pas encore tout à fait ce que nous attendions. Il nous reste donc à montrer que $\theta(E \oplus \{x : \mathtt{exn}[\varphi']\}) = \theta(E) \oplus \{x : \theta(\mathtt{exn}[\varphi'])\}$ ce qui est structurellement trivial.

Ainsi nos deux prémisses sont réunies et nous reconstituons notre dérivation attendue. □

Définition 5.2 (Relation de généralité entre schémas de types) *Nous dirons qu'un schéma de types σ est* plus général *qu'un schéma σ', et écrirons $\sigma \geq \sigma'$, si toutes les instances de σ' sont aussi des instances de σ.*

Lemme 5.13 (Le typage est stable sous des hypothèses renforcées) *Soit* $\mathrm{Dom}\,(E_1) \subseteq \mathrm{Dom}\,(E_2)$ *et $E_2(x) \geq E_1(x)$ pour tout $x \in \mathrm{Dom}\,(E_1)$. Alors, $E_1 \vdash a : \tau/\varphi$ implique $E_2 \vdash a : \tau/\varphi$.*

Preuve : Celle-ci est réalisée par induction structurelle sur la forme des expressions a.

Cas a est x.

Nous partons donc de l'hypothèse que $E_1 \vdash x : \tau/\varphi$ dont nous déduisons que nous avons également (règle 10) que $\tau \leq E_1(x)$ et $\vdash \varphi :: \mathtt{EXN}(\emptyset)$.

Nous voulons prouver que $E_2 \vdash x : \tau/\varphi$. Il nous faut donc prouver que $\tau \leq E_2(x)$. Puisque τ est une instance de $E_1(x)$ et que par hypothèse toute instance de $E_1(x)$ est aussi une instance de $E_2(x)$, ceci est trivialement vérifié.

De plus, $\vdash \varphi :: \mathtt{EXN}(\emptyset)$ est toujours vérifié. Nous avons ainsi deux prémisses qui nous permettent, par l'intermédiaire d'une dérivation 10, de déduire notre but initial.

Cas a est i.

Nous savons que $E_1 \vdash i : \mathtt{int}[i : \mathtt{Pre};\, \varphi']/\varphi$ et donc (règle 11) que $\vdash \varphi' :: \mathtt{INT}(\{i\})$ et $\vdash \varphi :: \mathtt{EXN}(\emptyset)$.

Nous voulons prouver que $E_2 \vdash i : \mathtt{int}[i : \mathtt{Pre};\, \varphi']/\varphi$.

Ceci est trivialement vérifié puisque le typage d'une expression de constante entière ne fait pas intervenir à proprement parler d'environnement de typage.

Cas a est $\lambda x.a$.

Nous savons donc que $E_1 \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi$ et nous voulons montrer que $E_2 \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi$.

De notre hypothèse de départ, nous déduisons que nous avons forcément utilisé une dérivation 12 de la forme :

$$\frac{\vdash \tau_1 \text{ bf} \qquad E_1 \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E_1 \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}$$

Nous souhaitons reconstituer une telle dérivation, terminant par :

$$\frac{\cdots}{E_2 \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}$$

Les deux prémisses de droite et de gauche ne faisant pas intervenir d'environnement de typage restent donc vérifiées dans le cas où l'on choisit E_2 plutôt que E_1. Il nous reste donc seulement à établir que $E_2 \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi'$.

Pour tout y, si $E_2(y) \geq E_1(y)$, alors trivialement $(E_2 \oplus \{x : \tau_1\})(y) \geq (E_1 \oplus \{x : \tau_1\})(y)$ et nous pouvons alors appliquer l'hypothèse de récurrence sur a et $E_1 \oplus \{x : \tau_1\}$ pour déduire notre prémisse (c'est-à-dire que $E_2 \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi'$).

Ainsi nous retrouvons bien les trois prémisses d'une dérivation 12 basée sur E_2 qui nous fournit le résultat attendu.

Cas a est $a_1(a_2)$.

Nous savons que $E_1 \vdash a_1(a_2) : \tau/\varphi$ et nous voulons prouver que :

$$E_2 \vdash a_1(a_2) : \tau/\varphi$$

Puisque, partant de l'hypothèse de départ, nous n'avons pu qu'utiliser une dérivation 13, nous avons également que $E_1 \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi$ et $E_1 \vdash a_2 : \tau'/\varphi$.

Appliquons l'hypothèse de récurrence sur ces deux prémisses et nous en déduisons que $E_2 \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi$ et $E_2 \vdash a_2 : \tau'/\varphi$ qui forment les deux prémisses d'une règle 13 basée sur E_2. Nous pouvons alors utiliser une telle dérivation ce qui nous amène directement le but recherché, à savoir que $E_2 \vdash a_1(a_2) : \tau/\varphi$.

Cas a est $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$.

Nous savons que $E_1 \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi$. Nous voulons montrer que :

$$E_2 \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi$$

Partant de notre hypothèse de départ, nous avons forcément une dérivation 14 de la forme :

$$\frac{E_1 \vdash a_1 : \tau_1/\varphi \qquad E_1 \oplus \{x : \mathrm{Gen}\,(\tau_1, E_1, \varphi)\} \vdash a_2 : \tau/\varphi}{E_1 \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi}$$

dont les prémisses étendent maintenant nos hypothèses.

Nous voulons donc construire une dérivation similaire, se terminant par :

$$\frac{\cdots}{E_2 \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi}$$

Nous devons donc en établir les prémisses. Avant déterminer la forme de ce prémisses, envisageons le problème suivant. Nommons $\alpha_i, \rho_j, \delta_k$ les variables généralisées de τ_1. Imaginons que l'environnement E_2 comporte malencontreusement des variables libres de même nom que ces $\alpha_i, \rho_j, \delta_k$. Alors il se produira une capture de variables. L'idée est que les variables $\alpha_i, \rho_j, \delta_k$ de τ_1 étant généralisées, elles sont alpha-convertibles et nous pouvons, sans perte de généralité, les renommer. Prenons donc une substitution ψ qui renomme les ces variables généralisées de τ_1 en variables $\alpha'_i, \rho'_j, \delta'_k$ n'appartenant pas à φ et non libres dans E_2.

Établissons donc la première de ces prémisses, c'est-à-dire $E_2 \vdash a_1 : \psi(\tau_1)/\varphi$. Appliquons le lemme de stabilité du typage par substitution (partie 3 du lemme 5.12) avec ψ sur le prémisse de gauche de notre règle de départ. Il nous vient que :

$$\psi(E_1) \vdash a_1 : \psi(\tau_1)/\psi(\varphi)$$

La substitution ψ affectant uniquement les variables généralisées de τ_1, et ces variables n'étant libres ni dans E_1 ni dans φ (par définition de Gen), du résultat ci-dessus nous déduisons que :

$$E_1 \vdash a_1 : \psi(\tau_1)/\varphi$$

Appliquons maintenant l'hypothèse de réccurrence sur cette dernière déduction. Il nous vient alors que $E_2 \vdash a_1 : \psi(\tau_1)/\varphi$ ce qui est bien la première prémisses que nous attendions.

La seconde prémisse doit donc être de la forme : $E_2 \oplus \{x : \mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi)\} \vdash a_2 : \tau/\varphi$. Trivialement, puisque par hypothèse $\mathrm{Dom}\,(E_1) \subseteq \mathrm{Dom}\,(E_2)$, nous avons que $\mathrm{Dom}\,(E_1 \oplus \{x : \mathrm{Gen}\,(\tau_1, E_1, \varphi)\}) \subseteq \mathrm{Dom}\,(E_2 \oplus \{x : \mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi)\})$, donc pour pouvoir appliquer l'hypothèse de réccurrence, il ne nous reste plus qu'à montrer que $E_2 \oplus \{x : \mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi)\} \geq E_1 \oplus \{x : \mathrm{Gen}\,(\tau_1, E_1, \varphi)\}$.

Puisque nous savons déjà (par hypothèse) que $E_2 \geq E_1$, ceci revient maintenant à montrer que $\{x : \mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi)\} \geq \{x : \mathrm{Gen}\,(\tau_1, E_1, \varphi)\}$, donc que $\mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi) \geq \mathrm{Gen}\,(\tau_1, E_1, \varphi)$.

- $\mathrm{Gen}\,(\tau_1, E_1, \varphi) = \forall \alpha_i, \rho_j, \delta_k. \tau_1$ où $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau_1) \setminus FV(E_1) \setminus FV(\varphi)$
- $Gen(\psi(\tau_1), E_2, \varphi) = \forall \alpha'_i, \rho'_j, \delta'_k, \alpha''_l, \rho''_m, \delta''_n. \psi(\tau_1)$, schéma de types dans lequel nous avons $\{\alpha'_i, \rho'_j, \delta'_k, \alpha''_l, \rho''_m, \delta''_n\} = FV(\psi_1(\tau_1)) \setminus FV(E_2) \setminus FV(\varphi)$. Or, par construction de ψ, ceci est égal à $\forall \alpha_i, \rho_j, \delta_k, \alpha''_l, \rho''_m, \delta''_n. \tau_1$.
  En effet, $FV(\psi(\tau_1)$ est égal à $FV(\tau_1)$ dans lequel les variables $\alpha_i, \rho_j, \delta_k$ sont remplacées par $\alpha'_i, \rho'_j, \delta'_k$. Sur l'intersection des domaines de E_1 et E_2, puisque par hypothèse E_2 est plus général que E_1, nous savons que $FV(E_2) \subseteq FV(E_1)$, donc la soustraction ensembliste $FV(\psi(\tau_1)) \setminus FV(E_2)$ « retire » moins de variables que la soustraction $FV(\tau_1) \setminus FV(E_1)$.
  Reste maintenant le problème des variables libres apparaissant dans les liaisons de E_2 et pas de E_1 (puisque par hypothèse $\mathrm{Dom}\,(E_1) \subseteq \mathrm{Dom}\,(E_2)$) qui pourraient *a priori* « être retirées » de notre ensemble de variables généralisées et donc faire en sorte que $Gen(\psi(\tau_1), E_2, \varphi)$ comporte moins de variables généralisées que $Gen(\tau_1, E_1, \varphi)$ et donc, soit moins général. Or ces variables ne peuvent pas se retrouver dans τ_1 (car par hypothèse $E_1 \vdash a_1 : \tau_1$) donc *a posteriori* non plus dans $\psi(\tau_1)$. Donc les supprimer de $FV(\psi(\tau_1))$ n'a pas d'effet, ne diminue pas le cardinal de notre ensemble de variables généralisées.

Ainsi $Gen(\psi(\tau_1), E_2, \varphi)$ comportant plus de variables généralisées que $\mathrm{Gen}\,(\tau_1, E_1, \varphi)$, il est bien plus général et nous avons bien $E_2 \oplus \{x : \mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi)\} \geq E_1 \oplus \{x : \mathrm{Gen}\,(\tau_1, E_1, \varphi)\}$ comme attendu.

Nous pouvons alors appliquer l'hypothèse de réccurrence et en déduire que $E_2 \oplus \{x : \mathrm{Gen}\,(\psi(\tau_1), E_2, \varphi)\} \vdash a_2 : \tau/\varphi$.

Ainsi, nous avons donc établi les deux prémisses attendues et pouvons désormais reconstituer une dérivation comme désiré, ce qui clos la preuve dans ce cas ci.

Cas a est $E \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3$.

Nous savons que $E_1 \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi$ et nous voulons prouver que :

$$E_2 \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi$$

Nous avons donc une dérivation 15 de la forme :

$$\frac{\begin{array}{c} E_1 \vdash a_1 : \tau_1/\varphi \qquad \vdash p : \tau_1 \Rightarrow E' \qquad \vdash \tau_1 - p \rightsquigarrow \tau_2 \\ E_1 \oplus E' \vdash a_2 : \tau/\varphi \qquad E_1 \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E_1 \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi}$$

qui étend nos hypothèses par ses prémisses.

Nous cherchons donc à reconstituer une telle dérivation, terminant par notre but, donc de la forme :

$$\frac{}{E_2 \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi}$$

Établissons-en donc les prémisses. La première est simplement obtenue par application de l'hypothèse de récurrence et nous donne que $E_2 \vdash a_1 : \tau_1/\varphi$.

Les deux prémisses suivantes ne font intervenir aucun environnement de typage, donc restent valides sous l'utilisation de E_2 à la place de E_1.

Puisque par hypothèse $E_2 \geq E_1$, trivialement $E_2 \oplus E' \geq E_1 \oplus E'$ donc nous pouvons à nouveau appliquer la récurrence sur la quatrième prémisse de notre règle de départ pour en déduire la quatrième prémisse de la règle que nous voulons reconstituer, ce qui nous donne bien que $E_2 \oplus E' \vdash a_2 : \tau/\varphi$.

De la même façon, $E_2 \oplus \{x : \tau_2\} \geq E_1 \oplus \{x : \tau_2\}$, et nous appliquons l'hypothèse de récurrence pour déduire notre dernière prémisse.

Toutes nos prémisses étant maintenant réunies, nous reconstruisons notre dérivation et closons ce cas de notre démonstration.

Cas a est C.

Il est aisé de voir que ce cas-ci se traite de la même façon que celui des constantes entières.

Cas a est $D(a)$.

Nous savons que $E_1 \vdash D(a) : \texttt{exn}[D(\tau);\ \varphi']/\varphi$ et nous désirons montrer que

$$E_2 \vdash D(a) : \texttt{exn}[D(\tau);\ \varphi']/\varphi$$

De notre hypothèse de départ, nous pouvons utiliser une dérivation 17 de la forme :

$$\frac{\tau \leq \text{TypeArg}(D) \qquad E_1 \vdash a : \tau/\varphi \qquad \vdash \varphi' :: \texttt{EXN}(\{D\}) \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E_1 \vdash D(a) : \texttt{exn}[D(\tau);\ \varphi']/\varphi}$$

Pour reconstituer une règle similaire terminée par notre but, il nous suffit d'appliquer l'hypothèse de récurrence sur sa seconde prémisse. Les autres prémisses restent trivialement valables sous E_2 plutôt que E_1. Nous pouvons donc reconstituer une dérivation comme attendu et clore ce cas de notre preuve.

Cas a est $\texttt{try}\ a_1\ \texttt{with}\ x \to a_2$.

Nous savons que $E_1 \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau/\varphi$. Nous voulons montrer que :

$$E_2 \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau/\varphi$$

De notre hypothèse de départ, nous pouvons utiliser une dérivation 18 de la forme :

$$\frac{E_1 \vdash a_1 : \tau/\varphi' \qquad E_1 \oplus \{x : \texttt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi}{E_1 \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau/\varphi}$$

Nous voulons donc reconstituer une dérivation de cette forme dont la conclusion sera le but que nous recherchons :

$$\frac{\dots}{E_2 \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau/\varphi}$$

Pour y parvenir, nous reconstituons les prémisses d'une telle règle. La première est aisément obtenue par application de l'hypothèse de récurrence sur la prémisse de gauche de la règle de départ, ce qui nous donne bien $E_2 \vdash a_1 : \tau/\varphi'$.

Quant à la seconde, nous voudrions l'obtenir également par récurrence sur l'autre prémisse de la règle de départ. Pour appliquer cette récurrence, il faut tout d'abord montrer que $E_2 \oplus \{x : \texttt{exn}[\varphi']\} \geq E_1 \oplus \{x : \texttt{exn}[\varphi']\}$, ce qui est trivialement vérifié puisque par hypothèse $E_2 \geq E_1$. Il faut également montrer que les domaines de ces deux environnements sont identiques, ce qui est aussi trivial puisque par hypothèse $\mathrm{Dom}\,(E_1) = \mathrm{Dom}\,(E_2)$. Ainsi nous appliquons maintenant notre hypothèse de récurrence, déduisons effectivement que $E_2 \oplus \{x : \texttt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi$, ce qui nous permet de reconstituer notre dérivation et ainsi de clore cette preuve. □

5.3 Correction du typage

Lemme 5.14 (Les valeurs n'ont pas effet) *Soit v une valeur et E_0 l'environnement de typage initial défini précédemment (cf. section 4.2). Supposons que $E_0 \vdash v : \tau/\varphi$. Alors, pour toute rangée φ' de sorte $\texttt{EXN}(\emptyset)$, nous avons également $E_0 \vdash v : \tau/\varphi'$.*

Preuve : La preuve s'effectue simplement en examinant les règles de typage pouvant s'appliquer aux valeurs (c'est-à-dire les règles 11, 12, 16 et 17). Pour chacune d'entre elles, nous pouvons voir que l'effet de l'expression est effectivement choisi non contraint et de sorte $\texttt{EXN}(\emptyset)$. □

Lemme 5.15 (Les valeurs sont indépendantes d'un environnement bien formé) *Soit v une valeur, et E_0 l'environnement de typage initial défini précédemment (cf. section 4.2). Supposons que $E_0 \vdash v : \tau/\varphi$. Alors, pour tout environnement de typage E étendant E_0 (c'est-à-dire tel que $E(\texttt{raise}) = E_0(\texttt{raise})$), nous avons également $E \vdash v : \tau/\varphi$.*

Preuve : Nous avons $\mathrm{Dom}\,(E_0) \subseteq \mathrm{Dom}\,(E)$ et $E_0(\texttt{raise}) \leq E(\texttt{raise})$. Donc nous pouvons appliquer le lemme disant que le typage est stable sous hypothèses renforcées (lemme 5.13). Ainsi, ayant $E_0 \vdash v : \tau/\varphi$, nous en déduisons que $E \vdash v : \tau/\varphi$, résultat attendu. □

Dans la suite, nous écrivons $\vdash v : \tau$ comme abréviation pour « il existe un φ tel que $E_0 \vdash v : \tau/\varphi$ ». Par les lemmes précédents (5.14 et 5.15), s'il existe un tel φ, alors $E \vdash v : \tau/\varphi$ est vérifié pour tout φ de sorte $\texttt{EXN}(\emptyset)$ et tout environnement E tel que $E(\texttt{raise}) = E_0(\texttt{raise})$.

Lemme 5.16 (Lemme de substitution) *Supposons que $E(\texttt{raise}) = E_0(\texttt{raise})$. Supposons de plus $\vdash v : \tau'$ et $E \oplus \{x : \forall\alpha_1 \ldots \alpha_n.\tau'\} \vdash a : \tau/\varphi$ où les variables $\alpha_1 \ldots \alpha_n$ ne sont pas libres dans E ni dans φ. Alors, $E \vdash a\{x \leftarrow v\} : \tau/\varphi$.*

Preuve : Cette preuve s'effectue par induction sur a. Dans ce qui suit, nous notons E_x pour l'environnement de typage $E \oplus \{x : \forall\alpha_1 \ldots \alpha_n.\tau'\}$.

Cas a est x.

Hypothèses :

$$\vdash v : \tau' \text{ c'est-à-dire } E_0 \vdash v : \tau'/\overline{\varphi} \tag{5.1}$$
$$E_x \vdash x : \tau/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash x\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

c'est-à-dire que :

$$E \vdash v : \tau/\varphi \tag{5.4}$$

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 10 de la forme :

$$\frac{\tau \leq E_x(x) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E_x \vdash x : \tau/\varphi}$$

dont les prémisses étendent nos hypothèses.

Puisque $E_x(x) = \forall \alpha_1 \ldots \alpha_n . \tau'$, et que $\tau \leq E_x(x)$, c'est que $\tau \leq \forall \alpha_1 \ldots \alpha_n . \tau'$, donc que $\tau = \theta(\tau')$ pour θ une substitution opérant sur les α_i.

L'hypothèse 5.1 nous dit que $E_0 \vdash v : \tau'/\overline{\varphi}$. Par le lemme disant que les valeurs n'ont pas d'effet (lemme 5.14), nous en déduisons que $E_0 \vdash v : \tau'/\varphi$ (puisqu'il nous autorise à choisir l'effet de la valeur comme nous le désirons).

L'environnement E étant tel que $E(\mathtt{raise}) = \mathtt{E}_0(\mathtt{raise})$, nous pouvons appliquer le lemme 5.15 et déduire que $E \vdash v : \tau'/\varphi$.

Appliquons maintenant le lemme de stabilité du typage par substitution (5.12, point 3). Il nous vient que $\theta(E) \vdash v : \theta(\tau')/\theta(\varphi)$, c'est-à-dire que $E \vdash v : \theta(\tau')/\varphi$ puisque les α_i ne sont pas libres (n'apparaissent pas) dans E ni dans φ.

Comme nous venons de voir que $\tau = \theta(\tau')$, nous déduisons de ceci que $E \vdash v : \tau/\varphi$, ce qui est ce que nous voulions démontrer en 5.4.

Cas a est i.

Hypothèses :

$$\vdash v : \tau' \text{ c'est-à-dire } E_0 \vdash v : \tau'/\overline{\varphi} \tag{5.1}$$
$$E_x \vdash i : \tau/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash i\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

c'est-à-dire que :

$$E \vdash i : \tau/\varphi \tag{5.4}$$

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 11 de la forme :

$$\frac{\vdash \varphi' :: \mathtt{INT}(\{i\}) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E_x \vdash i : \mathtt{int}[i : \mathtt{Pre};\, \varphi']/\varphi}$$

dont les prémisses étendent nos hypothèses.

De cette dérivation et de l'hypothèse 5.2 nous déduisons que $\tau = \mathtt{int}[i : \mathtt{Pre};\, \varphi']$. Puisque le typage de i ne requiert aucun environnement de typage particulier, nous pouvons utiliser

E plutôt que E_x, ainsi nous avons bien que $E \vdash i : \tau/\varphi$, ce qui est bien le but attendu en 5.4.

Cas a est $\lambda y.a$.
Hypothèses :

$$\vdash v : \tau' \text{ c'est-à-dire } E_0 \vdash v : \tau'/\overline{\varphi} \tag{5.1}$$

$$E_x \vdash \lambda y.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash (\lambda y.a)\{x \leftarrow v\} : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi \tag{5.3}$$

Note : si $x = y$, alors nous renommons y de façon à ce que y ne soit pas libre (c'est-à-dire n'apparaisse pas) dans E. Si $y = \texttt{raise}$, nous renommons également y.

Pour avoir obtenu 5.2 nous avons dû forcément utiliser une règle 12 de la forme :

$$\frac{\vdash \tau_1 \text{ bf} \qquad E_x \oplus \{y : \tau_1\} \vdash a : \tau_2/\varphi' \qquad \vdash \varphi :: \texttt{EXN}(\emptyset)}{E_x \vdash \lambda y.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}$$

dont les prémisses étendent nos hypothèses.

Nous voulons obtenir une dérivation dont la conclusion est notre but 5.3, c'est-à-dire de la forme :

$$\frac{\cdots}{E \vdash (\lambda y.a)\{x \leftarrow v\} : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}$$

donc nous cherchons à en montrer les prémisses, à savoir :

$$\vdash \tau_1 \text{ bf} \tag{5.4}$$

$$E \oplus \{y : \tau_1\} \vdash a\{x \leftarrow v\} : \tau_2/\varphi' \tag{5.5}$$

$$\vdash \varphi :: \texttt{EXN}(\emptyset) \tag{5.6}$$

Les buts 5.4 et 5.6 nous sont déjà fournis en tant qu'hypothèses induites par l'utilisation de la dérivation ci-dessus.

Nous devons donc maintenant montrer 5.5. Nous pouvons donc appliquer l'hypothèse de récurrence sur a de la règle de départ et nous obtenons bien 5.5.

Munis de ces trois prémisses, nous pouvons donc maintenant reconstituer une dérivation 12 comme souhaité ce qui clos ce cas de la preuve.

Cas a est $a_1(a_2)$.
Hypothèses :

$$\vdash v : \tau' \tag{5.1}$$

$$E_x \vdash a_1(a_2) : \tau/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash (a_1(a_2))\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

c'est-à-dire :

$$E \vdash a_1\{x \leftarrow v\}(a_2\{x \leftarrow v\}) : \tau/\varphi \tag{5.4}$$

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 13 de la forme :

$$\frac{E_x \vdash a_1 : (\tau'' \xrightarrow{\varphi} \tau)/\varphi \qquad E_x \vdash a_2 : \tau''/\varphi}{E_x \vdash a_1(a_2) : \tau/\varphi}$$

dont les prémisses étendent nos hypothèses.

Nous voulons obtenir une dérivation dont la conclusion est notre but 5.4, c'est-à-dire de la forme :

$$\frac{\cdots}{E \vdash a_1\{x \leftarrow v\}(a_2\{x \leftarrow v\}) : \tau/\varphi}$$

donc nous cherchons à en montrer les prémisses, à savoir :

$$E \vdash a_1\{x \leftarrow v\} : (\tau'' \xrightarrow{\varphi} \tau)/\varphi \tag{5.5}$$

$$E \vdash a_2\{x \leftarrow v\} : \tau''/\varphi \tag{5.6}$$

Pour obtenir chacun de ces deux buts, nous pouvons appliquer directement l'hypothèse de récurrence sur les prémisses correspondantes dans la dérivation de départ. Ainsi nous pouvons reconstituer une dérivation 13 comme désiré et clore ce cas de la preuve.

Cas a est $\texttt{let}\ y = a_1\ \texttt{in}\ a_2$.

Hypothèses :

$$\vdash v : \tau' \text{ c'est-à-dire } E_0 \vdash v : \tau' \tag{5.1}$$

$$E_x \vdash \texttt{let}\ y = a_1\ \texttt{in}\ a_2 : \tau/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash (\texttt{let}\ y = a_1\ \texttt{in}\ a_2)\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

c'est-à-dire :

$$E \vdash \texttt{let}\ y = a_1\{x \leftarrow v\}\ \texttt{in}\ a_2\{x \leftarrow v\} : \tau/\varphi \tag{5.4}$$

Note : si $x = y$, alors nous renommons y de façon à ce que y ne soit pas libre (c'est-à-dire n'apparaisse pas) dans E. Si $y = \texttt{raise}$, nous renommons également y.

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 14 de la forme :

$$\frac{E_x \vdash a_1 : \tau_1/\varphi \qquad E_x \oplus \{y : \mathrm{Gen}\,(\tau_1, E_x, \varphi)\} \vdash a_2 : \tau/\varphi}{E_x \vdash \texttt{let}\ y = a_1\ \texttt{in}\ a_2 : \tau/\varphi}$$

dont les prémisses étendent nos hypothèses.

Nous voulons obtenir une dérivation dont la conclusion est notre but 5.4, c'est-à-dire de la forme :

$$\frac{\cdots}{E \vdash \texttt{let}\ y = a_1\{x \leftarrow v\}\ \texttt{in}\ a_2\{x \leftarrow v\} : \tau/\varphi}$$

donc nous cherchons à en montrer les prémisses, à savoir :

$$E \vdash a_1\{x \leftarrow v\} : \tau_1/\varphi \tag{5.5}$$

$$E \oplus \{y : \mathrm{Gen}\,(\tau_1, E, \varphi)\} \vdash a_2\{x \leftarrow v\} : \tau/\varphi \tag{5.6}$$

Pour obtenir la première prémisse (5.5) nous pouvons directement appliquer l'hypothèse de récurrence sur E_x et a_1.

Pour obtenir la seconde prémisse (5.6), nous appliquons maintenant l'hypothèse de récurrence à la prémisse de droite de notre règle de départ et à a_2

Ainsi, nous reconstituons la dérivation 14 attendue ce qui clos ce cas de la preuve.

Cas a est $\texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid y \rightarrow a_3$.

Hypothèses :

$$\vdash v : \tau' \tag{5.1}$$
$$E_x \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid y \to a_3 : \tau/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash (\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid y \to a_3)\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

c'est-à-dire :

$$E \vdash \texttt{match}\ a_1\{x \leftarrow v\}\ \texttt{with}\ p \to a_2\{x \leftarrow v\} \mid y \to a_3\{x \leftarrow v\} : \tau/\varphi \tag{5.4}$$

Note : si $x = y$, alors nous renommons y de façon à ce que y ne soit pas libre (c'est-à-dire n'apparaisse pas) dans E.

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 15 de la forme :

$$\frac{\begin{array}{c} E_x \vdash a_1 : \tau_1/\varphi \qquad \vdash p : \tau_1 \Rightarrow E' \qquad \vdash \tau_1 - p \rightsquigarrow \tau_2 \\ E_x \oplus E' \vdash a_2 : \tau/\varphi \qquad E_x \oplus \{y : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E_x \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid y \to a_3 : \tau/\varphi}$$

dont les prémisses étendent nos hypothèses.

Nous voulons obtenir une dérivation dont la conclusion est notre but 5.4, c'est-à-dire de la forme :

$$\frac{\ldots}{E \vdash \texttt{match}\ a_1\{x \leftarrow v\}\ \texttt{with}\ p \to a_2\{x \leftarrow v\} \mid y \to a_3\{x \leftarrow v\} : \tau/\varphi}$$

donc nous cherchons à en montrer les prémisses, à savoir :

$$E \vdash a_1\{x \leftarrow v\} : \tau_1/\varphi \tag{5.5}$$
$$\vdash p : \tau_1 \Rightarrow E' \tag{5.6}$$
$$\vdash \tau_1 - p \rightsquigarrow \tau_2 \tag{5.7}$$
$$E \oplus E' \vdash a_2\{x \leftarrow v\} : \tau/\varphi \tag{5.8}$$
$$E \oplus \{y : \tau_2\} \vdash a_3\{x \leftarrow v\} : \tau/\varphi \tag{5.9}$$

Les exigences 5.6 et 5.7 ne nécessitant pas de référence à l'environnement restent ainsi valables de nos hypothèse de l'existence d'une dérivation 15.

Nous pouvons facilement déduire 5.5, 5.8 et 5.9 par hypothèse de récurrence appliquée à E_x et respectivement à a_1, a_2 et a_3 de la règle 15 de départ.

Ainsi nous reconstituons notre dérivation attendue et closons ce cas de la preuve.

Cas a est C.

On se convaincra aisément que ce cas est en tout point similaire à celui où a est i démontré précédemment.

Cas a est $D(a)$.

Hypothèses :

$$\vdash v : \tau' \tag{5.1}$$
$$E_x \vdash D(a) : \texttt{exn}[D(\tau);\ \varphi']/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash (D(a))\{x \leftarrow v\} : \texttt{exn}[D(\tau);\ \varphi']/\varphi \tag{5.3}$$

c'est-à-dire :

$$E \vdash D(a\{x \leftarrow v\}) : \mathtt{exn}[D(\tau);\ \varphi']/\varphi \tag{5.4}$$

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 17 de la forme :

$$\frac{\tau \leq \mathrm{TypeArg}(D) \qquad E_x \vdash a : \tau/\varphi \qquad \vdash \varphi' :: \mathtt{EXN}(\{D\}) \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E_x \vdash D(a) : \mathtt{exn}[D(\tau);\ \varphi']/\varphi}$$

dont les prémisses étendent nos hypothèses.

Nous voulons obtenir une dérivation dont la conclusion est notre but 5.4, c'est-à-dire :

$$\frac{\dots}{E \vdash D(a\{x \leftarrow v\}) : \mathtt{exn}[D(\tau);\ \varphi']/\varphi}$$

donc nous cherchons à en montrer les prémisses, à savoir :

$$\tau \leq \mathrm{TypeArg}(D) \tag{5.5}$$
$$E_x \vdash a\{x \leftarrow v\} : \tau/\varphi \tag{5.6}$$
$$\vdash \varphi' :: \mathtt{EXN}(\{D\}) \tag{5.7}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{5.8}$$

Les prémisses 5.5, 5.7 et 5.8 nous sont directement fournies par l'hypothèse d'existence de notre dérivation 17 de départ.

Quant à la prémisse 5.6, nous l'obtenons par application de l'hypothèse de récurrence sur E_x et a de notre règle de départ.

Ainsi nous reconstituons la dérivation 17 attendue et closons ce cas de la preuve.

Cas a est $\mathtt{try}\ a_1\ \mathtt{with}\ y \rightarrow a_2$.

Hypothèses :

$$\vdash v : \tau' \tag{5.1}$$
$$E_x \vdash \mathtt{try}\ a_1\ \mathtt{with}\ y \rightarrow a_2 : \tau/\varphi \tag{5.2}$$

On veut prouver que :

$$E \vdash (\mathtt{try}\ a_1\ \mathtt{with}\ y \rightarrow a_2)\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

c'est-à-dire :

$$E \vdash \mathtt{try}\ a_1\{x \leftarrow v\}\ \mathtt{with}\ y \rightarrow a_2\{x \leftarrow v\} : \tau/\varphi \tag{5.4}$$

Note : si $x = y$, alors nous renommons y de façon à ce que y ne soit pas libre (c'est-à-dire n'apparaisse pas) dans E.

Pour avoir obtenu 5.2 nous avons dû utiliser une règle 18 de la forme :

$$\frac{E_x \vdash a_1 : \tau/\varphi' \qquad E_x \oplus \{x : \mathtt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi}{E_x \vdash \mathtt{try}\ a_1\ \mathtt{with}\ y \rightarrow a_2 : \tau/\varphi}$$

dont les prémisses étendent nos hypothèses.

Nous voulons obtenir une dérivation dont la conclusion est notre but 5.4, c'est-à-dire :

$$\frac{\dots}{E \vdash \mathtt{try}\ a_1\{x \leftarrow v\}\ \mathtt{with}\ y \rightarrow a_2\{x \leftarrow v\} : \tau/\varphi}$$

donc nous cherchons à en montrer les prémisses, à savoir :

$$E \vdash a_1\{x \leftarrow v\} : \tau/\varphi' \tag{5.5}$$

$$E \oplus \{x : \mathtt{exn}[\varphi']\} \vdash a_2\{x \leftarrow v\} : \tau/\varphi \tag{5.6}$$

Ces deux prémisses sont aisément obtenues par application de l'hypothèse de récurrence à chacune des prémisses correspondantes de la règle de départ. Nous pouvons ainsi reconstituer la dérivation 18 attendue et clore ce cas de la preuve.

□

Lemme 5.17 (Lemme de substitution pour le filtrage) *Supposons que* $\vdash v : \tau'$, *que* $\vdash p : \tau' \Rightarrow E'$ *et que* $E \oplus E' \vdash a : \tau/\varphi$. *Si* $\xi = M(v,p)$ *est défini, alors* $E \vdash \xi(a) : \tau/\varphi$.

Preuve : Cette preuve s'effectue simplement par induction structurelle sur les motifs p.

Cas p est x.
Hypothèses :

$$E \vdash v : \tau' \tag{5.1}$$

$$\vdash x : \tau' \Rightarrow E' \tag{5.2}$$

$$E \oplus E' \vdash a : \tau/\varphi \tag{5.3}$$

$$\xi = M(v,x) \text{ est défini} \tag{5.4}$$

On veut prouver que :

$$E \vdash \xi(a) : \tau/\varphi \tag{5.5}$$

Puisque $M(v,x)$ est défini, ξ est de la forme $\{x \leftarrow v\}$. Donc notre but revient à démontrer maintenant que :

$$E \vdash a\{x \leftarrow v\} : \tau/\varphi \tag{5.6}$$

Partant de notre hypothèse 5.2, nous utilisons la règle 19 qui nous dit que E' est de la forme : $\{x : \tau'\}$. Donc en combinant cette déduction avec l'hypothèse 5.3, nous déduisons que $E \oplus \{x : \tau'\} \vdash a : \tau/\varphi$. Ceci ajouté à l'hypothèse 5.1 représente exactement les conditions d'application du lemme de substitution (lemme 5.16) démontré précédemment. Nous l'appliquons donc, ce qui nous donne que : $E \vdash a\{x \leftarrow v\} : \tau/\varphi$, exactement ce que nous attendions en but 5.6 et clos ce cas de la preuve.

Cas p est i.
Hypothèses :

$$E \vdash v : \tau' \tag{5.1}$$

$$\vdash i : \tau' \Rightarrow E' \tag{5.2}$$

$$E \oplus E' \vdash a : \tau/\varphi \tag{5.3}$$

$$\xi = M(v,i) \text{ est défini} \tag{5.4}$$

On veut prouver que :

$$E \vdash \xi(a) : \tau/\varphi \tag{5.5}$$

Notre hypothèse 5.2 implique, par la règle 20 que :

$$\tau' = \mathtt{int}[i : \mathtt{Pre};\ \varphi] \tag{5.6}$$

$$E' = \{\} \tag{5.7}$$

De plus, puisque $M(v,x)$ est défini, ξ est de la forme $\{\}$. Donc notre but revient à démontrer maintenant que :

$$E \vdash a : \tau/\varphi \tag{5.8}$$

La combinaison de l'hypothèse 5.3 et de la déduction 5.7 nous donne directement notre but, ce qui clos ce cas de la preuve.

Cas p est C.
On se convaincra aisément que ce cas est similaire à p est i démontré ci-dessus.

Cas p est $D(p)$.
Hypothèses :

$$E \vdash v : \tau' \tag{5.1}$$
$$\vdash D(p) : \tau' \Rightarrow E' \tag{5.2}$$
$$E \oplus E' \vdash a : \tau/\varphi \tag{5.3}$$
$$\xi = M(v, D(p)) \text{ est défini} \tag{5.4}$$

On veut prouver que :

$$E \vdash \xi(a) : \tau/\varphi \tag{5.5}$$

Notre hypothèse 5.2 implique, par la règle 22 que :

$$\tau' = \texttt{exn}[D(\tau''); \varphi''] \tag{5.6}$$
$$p : \tau'' \Rightarrow E' \tag{5.7}$$

De plus, puisque ξ est définie (hypothèse 5.4), nous déduisons par définition de M dans ce cas que :

$$v \text{ est de la forme } D(v') \tag{5.8}$$
$$\xi = M(v', p) \text{ est défini} \tag{5.9}$$

De 5.1, 5.6 et 5.8 nous déduisons maintenant que :

$$v' : \tau'' \tag{5.10}$$

L'hypothèse 5.3 et les déductions 5.9, 5.10 et 5.7 sont les conditions d'application de l'hypothèse de récurrence et nous permettent ainsi d'établir que $E \vdash \xi(a) : \tau/\varphi$, ce qui est le but 5.5 attendu.

□

Nous dirons qu'une valeur v *appartient à une rangée* φ si l'une des propriétés suivantes est vérifiée :
- v est un entier i et $\varphi = i : \texttt{Pre}; \varphi'$ pour un certain φ' ;
- v est une exception constante C et $\varphi = C : \texttt{Pre}; \varphi'$ pour un certain φ' ;
- v est une exception paramétrée $D(v')$ et $\varphi = D(\tau'); \varphi'$ pour un certain φ' et un certain τ' tel que $\vdash v' : \tau'$.

Lemme 5.18 (Forme des valeurs selon leur type) *Soit v une valeur. Supposons que $\vdash v : \tau$.*
- *Si τ est $\texttt{int}[\varphi]$, alors v est un entier i, et ce i appartient à φ.*
- *Si τ est $\texttt{exn}[\varphi]$, alors v est soit C soit $D(v')$, et dans les deux cas v appartient à φ.*
- *Si τ est $\tau_1 \xrightarrow{\varphi} \tau_2$, alors v est une fonction $\lambda x.a$, et $E_0 \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi$.*

Preuve : La preuve est effectuée par simple examen des règles de typage pouvant s'appliquer dans le cas des valeurs (règles 11, 12, 16 et 17) et n'appelle pas de commentaire particulier. □

Lemme 5.19 (Correction de la soustraction) *Si* $\vdash v : \tau$ *et* $M(v,p)$ *est indéfini (*v *n'est pas filtrée par le motif* p*) et* $\vdash \tau - p \rightsquigarrow \tau'$*, alors* $\vdash v : \tau'$*.*

Preuve : Cette preuve est réalisée par induction sur le motif p.

Cas p est x. Ce cas est trivialement vérifié puisque par définition de M, $M(p,v)$ est toujours défini quelle que soit v.

Cas p est i.

Hypothèses :

$$\vdash v : \tau \tag{5.1}$$
$$M(v,i) \text{ est indéfini} \tag{5.2}$$
$$\vdash \tau - i \rightsquigarrow \tau' \tag{5.3}$$

On veut prouver que :

$$\vdash v : \tau' \tag{5.4}$$

L'hypothèse 5.3 implique par la règle 23 que :

$$\tau = \mathtt{int}[i : \mathtt{Pre};\ \varphi] \tag{5.5}$$
$$\tau' = \mathtt{int}[i : \pi;\ \varphi] \tag{5.6}$$

Puisque $v : \tau$ (hypothèse 5.1), et que $\tau = \mathtt{int}[i : \mathtt{Pre};\ \varphi]$ (déduction 5.5), le lemme précédent (lemme 5.19) nous indique que v est un entier j et que j appartient à la rangée $i : \mathtt{Pre};\ \varphi$.

Le fait que $M(v,i)$ soit indéfini (hypothèse 5.2) implique de plus que $i \neq j$, donc j appartient à la rangée φ. En d'autres termes, $\varphi = j : \mathtt{Pre};\ \varphi'$ pour un certain φ', et donc par l'hypothèse 5.6, $\tau' = \mathtt{int}[j : \mathtt{Pre};\ i : \pi;\ \varphi']$.

Ainsi, par une règle 11, nous pouvons dériver que $\vdash j : \tau'$ et comme nous avons vu que notre valeur v est j, ceci revient bien à notre but 5.4.

Cas p est C.

On se convaincra aisément que ce cas est similaire au cas précédent.

Cas p est $D(p)$.

Hypothèses :

$$\vdash v : \tau \tag{5.1}$$
$$M(v, D(p)) \text{ est indéfini} \tag{5.2}$$
$$\vdash \tau - D(p) \rightsquigarrow \tau' \tag{5.3}$$

On veut prouver que :

$$\vdash v : \tau' \tag{5.4}$$

L'hypothèse 5.3 implique par la règle 26 que :

$$\tau = \mathtt{exn}[D(\tau_1);\ \varphi] \tag{5.5}$$
$$\tau' = \mathtt{exn}[D(\tau_1');\ \varphi] \tag{5.6}$$
$$\vdash \tau_1 - p_1 \rightsquigarrow \tau_1' \tag{5.7}$$

À partir de l'hypothèse 5.1 que v a le type τ, le lemme 5.18 montre que v est un terme construit $D(v_1)$ avec $\vdash v_1 : \tau_1$. De plus, (par hypothèse 5.2) $M(v_1, p_1)$ est indéfini sinon $M(v, p)$ serait défini.

En appliquant l'hypothèse de récurrence à v_1 et p_1, nous obtenons que $\vdash v_1 : \tau_1'$. En utilisant une dérivation 17, nous pouvons alors en déduire que $\vdash v : \tau'$ ce qui est le résultat attendu et ainsi clore ce cas de la preuve.

□

Lemme 5.20 (Effet des exceptions) *$E_0 \vdash \texttt{raise}\ v : \tau/\varphi$ si et seulement si v appartient à φ.*

Preuve : Une dérivation de typage pour $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$ a la forme suivante :

$$\dfrac{\dfrac{\texttt{exn}[\varphi] \xrightarrow{\varphi} \tau \leq E_0(\texttt{raise})}{E_0 \vdash \texttt{raise} : \texttt{exn}[\varphi] \xrightarrow{\varphi} \tau} \qquad E_0 \vdash v : \texttt{exn}[\varphi]/\varphi}{E_0 \vdash \texttt{raise}\ v : \tau/\varphi}$$

En utilisant le lemme des forme des valeurs selon leur type (lemme 5.18), si $E_0 \vdash v : \texttt{exn}[\varphi]/\varphi$, alors v appartient à φ.

Inversement, si v appartient à φ, alors nous pouvons dériver $E_0 \vdash v : \texttt{exn}[\varphi]/\varphi$ en utilisant les règles de typage 16 ou 17. □

Théorème 5.1 (Contraction de radical – Subject reduction –) *La réduction préserve les types : si $E_0 \vdash a : \tau/\varphi$ et $a \xrightarrow{*} a'$, alors $E_0 \vdash a' : \tau/\varphi$*

Preuve : Nous montrons tout d'abord ce résultat sur les réductions de tête (règles 2.1–2.11), par cas sur la règle de réduction utilisée.

Cas Règle 2.1 (β-réduction).

Hypothèses :

$$E_0 \vdash (\lambda x.a)(v) : \tau/\varphi \tag{5.1}$$

$$(\lambda x.a)(v) \xrightarrow{*} a\{x \leftarrow v\} \tag{5.2}$$

On veut prouver que :

$$E_0 \vdash a\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

De l'hypothèse 5.1, par la règle 13, nous déduisons que :

$$E_0 \vdash \lambda x.a : (\tau' \xrightarrow{\varphi} \tau)/\varphi \tag{5.4}$$

$$E_0 \vdash v : \tau'/\varphi \tag{5.5}$$

De la déduction 5.4, nous déduisons par une règle 12 que :

$$E_0 \oplus \{x : \tau'\} \vdash a : \tau/\varphi \tag{5.6}$$

Nous appliquons maintenant le lemme de substitution (lemme 5.16) aux déductions 5.5 et 5.6 ce qui nous permet de déduire que $E_0 \vdash a\{x \leftarrow v\} : \tau/\varphi$ ce qui est ce que nous attendions en 5.3.

Cas Règle 2.2 (réduction du `let`).

On se convaincra aisément que ce cas est traité de la même façon que le précédent, en utilisant une dérivation 14 à la place de 13. En effet, la règle de réduction se comporte exactement comme la précédente.

Cas Règle 2.3.
Hypothèses :

$$E_0 \vdash \texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi \tag{5.1}$$
$$\sigma = M(v,p) \text{ est défini} \tag{5.2}$$
$$\texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 \stackrel{*}{\rightarrow} \sigma(a_2) \tag{5.3}$$

On veut prouver que :

$$E_0 \vdash \sigma(a_2) : \tau/\varphi \tag{5.4}$$

De l'hypothèse 5.1, nous déduisons par une règle 15 que :

$$E_0 \vdash v : \tau_1/\varphi \tag{5.5}$$
$$\vdash p : \tau_1 \Rightarrow E' \tag{5.6}$$
$$\vdash \tau_1 - p \rightsquigarrow \tau_2 \tag{5.7}$$
$$E_0 \oplus E' \vdash a_2 : \tau/\varphi \tag{5.8}$$
$$E_0 \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \tag{5.9}$$

Par application du lemme de substitution pour le filtrage (lemme 5.17) aux déductions 5.5, 5.6, 5.8 et à l'hypothèse 5.2, nous déduisons directement que $E_0 \vdash \sigma(a_2) : \tau/\varphi$ ce qui était notre but 5.3.

Cas Règle 2.4.
Hypothèses :

$$E_0 \vdash \texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi \tag{5.1}$$
$$\sigma = M(v,p) \text{ est indéfini} \tag{5.2}$$
$$\texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 \stackrel{*}{\rightarrow} a_3\{x \leftarrow v\} \tag{5.3}$$

On veut prouver que :

$$E_0 \vdash a_3\{x \leftarrow v\} : \tau/\varphi \tag{5.4}$$

De l'hypothèse 5.1, nous déduisons par une règle 15 que :

$$E_0 \vdash v : \tau_1/\varphi \tag{5.5}$$
$$\vdash p : \tau_1 \Rightarrow E' \tag{5.6}$$
$$\vdash \tau_1 - p \rightsquigarrow \tau_2 \tag{5.7}$$
$$E_0 \oplus E' \vdash a_2 : \tau/\varphi \tag{5.8}$$
$$E_0 \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \tag{5.9}$$

Puisque $M(v,p)$ est indéfini (hypothèse 5.2), et que $\vdash \tau_1 - p \rightsquigarrow \tau_2$ (déduction 5.7), nous pouvons déduire, par le lemme de correction de la soustraction (lemme 5.19), que :

$$\vdash v : \tau' \tag{5.10}$$

où τ' est le type obtenu en soustrayant p à τ.

Nous pouvons alors appliquer le lemme de substitution (lemme 5.16) aux déductions 5.9 et 5.10, ce qui établit que $E_0 \vdash a_3\{x \leftarrow v\} : \tau/\varphi$, résultat attendu en 5.4.

Cas Règle 2.5.

Ce cas est trivial puisque $\vdash v : \tau$ par hypothèse.

Cas Règles 2.6.
Hypothèses :

$$E_0 \vdash (\texttt{raise}\ v)(a) : \tau/\varphi \tag{5.1}$$

$$(\texttt{raise}\ v)(a) \overset{*}{\to} \texttt{raise}\ v \tag{5.2}$$

On veut prouver que :

$$E_0 \vdash \texttt{raise}\ v : \tau/\varphi \tag{5.3}$$

De l'hypothèse 5.1, par une dérivation 13, nous déduisons que :

$$E_0 \vdash \texttt{raise}\ v : (\tau' \overset{\varphi}{\to} \tau)/\varphi \tag{5.4}$$

$$E_0 \vdash a : \tau'/\varphi \tag{5.5}$$

En appliquant une première fois le lemme d'effet des exceptions (lemme 5.20) à 5.4, (utilisation de l'équivalence dans le sens $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$), nous déduisons que :

$$v \text{ appartient à } \varphi \tag{5.6}$$

Nous appliquons alors ce même lemme à 5.6 (en utilisant l'équivalence dans le sens inverse) ce qui nous permet de choisir le type que nous voulons pour cette expression, (à savoir τ et non $\tau' \overset{\varphi}{\to} \tau$ comme nous donnait la déduction 5.4 et ainsi de déduire le but 5.3 attendu.

Cas Règles 2.6 à 2.10.
Ces cas peuvent se regrouper et être traités similairement au cas précédent. Afin de ne pas allonger inutilement cette preuve mais de continuer à l'exposer de façon relativement complète, résumons le schéma directeur de tous ces cas.

Dans toutes ces règles, a' est $\texttt{raise}\ v$ et a est une expression contenant $\texttt{raise}\ v$ comme sous-expression en dehors d'une construction `try`. Ainsi, dans une dérivation de $E_0 \vdash a : \tau/\varphi$, une sous-expression $\texttt{raise}\ v$ se verra assigné l'effet φ. Par le lemme d'effet des exceptions (lemme 5.20), il s'ensuit que v appartient à φ, et que $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$.

Cas Règle 2.11.
Hypothèses :

$$E_0 \vdash \texttt{try raise}\ v\ \texttt{with}\ x \to a_2 : \tau/\varphi \tag{5.1}$$

$$\texttt{try raise}\ v\ \texttt{with}\ x \to a_2 \overset{*}{\to} a_2\{x \leftarrow v\} \tag{5.2}$$

On veut prouver que :

$$E_0 \vdash a_2\{x \leftarrow v\} : \tau/\varphi \tag{5.3}$$

De l'hypothèse 5.1, par une dérivation 13, nous déduisons que :

$$E_0 \vdash \texttt{raise}\ v : \tau/\varphi' \tag{5.4}$$

$$E_0 \oplus \{x : \texttt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi \tag{5.5}$$

De plus, 5.4 implique par existence d'une dérivation adaptée et du schéma supposé de $\texttt{raise}\ (\forall \alpha, \rho.\, \texttt{exn}[\rho] \overset{\rho}{\to} \alpha)$ que $v : \texttt{exn}[\varphi']$.

Nous pouvons alors y appliquer le lemme de substitution (lemme 5.16) avec la déduction 5.5, ce qui nous donne bien que $E_0 \vdash a_2\{x \leftarrow v\} : \tau/\varphi$, but attendu en 5.3.

Ce résultat s'étend alors aux réductions sous contexte Γ (règle 2.12) par une simple induction structurelle sur Γ.

$\square$

Théorème 5.2 (Correction de l'analyse d'exceptions) *Soit a un programme complet. Supposons que $E_0 \vdash a : \tau/\varphi$ et $a \xrightarrow{*}$ `raise` v. Alors, v appartient à φ.*

Preuve : Par le théorème de contraction de radical (5.1) il s'ensuit que $E_0 \vdash$ `raise` $v : \tau/\varphi$. Le lemme d'effets des exceptions (5.20) montre alors que v appartient φ. □

Lemme 5.21 (Progression – Les programmes bien typés ne sont pas bloqués) *Si $E_0 \vdash a : \tau/\varphi$, alors soit a est une valeur v, soit a est une exception non rattrapée* `raise` *v, soit il existe a' tel que $a \xrightarrow{*} a'$.*

Preuve : Cette preuve est effectuée par récurrence sur la structure de a et par cas sur la forme de a.

Cas a est x.
Hypothèse :

$$E_0 \vdash x : \tau/\varphi \tag{5.1}$$

Puisque par hypothèse l'environnement de typage E_0 ne contient que l'identificateur `raise`, nous en déduisons que x est `raise`, donc c'est bien une valeur par hypothèse.

Cas a est i, C ou $\lambda x.a_1$.
Dans tous ces cas, l'expression a est donc, par hypothèse, une valeur.

Cas a est $a_1(a_2)$.
Hypothèse :

$$E_0 \vdash a_1(a_2) : \tau/\varphi \tag{5.1}$$

Par existence d'une dérivation 13 appliquée à 5.1, nous déduisons que :

$$E_0 \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \tag{5.2}$$
$$E_0 \vdash a_2 : \tau'/\varphi \tag{5.3}$$

Nous appliquons l'hypothèse de récurrence à 5.2 ainsi qu'à 5.3, nous déduisons que :

a_1 est une valeur ou une exception non rattrapée ou peut se réduire (5.4)
a_2 est une valeur ou une exception non rattrapée ou peut se réduire (5.5)

Nous devons donc examiner la combinatoire offerte par les trois cas possibles pour chaque expression a_1 et a_2.

- Si a_1 et a_2 sont des valeurs, puisque a_1 a un type fonctionnel, il doit être une λ-abstraction (lemme 5.18), et ainsi a se réduit par la règle 2.1,
- si a_1 est une exception non rattrapée `raise` v, a se réduit par la règle 2.6,
- si a_1 est une valeur et a_2 une exception non rattrapée, alors a_1 doit être une λ-abstraction et a se réduit par la règle 2.7,
- sinon, soit a_1 se réduit soit a_1 est une λ-abstraction et a_2 se réduit. Dans ces deux cas, a se réduit par la règle de réduction au contexte 2.12.

Cas a est `let` $x = a_1$ `in` a_2.
Hypothèse :

$$\texttt{let } x = a_1 \texttt{ in } a_2 : \tau/\varphi \tag{5.1}$$

Par existence d'une dérivation 14 appliquée à 5.1, nous déduisons que :

$$E_0 \vdash a_1 : \tau_1/\varphi \tag{5.2}$$
$$E_0 \oplus \{x : \mathrm{Gen}\,(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi \tag{5.3}$$

Appliquons l'hypothèse de récurrence sur a_1 à partir de 5.2. Il vient que :

$$a_1 \text{ est une valeur ou une exception non rattrapée ou peut se réduire} \tag{5.4}$$

Examinons chacun de ces trois cas :
- Soit a_1 est une valeur et alors a peut se β-réduire par la règle 2.2,
- soit a_1 est une exception non rattrapée et a se réduit par la règle 2.9,
- soit a_1 se réduit et a se réduit également par la règle de réduction dans un contexte (règle 2.12).

Cas a est $\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3$.
Hypothèse :

$$\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi \tag{5.1}$$

Par existence d'une dérivation 15 appliquée à 5.1, nous déduisons que :

$$E_0 \vdash a_1 : \tau_1/\varphi \tag{5.2}$$
$$\vdash p : \tau_1 \Rightarrow E' \tag{5.3}$$
$$\vdash \tau_1 - p \rightsquigarrow \tau_2 \tag{5.4}$$
$$E_0 \oplus E' \vdash a_2 : \tau/\varphi \tag{5.5}$$
$$E_0 \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \tag{5.6}$$

Appliquons l'hypothèse de récurrence sur a_1 à partir de 5.2. Il vient que :

$$a_1 \text{ est une valeur ou une exception non rattrapée ou peut se réduire} \tag{5.7}$$

Examinons chacun de ces trois cas :
- Soit a_1 est une valeur et alors a peut se réduire par la règle 2.3 ou 2.4,
- soit a_1 est une exception non rattrapée et alors a se réduit par la règle 2.10,
- soit a_1 se réduit et alors a se réduit également, par la règle de réduction dans un contexte (règle 2.12).

Cas a est $D(a_1)$.
Hypothèse :

$$D(a_1) : \texttt{exn}[D(\tau);\ \varphi']/\varphi \tag{5.1}$$

Par existence d'une dérivation 17 appliquée à 5.1, nous déduisons que :

$$E_0 \vdash a_1 : \tau/\varphi \tag{5.2}$$

Par induction, puisque a_1 est bien typée dans E_0 (déduction 5.2), a_1 est soit une valeur, soit se réduit, soit est une exception non rattrapée $\texttt{raise}\ v$.

Dans le premier cas, a est une valeur ; dans le second cas, a se réduit par la règle du contexte (règle 2.12) ; dans le dernier cas, a se réduit alors par la règle 2.8.

Cas a est $\texttt{try}\ a_1\ \texttt{with}\ x \to a_2$.
Hypothèse :

$$\texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau/\varphi \tag{5.1}$$

Par existence d'une dérivation 18 appliquée à 5.1, nous déduisons que :

$$E_0 \vdash a_1 : \tau/\varphi' \tag{5.2}$$
$$E_0 \oplus \{x : \texttt{exn}[\varphi']\} \vdash a_2 : \tau/\varphi \tag{5.3}$$

Appliquons l'hypothèse de récurrence sur a_1 à partir de 5.2. Il vient que :

$$a_1 \text{ est une valeur ou une exception non rattrapée ou peut se réduire} \tag{5.4}$$

Examinons chacun de ces trois cas :

- Soit a_1 est une valeur et a peut se réduire par la règle 2.5,
- soit a_1 est une exception non rattrapée et alors a se réduit par la règle 2.11,
- soit a_1 se réduit et dans ce cas a se réduit également, par la règle du contexte (2.12).

□

Théorème 5.3 (Correction du typage) *Soit a un programme complet. Supposons que $E_0 \vdash a : \tau/\varphi$. Si $a \overset{*}{\rightarrow} a'$ et a' est en forme normale vis-à-vis des règles de réduction, alors a' est soit une valeur v, soit une exception non rattrapée* `raise` *v.*

Preuve : Corollaire du théorème de contraction de radical (5.1) et du lemme de progression ci-dessus (5.21). □

Théorème 5.4 (Correction du typage des exceptions) *Si $E_0 \vdash a : \tau/\varphi$ et $a \overset{*}{\rightarrow}$* `raise` *$v$ (l'évaluation de a produit une exception non rattrapée égale à v), alors v appartient à φ.*

Preuve : Corollaire du théorème de correction de l'analyse d'exceptions (5.2) et du lemme de progression ci-dessus (5.21). □

Chapitre 6

L'algorithme d'inférence

Nous avons étudié jusqu'à présent le *système de types* sur lequel notre analyse d'exceptions est basée, c'est-à-dire la *spécification* de cette analyse. Une telle spécification décrit donc les propriétés et résultats de l'analyse.

Comme nous l'avons précisé précédemment, nous désirons que notre analyseur puisse deviner automatiquement les types étiquetant chaque expression (donc par extension les exceptions pouvant s'échapper) à partir du programme *non annoté* fourni par l'utilisateur. Ce processus est appelé *inférence de types* . Dans la présente section, nous allons donc décrire l'algorithme permettant de réaliser cette inférence.

Nous avons vu qu'une expression close a de type τ, était également de type τ' pour tout type τ' moins général que τ. Le type présentant le plus d'intérêt (c'est-à-dire intuitivement autorisant la plus large utilisation de cette expression) est bien entendu le type le plus général. Nous souhaitons donc que notre algorithme d'inférence détermine ce type *le plus général*, parmi tous les types possibles d'une expression.

6.1 Rappels sur l'unification

L'algorithme d'inférence de types repose de manière fondamentale sur *l'unification* entre termes de types. Nous dirons que deux termes τ_1 et τ_2 sont *unifiables* s'il existe une substitution φ telle que $\varphi(\tau_1) = \varphi(\tau_2)$. La substitution φ est alors appelée *unificateur* des termes τ_1 et τ_2. Ainsi deux termes sont unifiables s'il est possible d'instancier tout ou partie de leurs variables par une même substitution, de telle façon qu'ils deviennent structurellement égaux.

Ainsi, les termes de types ML $\alpha \to \mathtt{int}$ et $\mathtt{bool} \to \beta$ sont unifiables car il est possible de leur appliquer la substitution $\{\alpha \leftarrow \mathtt{bool}; \beta \leftarrow \mathtt{int}\}$ afin de les identifier structurellement en $\mathtt{int} \to \mathtt{bool}$.

Il est parfois possible de trouver plusieurs substitutions pour identifier deux termes. Considérons les deux termes $\tau_1 = \alpha \to \beta$ et $\tau_2 = \alpha \to \mathtt{int}$. Les deux substitutions $\varphi_1 = \{\alpha \leftarrow \mathtt{string}; \beta \leftarrow \mathtt{int}\}$ et $\varphi_2 = \{\beta \leftarrow \mathtt{int}\}$ sont des unificateurs de τ_1 et τ_2 puisque $\varphi_1(\tau_1) = \varphi_1(\tau_2) = \mathtt{string} \to \mathtt{int}$ et $\varphi_2(\tau_1) = \varphi_2(\tau_2) = \alpha \to \mathtt{int}$. Nous appellerons *unificateur principal* la substitution qui effectue le moins de modifications aux deux types pour les rendre égaux. Ceci revient à dire que cet unificateur principal μ est une substitution « minimale » et donc que tout autre unificateur φ « inclut » cette substitution et peut alors s'écrire sous la forme $\varphi = \varphi' \circ \mu$. Dans l'exemple précédent, φ_2 est principale mais pas φ_1 ; d'ailleurs $\varphi_1 = \{\alpha \leftarrow \mathtt{string}\} \circ \varphi_2$.

6.2 Formalisation des unifications

Afin de pouvoir raisonner formellement sur l'algorithme d'inférence de notre analyse, nous allons maintenant décrire les algorithmes d'unification sur lesquels il repose.

Puisque notre algèbre de types est conceptuellement subdivisée en quatre parties (types, rangées, éléments de rangées et annotations de présence), nous allons successivement envisager l'algorithme associé à chacune d'entre elles. De plus, cette algèbre étant accompagnée d'une théorie équationnelle (équations (4.1) et (4.2) vues en section 4.1.8), nous allons présenter l'unification modulo ces équations.

Dans ce qui va suivre, les systèmes d'équations bien sortés sont de la forme $Q = \{\tau_i = \tau'_i;\ \varphi_j = \varphi'_j;\ \pi_l = \pi'_l\}$, composés d'équations entre types, rangées, éléments de rangées, et annotations de présence tel que pour tout i, $\vdash \tau_i$ bf et $\vdash \tau'_i$ bf, et pour tout j, il existe une sorte κ_j telle que $\vdash \varphi_j :: \kappa_j$ et $\vdash \varphi'_j :: \kappa_j$.

Dans la présentation de l'algorithme d'unification qui va suivre, afin d'expliciter de façon précise la génération de « variables de rangées fraîches », nous ajoutons un paramètre supplémentaire V à cet algorithme qui devient alors $\mathtt{mgu}_V(Q)$. Ce paramètre est un ensemble de variables qui ne doivent pas être utilisées comme variables fraîches durant l'unification. Nous supposons toujours que V est fini, ainsi il est toujours possible de choisir une variable de rangée hors de V et de n'importe quelle sorte donnée.

Nous définissons le *constructeur de tête* $H(\epsilon)$ d'un élément de rangée ϵ de la façon suivante :

$$H(i : \pi) = i \qquad H(C : \pi) = C \qquad H(D(\tau)) = D$$

L'algorithme d'unification se présente dans le même style que celui de Robinson [51], et prend en compte l'axiome de commutativité à gauche (équation 4.1) de la même façon que dans [47]. C'est-à-dire, pour unifier deux rangées $\epsilon_1;\ \varphi_1$ et $\epsilon_2;\ \varphi_2$ quand les constructeurs de tête de ϵ_1 et ϵ_2 diffèrent, nous prenons une variable de rangée fraîche ρ de sorte appropriée et résolvons les deux équations $\varphi_1 = \epsilon_2;\ \rho$ et $\varphi_2 = \epsilon_1;\ \rho$. Il est facile de voir que toute solution de ces équations convient également pour $\epsilon_1;\ \varphi_1 = \epsilon_2;\ \varphi_2$.

Si aucun des cas énumérés ci-dessous n'est applicable, alors $\mathtt{mgu}_V(Q)$ est indéfini.

$$\mathtt{mgu}_V(\emptyset) \quad = \quad id$$

6.2.1 Unification entre types

$$\begin{array}{rcl}
\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q) & = & \mathtt{mgu}_V(Q) \\
\mathtt{mgu}_V(\{\alpha = \tau\} \cup Q) & = & \mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\} \text{ si } \alpha \notin FV(\tau) \\
\mathtt{mgu}_V(\{\tau = \alpha\} \cup Q) & = & \mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\} \text{ si } \alpha \notin FV(\tau) \\
\mathtt{mgu}_V(\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q) & = & \mathtt{mgu}_V(\{\varphi_1 = \varphi_2\} \cup Q) \\
\mathtt{mgu}_V(\{\mathtt{exn}[\varphi_1] = \mathtt{exn}[\varphi_2]\} \cup Q) & = & \mathtt{mgu}_V(\{\varphi_1 = \varphi_2\} \cup Q) \\
\mathtt{mgu}_V(\{\tau_1 \xrightarrow{\varphi_1} \tau'_1 = \tau_2 \xrightarrow{\varphi_2} \tau'_2\} \cup Q) & = & \mathtt{mgu}_V(\{\tau_1 = \tau_2; \varphi_1 = \varphi_2; \tau'_1 = \tau'_2\} \cup Q)
\end{array}$$

6.2.2 Unification entre rangées

$$
\begin{array}{rcl}
\mathtt{mgu}_V(\{\rho = \rho\} \cup Q) & = & \mathtt{mgu}_V(Q) \\
\mathtt{mgu}_V(\{\rho = \varphi\} \cup Q) & = & \mathtt{mgu}_V(Q\{\rho \leftarrow \varphi\}) \circ \{\rho \leftarrow \varphi\} \text{ si } \rho \notin FV(\varphi) \\
\mathtt{mgu}_V(\{\varphi = \rho\} \cup Q) & = & \mathtt{mgu}_V(Q\{\rho \leftarrow \varphi\}) \circ \{\rho \leftarrow \varphi\} \text{ si } \rho \notin FV(\varphi) \\
\mathtt{mgu}_V(\{\top = \top\} \cup Q) & = & \mathtt{mgu}_V(Q) \\
\mathtt{mgu}_V(\{(i : \pi;\, \varphi) = \top\} \cup Q) & = & \mathtt{mgu}_V(\{\pi = \mathtt{Pre};\, \varphi = \top\} \cup Q) \\
\mathtt{mgu}_V(\{\top = (i : \pi;\, \varphi)\} \cup Q) & = & \mathtt{mgu}_V(\{\pi = \mathtt{Pre};\, \varphi = \top\} \cup Q) \\
\mathtt{mgu}_V(\{(\epsilon_1;\, \varphi_1) = (\epsilon_2;\, \varphi_2)\} \cup Q) & = & \mathtt{mgu}_V(\{\epsilon_1 = \epsilon_2;\, \varphi_1 = \varphi_2\} \cup Q) \\
 & & \text{si } H(\epsilon_1) = H(\epsilon_2) \\
\mathtt{mgu}_V(\{(\epsilon_1;\, \varphi_1) = (\epsilon_2;\, \varphi_2)\} \cup Q) & = & \mathtt{mgu}_{V \cup \{\rho\}}(\{\varphi_1 = (\epsilon_2;\, \rho);\, \varphi_2 = (\epsilon_1;\, \rho)\} \cup Q) \\
 & & \text{si } H(\epsilon_1) \neq H(\epsilon_2) \\
 & & \text{et } \rho \notin V \text{ et } \rho \text{ non libre dans la partie gauche} \\
 & & \text{et } \rho \text{ est de sorte } t(S \cup \{H(\epsilon_1), H(\epsilon_2)\}) \\
 & & \text{où } t(S) \text{ est la sorte de } \epsilon_1;\, \varphi_1 \text{ et } \epsilon_2;\, \varphi_2
\end{array}
$$

6.2.3 Unification entre éléments de rangées

$$
\begin{array}{rcl}
\mathtt{mgu}_V(\{(i : \pi_1) = (i : \pi_2)\} \cup Q) & = & \mathtt{mgu}_V(\{\pi_1 = \pi_2\} \cup Q) \\
\mathtt{mgu}_V(\{(C : \pi_1) = (C : \pi_2)\} \cup Q) & = & \mathtt{mgu}_V(\{\pi_1 = \pi_2\} \cup Q) \\
\mathtt{mgu}_V(\{D(\tau_1) = D(\tau_2)\} \cup Q) & = & \mathtt{mgu}_V(\{\tau_1 = \tau_2\} \cup Q)
\end{array}
$$

6.2.4 Unification entre annotations de présence

$$
\begin{array}{rcl}
\mathtt{mgu}_V(\{\delta = \pi\} \cup Q) & = & \mathtt{mgu}_V(Q\{\delta \leftarrow \pi\}) \circ \{\delta \leftarrow \pi\} \\
\mathtt{mgu}_V(\{\pi = \delta\} \cup Q) & = & \mathtt{mgu}_V(Q\{\delta \leftarrow \pi\}) \circ \{\delta \leftarrow \pi\} \\
\mathtt{mgu}_V(\{\mathtt{Pre} = \mathtt{Pre}\} \cup Q) & = & \mathtt{mgu}_V(Q)
\end{array}
$$

6.3 L'algorithme d'inférence

Maintenant que nous avons vu les algorithmes d'unification sur les éléments de notre algèbre de types, nous allons pouvoir nous concentrer sur l'algorithme d'inférence lui-même, c'est-à-dire celui qui annote les expressions par des types.

L'existence d'un tel algorithme est cruciale pour que notre analyseur d'exceptions puisse fonctionner à partir de programmes sans que l'utilisateur n'ait besoin de spécifier d'annotations de types. Toutefois cette existence ne suffit pas, il est également nécessaire qu'il satisfasse les deux propriétés que nous avons énoncées et que nous allons expliquer informellement ci-dessous.

Intuitivement, sa première propriété (la *correction*) dit que si l'algorithme renvoie un type pour une expression e et un environnement de typage E donnés, alors ce type est correct vis-à-vis des règles de typage énoncées précédemment. Ceci signifie que l'algorithme d'inférence ne donne pas de résultats *faux*.

La seconde propriété de cet algorithme (c'est-à-dire sa *complétude*) expose le fait que s'il existe un type pour une expression a et un environnement de typage E donnés, alors notre algorithme le trouvera. Autrement dit, il n'oublie pas de solution.

Tout comme nous l'avons fait lors de la présentation des algorithmes d'unification, dans la formulation de l'algorithme d'inférence qui suit, nous rendons explicite la notion de variables « fraîches » (laissée implicite dans la plupart des formulations de l'algorithme de Damas-Milner), afin que notre argument de complétude de W puisse être établi de manière plus précise. Ainsi, nous ajoutons un nouveau paramètre V et un nouveau résultat V' a l'algorithme W, qui devient $W(E, a, V) = (\tau, \varphi, \theta, V')$. Le paramètre V est un ensemble de variables de type ne pouvant pas être utilisées comme variables fraîches durant l'exécution de l'algorithme W. Le résultat V' est V plus toutes les variables de type qui ont été utilisées comme variables fraîches durant cette exécution de W (et qui ne doivent plus désormais être utilisées à nouveau comme variables fraîches ultérieurement).

Le résultat de l'algorithme $W(E, a, V)$ est le quadruplet $(\tau, \varphi, \theta, V')$ défini par induction sur une expression a de la façon suivante :

• **Si a est x (avec $x \in \mathrm{Dom}(E)$) :**
soit $\rho \notin V$ une variable de rangée fraîche de sorte $\mathtt{EXN}(\emptyset)$
prendre $(\tau, V') = \mathtt{Inst}(E(x), V \cup \{\rho\})$ et $\varphi = \rho$ et $\theta = id$.

• **Si a est i :**
soit $\rho \notin V$ une variable de rangée fraîche de sorte $\mathtt{INT}(\{i\})$
soit $\rho' \notin V$ une variable de rangée fraîche de sorte $\mathtt{EXN}(\emptyset)$
prendre $\tau = \mathtt{int}[i : \mathtt{Pre}; \rho]$ et $\varphi = \rho'$ et $\theta = id$ et $V' = V \cup \{\rho, \rho'\}$.

• **Si a est $\lambda x.a_1$:**
soit $\alpha \notin V$ une variable de type fraîche
soit $(\tau_1, \varphi_1, \theta_1, V_1) = W(E \oplus \{x : \alpha\}, a_1, V \cup \{\alpha\})$
soit $\rho \notin V_1$ une variable de rangée fraîche de sorte $\mathtt{EXN}(\emptyset)$
prendre $\tau = \theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1$ et $\varphi = \rho$ et $\theta = \theta_1$ et $V' = V_1 \cup \{\rho\}$.

• **Si a est $a_1(a_2)$:**
soit $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1)$
soit $\alpha \notin V_2$ une variable de type fraîche
soit $\mu = \mathtt{mgu}_{V_2 \cup \{\alpha\}}\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha;\ \theta_2(\varphi_1) = \varphi_2\}$
prendre $\tau = \mu(\alpha)$ et $\varphi = \mu(\varphi_2)$ et $\theta = \mu \circ \theta_2 \circ \theta_1$ et $V' = V_2 \cup \{\alpha\} \cup \mathrm{Dom}(\mu) \cup \mathrm{Rng}(\mu)$.

• **Si a est $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$:**
soit $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E) \oplus \{x : \mathrm{Gen}(\tau_1, \theta_1(E), \varphi_1)\}, a_2, V_1)$
soit $\mu = \mathtt{mgu}_{V_2}\{\theta_2(\varphi_1) = \varphi_2\}$
prendre $\tau = \mu(\tau_2)$ et $\varphi = \mu(\varphi_2)$ et $\theta = \mu \circ \theta_2 \circ \theta_1$ et $V' = V_2 \cup \mathrm{Dom}(\mu) \cup \mathrm{Rng}(\mu)$.

• **Si a est $\mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3$:**
soit $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
soit $(E', \tau', \xi, V_1') = \mathtt{Patsubtr}(p, \tau_1, V_1)$
soit $(\tau_2, \varphi_2, \theta_2, V_2) = W(\xi(\theta_1(E)) \oplus E', a_2, V_1')$
soit $(\tau_3, \varphi_3, \theta_3, V_3) = W(\theta_2(\xi(\theta_1(E))) \oplus \{x : \theta_2(\tau')\}, a_3, V_2)$
soit $\mu = \mathtt{mgu}_{V_3}\{\theta_3(\tau_2) = \tau_3;\ \theta_3(\varphi_2) = \varphi_3;\ \theta_3(\theta_2(\xi(\varphi_1))) = \varphi_3\}$
prendre $\tau = \mu(\tau_3)$ et $\varphi = \mu(\varphi_3)$ et $\theta = \mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1$ et $V' = V_3 \cup \mathrm{Dom}(\mu) \cup \mathrm{Rng}(\mu)$.

• **Si a est C :**
soit $\rho \notin V$ une variable de rangée fraîche de sorte $\mathtt{EXN}(\{C\})$
soit $\rho' \notin V$ une variable de rangée fraîche de sorte $\mathtt{EXN}(\emptyset)$
prendre $\tau = \mathtt{exn}[C : \mathtt{Pre}; \rho]$ et $\varphi = \rho'$ et $\theta = id$ et $V' = V \cup \{\rho, \rho'\}$.

• **Si a est $D(a_1)$:**
soit $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, V_2) = \mathtt{Inst}(\mathit{TypeArg}(D), V_1)$

soit $\mu = \mathtt{mgu}_{V_2}\{\tau_2 = \tau_1\}$
soit $\rho \notin V_2 \cup \mathrm{Dom}\,(\mu)$ **une variable de rangée fraîche de sorte** $\mathtt{EXN}(\{D\})$
prendre $\tau = \mu(\mathtt{exn}[D(\tau_1);\, \rho])$ **et** $\varphi = \mu(\varphi_1)$ **et** $\theta = \mu \circ \theta_1$ **et** $V' = V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu) \cup \{\rho\}$.

• **Si** a **est** $\mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2$ **:**
soit $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E) \oplus \{x : \mathtt{exn}[\varphi_1]\}, a_2, V_1)$
soit $\mu = \mathtt{mgu}_{V_2}\{\theta_2(\tau_1) = \tau_2\}$
prendre $\tau = \mu(\tau_2)$ **et** $\varphi = \mu(\varphi_2)$ **et** $\theta = \mu \circ \theta_2 \circ \theta_1$ **et** $V' = V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$.

Notre algorithme utilise deux fonctions auxiliaires `Inst` et `Patsubtr`. Nous allons maintenant donner leur définition.

La fonction auxiliaire $\mathtt{Inst}(\sigma, V)$ (instance triviale ou instance la plus générale) :
$\mathtt{Inst}(\forall \alpha_i, \rho_j, \delta_k.\, \tau,\ V)$ est $(\tau\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\},\ V \cup \{\alpha'_i, \rho'_j, \delta'_k\})$ où $\alpha'_i, \rho'_j, \delta'_k$ sont des variables fraîches n'appartenant pas à V telles que ρ'_j et ρ_j ont la même sorte pour tout j.

La fonction auxiliaire `Patsubtr` (typage des motifs et soustraction de motifs) :
$\mathtt{Patsubtr}(p, \tau, V)$ est le quadruplet (E, τ', θ, V') défini par induction sur un motif p de la façon suivante :

• **Si** p **est** x **:**
soit $\alpha \notin V$ **une variable de type fraîche**
prendre $E = \{x : \tau\}$ **et** $\tau' = \alpha$ **et** $\theta = id$ **et** $V' = V \cup \{\alpha\}$.

• **Si** p **est** i **:**
soit $\rho \notin V$ **une variable de rangée fraîche de sorte** $\mathtt{INT}(\{i\})$
soit $\mu = \mathtt{mgu}_{V \cup \{\rho\}}\{\tau = \mathtt{int}[i : \mathtt{Pre};\, \rho]\}$
soit $\delta \notin V \cup \{\rho\} \cup \mathrm{Dom}\,(\mu)$ **une variable de présence fraîche**
prendre $E = \emptyset$ **et** $\tau' = \mu(\mathtt{int}[i : \delta;\, \rho])$ **et** $\theta = \mu$ **et** $V' = V \cup Dom(\mu) \cup \{\rho, \delta\}$.

• **Si** p **est** C **:**
soit $\rho \notin V$ **une variable de rangée fraîche de sorte** $\mathtt{EXN}(\{C\})$
soit $\mu = \mathtt{mgu}_{V \cup \{\rho\}}\{\tau = \mathtt{exn}[C : \mathtt{Pre};\, \rho]\}$
soit $\delta \notin V \cup \{\rho\} \cup \mathrm{Dom}\,(\mu)$ **une variable de présence fraîche**
prendre $E = \emptyset$ **et** $\tau' = \mu(\mathtt{exn}[C : \delta;\, \rho])$ **et** $\theta = \mu$ **et** $V' = V \cup \{\rho, \delta\}$.

• **Si** p **est** $D(p_1)$ **:**
soit $(\tau_1, V_0) = \mathtt{Inst}(\mathit{TypeArg}(D))$
soit $(E_1, \tau'_1, \theta_1, V_1) = \mathtt{Patsubtr}(p_1, \tau_1, V)$
soit $\rho \notin V_1$ **une variable de rangée fraîche de sorte** $\mathtt{EXN}(\{D\})$
soit $\mu = \mathtt{mgu}_{V_1 \cup \{\rho\}}\{\theta_1(\tau) = \mathtt{exn}[D(\theta_1(\tau_1));\, \rho]\}$
prendre $E = \mu(E_1)$ **et** $\tau' = \mathtt{exn}[D(\mu(\tau'_1));\, \mu(\rho)]$ **et** $\theta = \mu \circ \theta_1$ **et** $V' = V_1 \cup \{\rho\} \cup \mathrm{Dom}\,(\mu)$.

6.4 Preuves des propriétés de l'inférence de types

Puisque l'algorithme d'inférence W repose fondamentalement sur l'algorithme d'unification, nous allons maintenant démontrer les trois propriétés importantes de cet algorithme, à savoir sa correction, sa complétude puis sa principalité.

Auparavant, nous aurons besoin de montrer que l'exécution de cet algorithme termine et ce pour deux raisons. Tout d'abord, ceci est nécessaire pour assurer que notre analyse termine elle aussi. D'autre part, ceci nous permettra de définir une notion de complexité sur les ensembles d'équations, nous autorisant ainsi à raisonner par récurrence sur l'exécution de l'algorithme pour en établir formellement les propriétés que nous voulons démontrer.

Pour finir, nous démontrerons ces mêmes propriétés pour l'algorithme W lui-même, en nous appuyant sur celles de l'algorithme d'unification.

Dans les théorèmes et lemmes qui vont suivre, nous noterons $\theta_1 =_V \theta_2$ pour dire que $\theta_1(v) = \theta_2(v)$ pour toute variable $v \in V$.

Notons que l'existence d'unificateurs principaux découle du fait que notre théorie équationelle est syntaxique et régulière [46].

6.4.1 Terminaison l'algorithme d'unification

En regardant la description formelle des algorithmes d'unification, nous pouvons nous rendre compte que la taille des ensembles d'équations ne diminue pas forcément en terme de nombre d'équations présentes dans les appels récursifs à $\mathtt{mgu}_V$. Par exemple, $\mathtt{mgu}_V(\{\tau_1 \stackrel{\varphi_1}{\to} \tau_1' = \tau_2 \stackrel{\varphi_2}{\to} \tau_2'\} \cup Q)$ est égal à $\mathtt{mgu}_V(\{\tau_1 = \tau_2; \varphi_1 = \varphi_2; \tau_1' = \tau_2'\} \cup Q)$ cas dans lequel l'appel récursif à $\mathtt{mgu}_V$ est effectué sur un ensemble comportant deux équations supplémentaires.

Ceci nous empêche *a priori* d'utiliser un argument de récurrence simple sur la taille de ces ensembles pour prouver d'une part sa terminaison, et d'autre part ses propriétés de correction et de complétude. Aussi allons-nous maintenant définir une métrique particulière, strictement décroissante lors d'un appel récursif de $\mathtt{mgu}_V$, qui permette d'une part de prouver la terminaison de l'algorithme et d'autre part d'appliquer des hypothèses de récurrence entre ses appels.

Rappels sur les ordres bien fondés

Nous commençons par donner quelques rappels de définitions et propriétés à propos des ordres bien fondés et des multi-ensembles. Nous ne démontrons rien ici, ceci sortant du cadre de cette thèse, mais nous voulons donner au lecteur non familier avec ces notions, une explication simple des outils que nous allons utiliser pour prouver la terminaison de notre algorithme.

Définition 6.1 (Multi-ensemble) *Un multi-ensemble M est un ensemble dans lequel des éléments peuvent apparaître plusieurs fois. Plus formellement, si D est un ensemble, l'ensemble $\mathcal{M}(D)$ des multi-ensembles finis d'éléments de D est formé des applications de D dans $\mathbb{N}$ qui sont nulles sauf pour un nombre fini d'éléments de D.*

Définition 6.2 (Ordre bien fondé sur les multi-ensembles) *Soit $<$ une relation d'ordre bien fondée sur un ensemble E. On étend cette relation (que l'on note alors $<_M$) à $\mathcal{M}(E)$ (l'ensemble des multi-ensembles finis d'éléments de E) par : $M' <_M M$ si M' est obtenu*

- *soit en supprimant un élément x de M,*
- *soit en remplaçant une occurrence d'un élément x de M par un multi-ensemble d'éléments x_i' tous plus petits que x (c'est-à -dire tels que pour tout i, $x_i' < x$).*

Propriété 6.1 *On peut démontrer que $<_M$ est bien fondé sur $\mathcal{M}(E)$ si et seulement si $<$ est bien fondé sur E.*

Définition 6.3 (Extension lexicographique d'un ordre fondé) *Soit $<_1$ un ordre bien fondé sur un ensemble E_1 et $<_2$ un ordre bien fondé sur un ensemble E_2. L'ordre lexicographique $<_{\mathtt{lex}}$ est défini sur $E_1 \times E_2$ par $(x_1, x_2) <_{\mathtt{lex}} (y_1, y_2)$:*

- *si $x_1 < y_1$,*
- *ou si $x_1 = y_1$ et $x_2 < y_2$.*

Propriété 6.2 *On peut démontrer que* $<_{\texttt{lex}}$ *est bien fondé sur* $E_1 \times E_2$ *si et seulement si* $<_1$ *et* $<_2$ *sont respectivement bien fondés sur* E_1 *et* E_2.

Lemmes auxiliaires

Nous commençons par montrer deux lemmes qui vont nous servir à prouver la terminaison de notre algorithme d'unification.

Lemme 6.1 (Les sortes sont préservées par réécriture) *Soit* Q *un ensemble d'équations bien sortées. Si l'algorithme* $\texttt{mgu}_V$ *se rappelle récursivement sur* Q', *alors toute équation de* Q' *est bien sortée.*

Preuve : La preuve est une simple récurrence sur l'exécution de l'algorithme `mgu`. Seuls quelques cas ne sont pas résolus trivialement, cas que nous examinons maintenant.

Cas $\texttt{mgu}_V(\{\rho = \varphi\} \cup Q)$ et $\texttt{mgu}_V(\{\varphi = \rho\} \cup Q)$.
Dans ces deux cas, l'algorithme se rappelle sur l'ensemble $Q\{\rho \leftarrow \varphi\}$. Puisque par hypothèse notre ensemble d'équations est bien sorté, nous savons que la sorte de ρ est égale à celle de φ. Ainsi, dans $Q\{\rho \leftarrow \varphi\}$ remplacer ρ par la rangée φ ne change pas les sortes des équations (lemme 5.11).

Cas $\texttt{mgu}_V(\{(i : \pi;\, \varphi) = \top\} \cup Q)$ et $\texttt{mgu}_V(\{\top = (i : \pi;\, \varphi)\} \cup Q)$.
Dans ces deux cas, l'algorithme se rappelle sur l'ensemble $\{\pi = \texttt{Pre};\, \varphi = \top\} \cup Q)$. Par hypothèse notre ensemble initial est bien formé. Donc Q l'est également. Il reste à montrer que $\{\pi = \texttt{Pre};\, \varphi = \top\}$ aussi. Sa première partie ($\pi = \texttt{Pre}$) l'est trivialement puisqu'elle ne fait intervenir aucune sorte. Quant à $\varphi = \top$, puisque $\top$ possède toutes les sortes, nous lui attribuons la même sorte que φ et nous concluons.

Cas $\texttt{mgu}_V(\{(\epsilon_1;\, \varphi_1) = (\epsilon_2;\, \varphi_2)\} \cup Q)$ si $H(\epsilon_1) = H(\epsilon_2)$.
Dans ce cas, l'algorithme se rappelle sur l'ensemble $\{\epsilon_1 = \epsilon_2;\, \varphi_1 = \varphi_2\} \cup Q$. Puisque notre ensemble initial est bien sorté, $(\epsilon_1;\, \varphi_1)$ et $(\epsilon_2;\, \varphi_2)$ sont de même sorte S. Ainsi φ_1 et φ_2 sont toutes deux de sorte $H(\epsilon_1) \cup S$, et donc l'équation $\varphi_1 = \varphi_2$ est bien sortée. De plus, par récurrence, les types présents dans ϵ_1 et ϵ_2 étant bien sortés, l'équation $\epsilon_1 = \epsilon_2$ l'est également, ce qui nous permet de conclure au bon sortage, dans sa globalité, de l'ensemble récursivement utilisé par l'algorithme.

Cas $\texttt{mgu}_V(\{(\epsilon_1;\, \varphi_1) = (\epsilon_2;\, \varphi_2)\} \cup Q)$ si $H(\epsilon_1) \neq H(\epsilon_2)$.
Dans ce dernier cas, l'algorithme se rappelle sur l'ensemble $\{\varphi_1 = (\epsilon_2;\, \rho);\, \varphi_2 = (\epsilon_1;\, \rho)\} \cup Q$. Soit S la sorte des deux rangées $(\epsilon_1;\, \varphi_1)$ et $(\epsilon_2;\, \varphi_2)$. Par construction, nous savons que $\rho :: \{S \cup \{H(\epsilon_1), H(\epsilon_2)\}$. Ainsi $\varphi_1 :: S \cup \{(\epsilon_1\}$, de même pour $(\epsilon_2;\, \rho)$, ce qui confère un sortage correct pour la première équation de l'ensemble récursivement utilisé. De plus, $\varphi_2 :: S \cup \{(\epsilon_2\}$ de même que pour $(\epsilon_1;\, \rho)$, donc la seconde équation est également bien sortée. Puisque notre ensemble initial était bien sorté, il en est de même pour Q, ce qui nous permet de conclure au bon sortage de tout l'ensemble sur lequel l'algorithme $\texttt{mgu}_V$ se rappelle récursivement dans ce cas.

□

Lemme 6.2 (La réécriture ne génère pas de nouveaux constructeurs de sorte)

Soit Q *un ensemble d'équations bien sortées. Si l'algorithme* $\texttt{mgu}_V$ *se rappelle récursivement sur* Q', *alors l'ensemble des constructeurs de sortes nécessaires pour sorter* Q' *est inclus dans celui nécessaire pour sorter* Q.

Preuve : Cette preuve est également effectuée par simple récurrence sur l'exécution de l'algorithme $\texttt{mgu}_V$. Il suffit alors d'analyser chaque règle de l'algorithme d'unification pour s'apercevoir que les ensembles utilisés récursivement sont constitués de rangées bâties avec des éléments de rangée présents dans l'ensemble de départ. □

Terminaison de l'algorithme $\mathtt{mgu}_V$

> **Théorème 6.1 (Unification : terminaison)** *Soit Q un ensemble d'équations bien sortées et V un ensemble fini de variables, alors l'algorithme $\mathtt{mgu}_V$ termine (avec succès ou échec) lorsqu'il est appliqué à l'ensemble d'équations Q.*

Preuve : Nous définissons respectivement la taille d'un terme de type τ, d'une rangée φ et d'une annotation de présence π et les notons respectivement $\mid \tau \mid$, $\mid \varphi \mid$ et $\mid \pi \mid$ de la façon suivante :

$$\begin{array}{rcl} \mid \alpha \mid & = & 1 \\ \mid \mathtt{int}[\varphi] \mid = \mid \mathtt{exn}[\varphi] \mid & = & 1 + \mid \varphi \mid \\ \mid \tau_1 \stackrel{\varphi}{\rightarrow} \tau_2 \mid & = & 1 + \mid \tau_1 \mid + \mid \tau_2 \mid + \mid \varphi \mid \end{array}$$

$$\begin{array}{rcl} \mid \rho \mid = \mid \top \mid & = & 1 \\ \mid \epsilon ; \varphi \mid & = & \mid \epsilon \mid + \mid \varphi \mid \end{array}$$

$$\begin{array}{rcl} \mid i : \pi \mid = \mid C : \pi \mid & = & 1 + \mid \pi \mid \\ \mid D(\tau) \mid & = & 1 + \mid \tau \mid \end{array}$$

$$\mid \mathtt{Pre} \mid = \mid \delta \mid \ = \ 1$$

Soit Q un ensemble fini composé d'équations bien formées $\{t_1 = t'_1; \ldots t_m = t'_m\}$ où les t_i représentent aussi bien des types τ que des rangées φ ou des annotations de présence π. Soit $SK_{\mathtt{max}}$ la taille de la sorte de longueur maximale des types présents dans nos équations. L'existence de cette longueur maximale découle du fait que les types, donc les sortes sont en nombre fixe et donné dans notre ensemble d'équations qui est fini et que la réécriture de nos ensembles d'équations préserve ces sortes (lemme 6.1) et n'en créé pas de nouvelles (lemme 6.2).

Les sortes étant des ensembles finis, nous appelons « taille » le cardinal de cet ensemble. Nous définissons maintenant les trois fonctions d'interprétation suivantes :

- $\Phi_1(Q)$ est le multi-ensemble composé des éléments $SK_{\mathtt{max}} - \mid \kappa_E \mid$ pour toute équation E apparaissant dans Q, où nous notons κ_E la sorte de l'équation E et $\mid \kappa_E \mid$ la taille de cette sorte.
- $\Phi_2(Q)$ est le nombre de variables présentes dans le système (variables de type, de rangée ou de présence).
- $\Phi_3(Q)$ est le multi-ensemble $\{m_1; \ldots; m_n\}$ où $m_i = \mathtt{max}(\mid t_i \mid, \mid t'_i \mid)$ pour chaque équation $\{t_i = t'_i\}$ de l'ensemble Q.

Nous définissons alors $\Phi(Q)$ comme le triplet $(\Phi_1(Q), \Phi_2(Q), \Phi_3(Q))$. Nous munissons alors l'ensemble de ces triplets de la composée lexicographique des trois ordres suivants :

- Pour Φ_1 : l'extension multi-ensemble de l'ordre habituel sur les entiers naturels,
- Pour Φ_2 : l'ordre habituel sur les entiers naturels,
- Pour Φ_3 : l'extension multi-ensemble de l'ordre habituel sur les entiers naturels.

L'ordre obtenu est alors bien fondé puisqu'il est construit à partir d'extensions multi-ensemble et lexicographique d'ordres bien fondés. Nous devons maintenant montrer que si notre algorithme $\mathtt{mgu}_V$, appliqué à un système Q, se rappelle récursivement sur un ensemble Q', alors $Q' < Q$. Nous allons donc étudier le sens de variation de la fonction d'interprétation Φ, c'est-à-dire des trois fonctions Φ_1, Φ_2 et Φ_3 pour chaque règle de l'algorithme. Nous synthétisons ces résultats dans les tableaux récapitulatifs suivants :

Cas initial		Appel récursif	Φ_1	Φ_2	Φ_3	Note
$\{\alpha = \alpha\} \cup Q$	$\to$	Q	$=$	$\leq$	$<$	
$\{\alpha = \tau\} \cup Q$	$\to$	$Q\{\alpha \leftarrow \tau\}$	$=$	$<$		car $\alpha \notin \tau$
$\{\tau = \alpha\} \cup Q$	$\to$	$Q\{\alpha \leftarrow \tau\}$	$=$	$<$		car $\alpha \notin \tau$
$\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q$	$\to$	$\{\varphi_1 = \varphi_2\} \cup Q$	$=$	$\leq$	$<$	
$\{\mathtt{exn}[\varphi_1] = \mathtt{exn}[\varphi_2]\} \cup Q$	$\to$	$\{\varphi_1 = \varphi_2\} \cup Q$	$=$	$\leq$	$<$	
$\{\tau_1 \xrightarrow{\varphi_1} \tau_1' = \tau_2 \xrightarrow{\varphi_2} \tau_2'\} \cup Q$	$\to$	$\{\tau_1 = \tau_2; \varphi_1 = \varphi_2; \tau_1' = \tau_2'\} \cup Q$	$=$	$\leq$	$<$	

Cas initial		Appel récursif	Φ_1	Φ_2	Φ_3	Note
$\{\rho = \rho\} \cup Q$	$\to$	Q	$\leq$	$\leq$	$<$	
$\{\rho = \varphi\} \cup Q$	$\to$	$Q\{\rho \leftarrow \varphi\})$	$\leq$	$<$		car $\rho \notin \varphi$
$\{\varphi = \rho\} \cup Q$	$\to$	$Q\{\rho \leftarrow \varphi\}$	$\leq$	$<$		car $\rho \notin \varphi$
$\{\top = \top\} \cup Q$	$\to$	Q	$=$	$=$	$<$	
$\{(i : \pi;\ \varphi) = \top\} \cup Q$	$\to$	$\{\pi = \mathtt{Pre};\ \varphi = \top\} \cup Q$	$<$			
$\{\top = (i : \pi;\ \varphi)\} \cup Q$	$\to$	$\{\pi = \mathtt{Pre};\ \varphi = \top\} \cup Q$	$<$			
$\{(\epsilon_1;\ \varphi_1) = (\epsilon_2;\ \varphi_2)\} \cup Q)$ si $H(\epsilon_1) = H(\epsilon_2)$	$\to$	$\{\epsilon_1 = \epsilon_2;\ \varphi_1 = \varphi_2\} \cup Q$	$<$			
$\{(\epsilon_1;\ \varphi_1) = (\epsilon_2;\ \varphi_2)\} \cup Q$ si $H(\epsilon_1) \neq H(\epsilon_2)$	$\to$	$\{\varphi_1 = (\epsilon_2;\ \rho);\ \varphi_2 = (\epsilon_1;\ \rho)\} \cup Q$	$<$			

Afin de clarifier les quatres derniers cas, donnons une explication quant à la décroissance de leur critère Φ_1. Pour les deux premiers, dans l'ensemble d'équations initial, appelons S la sorte de $(i : \pi;\ \varphi) = \top$ (rappelons que $\top$ possède toutes les sortes). Lors de l'appel récursif, l'ensemble comporte maintenant l'équation $\{\varphi = \top\}$ pour laquelle φ est alors de sorte (par la règle de sortage (3) vue en section 4.1) $S \cup \{i\}$, dont la taille est supérieure à la précédente ce qui cause bien une décroissance de notre fonction d'interprétation Φ_1.

Le troisième cas s'ensuit pour la même raison, S étant la sorte de $(\epsilon_1;\ \varphi_1) = (\epsilon_2;\ \varphi_2)$, dans l'appel récursif, $\varphi_1 = \varphi_2$ a donc comme sorte $S \cup \{H(\epsilon_1)\}$ ou $S \cup \{H(\epsilon_2)\}$ puisque $H(\epsilon_1) = H(\epsilon_2)$.

Enfin le dernier cas est encore traité similairement. Soit S la sorte de $(\epsilon_1;\ \varphi_1) = (\epsilon_2;\ \varphi_2)$. Nous savons que ρ est de sorte $S \cup H(\epsilon_1) \cup H(\epsilon_2)$. Donc nous déduisons que $\varphi_1 = (\epsilon_2;\ \rho) :: S \cup H(\epsilon_1)$ et $\varphi_2 = (\epsilon_1;\ \rho) :: S \cup H(\epsilon_2)$ et concluons comme les cas précédents.

Cas initial		Appel récursif	Φ_1	Φ_2	Φ_3
$\{(i : \pi_1) = (i : \pi_2)\} \cup Q$	$\to$	$\{\pi_1 = \pi_2\} \cup Q$	$=$	$=$	$<$
$\{(C : \pi_1) = (C : \pi_2)\} \cup Q$	$\to$	$\{\pi_1 = \pi_2\} \cup Q$	$=$	$=$	$<$
$\{D(\tau_1) = D(\tau_2)\} \cup Q$	$\to$	$\{\tau_1 = \tau_2\} \cup Q$	$=$	$=$	$<$

Cas initial		Appel récursif	Φ_1	Φ_2	Φ_3	Note
$\{\delta = \pi\} \cup Q$	$\to$	$Q\{\delta \leftarrow \pi\}$	$=$	$<$		car $\delta \notin \pi$
$\{\pi = \delta\} \cup Q$	$\to$	$Q\{\delta \leftarrow \pi\}$	$=$	$<$		car $\delta \notin \pi$
$\{\mathtt{Pre} = \mathtt{Pre}\} \cup Q$	$\to$	Q	$=$	$=$	$<$	

□

6.4.2 Correction de l'algorithme d'unification

Nous commençons cette section par l'énoncé d'un résultat sur la composition de substitutions préservant les sortes. Nous ne donnons pas ici la preuve de ce résultat, celle-ci pouvant être trouvée dans [45].

Lemme 6.3 (Composition de substitutions préservant les sortes) *Soit θ_1 et θ_2 deux substitutions préservant les sortes. Alors $\theta_2 \circ \theta_1$ préserve également les sortes.*

Théorème 6.2 (Unification : correction) *Soit Q un ensemble d'équations bien formées et V un ensemble fini de variables.*

Si $\theta = \mathtt{mgu}_V(Q)$ est défini, alors θ est un unificateur de Q (autrement dit, $\theta(\tau_1) = \theta(\tau_2)$ pour toute équation $\{\tau_1 = \tau_2\}$ de Q), et θ préserve les sortes, et $\mathrm{Dom}\,(\theta) \cap V \subseteq FV(Q)$, *et* $\mathrm{Rng}\,(\theta) \cap V \subseteq FV(Q)$.

Preuve : La preuve est effectuée par raisonnement inductif sur l'exécution de $\mathtt{mgu}_V$ et par cas sur la forme de l'ensemble d'équations Q. Nous remarquerons que de nombreux cas présentent une structure similaire (au niveau des types, comme au niveau des rangées ou des annotations de présence). Afin de ne pas allonger inutilement la démonstration de ce théorème, nous nous autoriserons à traiter exhaustivement le premier d'entre eux et à y faire ensuite simplement référence quand les cas suivants sont similaires.

Cas $\mathtt{mgu}_V(\emptyset)$.

Dans ce cas, la substitution θ renvoyée par l'algorithme est l'identité et celle-ci satisfait trivialement nos exigences.

Cas $\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q)$.

Dans ce cas, la substitution θ renvoyée par l'algorithme est de la forme $\theta = \mathtt{mgu}_V(Q)$. Par hypothèse de récurrence, θ vérifie nos exigences sur Q (préservation des sortes et propriétés sur Dom et Rng de θ).

Ensuite, trivialement on a bien que $\theta(\alpha) = \theta(\alpha)$ ainsi que préservation des sortes et des propriétés sur Dom et Rng de θ pour $\{\alpha = \alpha\} \cup Q$, ce qui nous permet de conclure.

Cas $\mathtt{mgu}(\{\alpha = \tau\} \cup Q)$ si α n'apparaît pas dans τ.

Dans ce cas, la substitution θ renvoyée par l'algorithme est de la forme $\theta = \mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}$. Appelons θ' la substitution $\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\})$.

Il faut donc prouver que :

$$\theta(\alpha) = \theta(\tau) \tag{6.1}$$

$$\forall \{\tau_1 = \tau_2\} \in Q, \theta(\tau_1) = \theta(\tau_2) \tag{6.2}$$

$$\theta \text{ préserve les sortes} \tag{6.3}$$

$$\mathrm{Dom}\,(\theta) \cap V \subseteq FV(Q) \tag{6.4}$$

$$\mathrm{Rng}\,(\theta) \cap V \subseteq FV(Q) \tag{6.5}$$

- Pour 6.1, nous devons montrer que $\theta(\alpha) = \theta(\tau)$,
  c'est-à-dire que $(\theta' \circ \{\alpha \leftarrow \tau\})(\alpha) = (\theta' \circ \{\alpha \leftarrow \tau\})(\tau)$,
  ou encore que $\theta'(\tau) = \theta' \circ \{\alpha \leftarrow \tau\}(\tau)$.
  Comme α n'apparaît pas dans τ, dans le membre droit de l'égalité attendue, la substitution $\{\alpha \leftarrow \tau\}$ n'a pas d'effet sur τ. Ainsi ce membre peut être réécrit en $\theta'(\tau)$. De ce fait, il nous reste à montrer que $\theta'(\tau) = \theta'(\tau)$, ce qui est trivialement vrai.
- Pour 6.2, par récurrence, puisque θ' est la substitution retournée par $\mathtt{mgu}_V$ sur $Q\{\alpha \leftarrow \tau\}$ et que cette substitution est justement un unificateur de chacune des équations de cet ensemble, nous savons que pour toute équation $\{\tau_1' = \tau_2'\}$ dans $Q\{\alpha \leftarrow \tau\}$ on a :

  $$\theta'(\tau_1') = \theta'(\tau_2') \tag{6.6}$$

  Donc, pour toute équation $\{\tau_1 = \tau_2\}$ dans Q on a que : $\{\tau_1\{\alpha \leftarrow \tau\} = \tau_2\{\alpha \leftarrow \tau\}\}$ est une équation de $Q\{\alpha \leftarrow \tau\}$, et par (6.6), nous déduisons donc que :

  $$\theta'(\tau_1\{\alpha \leftarrow \tau\}) = \theta'(\tau_2\{\alpha \leftarrow \tau\})$$
  $$\underbrace{(\theta' \circ \{\alpha \leftarrow \tau\})}_{=\theta}(\tau_1) = \underbrace{(\theta' \circ \{\alpha \leftarrow \tau\})}_{=\theta}(\tau_2)$$

  Or comme nous l'avons notée précédemment, la substitution θ est égale à $\theta' \circ \{\alpha \leftarrow \tau\}$, donc nous avons bien que pour toute équation $\{\tau_1 = \tau_2\}$ dans Q, $\{\theta(\tau_1) = \theta(\tau_2)\}$.

- Pour 6.3, il faut donc montrer que $\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}$ préserve les sortes. Par hypothèse de récurrence, nous savons que $\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\})$ les préserve. Nous devons maintenant montrer que $\{\alpha \leftarrow \tau\}$ les préserve également.
  Puisque l'ensemble d'équations considéré est bien sorté (voir définition en section 6.2) par hypothèse de notre théorème, nous pouvons déduire que $\{\alpha \leftarrow \tau\}$ préserve effectivement les sortes (c'est-à-dire que pour toute variable de type β égale ou différente de α, on a $(\{\alpha \leftarrow \tau\})(\beta)$ est bien formé et que –trivialement puisque la substitution ne fait pas intervenir de variable de rangée– pour toute variable de rangée ρ, on a $\vdash \{\alpha \leftarrow \tau\})(\rho) :: K(\rho))$.
  Ainsi chacune des deux parties de θ préservant les sortes, leur composition, (donc θ) préserve effectivement les sortes (par le lemme 6.3).
- Pour 6.4, nous voulons montrer que :

$$\mathrm{Dom}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}) \cap V \subseteq FV(\{\alpha = \tau\} \cup Q) \tag{6.7}$$

  Par hypothèse de récurrence, nous savons que :

$$\mathrm{Dom}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\})) \cap V \subseteq FV(Q\{\alpha \leftarrow \tau\})$$

  Or $FV(Q\{\alpha \leftarrow \tau\}) \subseteq FV(Q) \cup FV(\tau)$, puisque $\alpha \notin FV(\tau)$. Donc :

$$\mathrm{Dom}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\})) \cap V \subseteq FV(Q) \cup FV(\tau)$$

  De plus $\mathrm{Dom}\,(\{\alpha \leftarrow \tau\}) = \{\alpha\}$. Donc nous avons :

$$\mathrm{Dom}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}) \cap V \subseteq FV(Q) \cup FV(\tau) \cup \{\alpha\}$$

  Et comme $FV(\{\alpha = \tau\} \cup Q)$ est bien égal à $FV(Q) \cup FV(\tau) \cup \{\alpha\}$, nous avons bien démontré le but attendu en 6.7.
- Pour 6.5, nous voulons montrer que :

$$\mathrm{Rng}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}) \cap V \subseteq FV(\{\alpha = \tau\} \cup Q) \tag{6.8}$$

  Par hypothèse de récurrence, nous savons que :

$$\mathrm{Rng}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\})) \cap V \subseteq FV(Q\{\alpha \leftarrow \tau\}) \subseteq FV(Q) \cup FV(\tau)$$

  De plus $\mathrm{Rng}\,(\{\alpha \leftarrow \tau\}) = FV(\tau)$. Nous avons donc :

$$\mathrm{Rng}\,(\mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}) \cap V \subseteq FV(Q) \cup FV(\tau)$$

  Comme $FV(\{\alpha = \tau\} \cup Q)$ est égal à $FV(Q) \cup FV(\tau) \cup \{\alpha\}$ et que $FV(Q) \cup FV(\tau) \subseteq FV(Q) \cup FV(\tau) \cup \{\alpha\}$, nous déduisons que $FV(Q) \cup FV(\tau) \subseteq FV(\{\alpha = \tau\} \cup Q)$, ce qui démontre bien notre but 6.8.

Nous avons donc prouvé que notre substitution θ satisfaisait nos cinq conditions requises, ce qui clos la preuve dans le cas présent.

Cas $\mathtt{mgu}(\{\tau = \alpha\} \cup Q)$ si α n'apparaît pas dans τ.
On se convaincra aisément que ce cas se démontre de la même façon que le précédent.

Cas $\mathtt{mgu}(\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q)$, $\mathtt{mgu}(\{\mathtt{exn}[\varphi_1] = \mathtt{exn}[\varphi_2]\} \cup Q)$ et $\mathtt{mgu}(\{\tau_1 \xrightarrow{\varphi_1} \tau_1' = \tau_2 \xrightarrow{\varphi_2} \tau_2'\} \cup Q)$.
L'algorithme se rappelant récursivement, nous pouvons directement appliquer l'hypothèse de récurrence pour conclure.

Cas $\mathtt{mgu}_V(\{\rho = \rho\} \cup Q)$.
Ce cas est traité similairement au cas $\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q)$.

Cas $\mathtt{mgu}_V(\{\rho = \varphi\} \cup Q)$ et $\mathtt{mgu}_V(\{\varphi = \rho\} \cup Q)$.

Ces cas sont traités similairement au cas $\mathtt{mgu}_V(\{\alpha = \tau\} \cup Q)$.

Cas $\mathtt{mgu}_V(\{\top = \top\} \cup Q)$.
Ce cas s'ensuit directement de l'hypothèse de récurrence, l'algorithme se rappelant récursivement.

Cas $\mathtt{mgu}_V(\{(i:\pi;\ \varphi) = \top\} \cup Q)$, $\mathtt{mgu}_V(\{\top = (i:\pi;\ \varphi)\} \cup Q)$ et $\mathtt{mgu}_V(\{(\epsilon_1;\ \varphi_1) = (\epsilon_2;\ \varphi_2)\} \cup Q)$ si $H(\epsilon_1) = H(\epsilon_2)$.
Comme ci-dessus ou dans le cas $\mathtt{mgu}(\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q)$, ces deux cas se traitent par récurrence.

Cas $\mathtt{mgu}_V(\{(\epsilon_1;\ \varphi_1) = (\epsilon_2;\ \varphi_2)\} \cup Q_1)$ si $H(\epsilon_1) \neq H(\epsilon_2)$.
Notons $Q' = \{\varphi_1 = (\epsilon_2;\ \rho);\ \varphi_2 = (\epsilon_1;\ \rho)\} \cup Q_1$. Supposons que $\theta = \mathtt{mgu}_{V \cup \{\rho\}}(Q')$ soit définie. Par hypothèse de récurrence, θ est une solution de Q', préserve les sortes, et $(\mathrm{Dom}\,(\theta) \cup \mathrm{Rng}\,(\theta)) \cap (V \cup \{\rho\}) \subseteq FV(Q')$. Puisque θ est une solution de Q', c'est également une solution de Q_1. De plus,

$$\begin{array}{rclcl} \theta(\epsilon_1;\ \varphi_1) & = & \theta(\epsilon_1);\ \theta(\varphi_1) & = & \theta(\epsilon_1);\ \theta(\epsilon_2);\ \theta(\rho) \\ \theta(\epsilon_2;\varphi_2) & = & \theta(\epsilon_2);\ \theta(\varphi_2) & = & \theta(\epsilon_2);\ \theta(\epsilon_1);\ \theta(\rho) \end{array}$$

Ces deux rangées sont égales modulo l'équation (4.2) vue en section 4.1.8. Ainsi, θ est une solution de Q.
Finalement, par hypothèse de récurrence, nous avons $(\mathrm{Dom}\,(\theta) \cup \mathrm{Rng}\,(\theta)) \cap \{V \cup \{\rho\}\} \subseteq FV(Q')$. Puisque $FV(Q') = FV(Q) \cup \{\rho\}$, il s'ensuit que $(\mathrm{Dom}\,(\theta) \cup \mathrm{Rng}\,(\theta)) \cap V \subseteq FV(Q)$ comme attendu.

Cas $\mathtt{mgu}_V(\{(i:\pi_1) = (i:\pi_2)\} \cup Q)$, $\mathtt{mgu}_V(\{(C:\pi_1) = (C:\pi_2)\} \cup Q)$ et $\mathtt{mgu}_V(\{D(\tau_1) = D(\tau_2)\} \cup Q)$.
L'algorithme se rappelant récursivement, on applique simplement l'hypothèse de récurrence.

Cas $\mathtt{mgu}_V(\{\delta = \pi\} \cup Q)$ et $\mathtt{mgu}_V(\{\pi = \delta\} \cup Q)$.
Ces deux cas sont traités similairement au cas $\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q)$.

Cas $\{\mathtt{Pre} = \mathtt{Pre}\} \cup Q)$.
Il suffit de faire usage de l'hypothèse de récurrence pour conclure.

□

6.4.3 Complétude et principalité de l'algorithme d'unification

Théorème 6.3 (Unification : complétude et principalité) *Soit Q un ensemble d'équations bien formées et V un ensemble fini de variables.*
Si ψ préserve les sortes et est une solution de Q, et si $FV(Q) \subseteq V$, alors $\theta = \mathtt{mgu}_V(Q)$ est définie et il existe une substitution préservant les sortes ξ telle que $\psi =_V \xi \circ \theta$.

Preuve : La preuve est effectuée par une récurrence sur l'ordre $\Phi(Q)$ et une analyse par cas sur la forme de Q.

Cas $Q = \emptyset$.
L'ensemble d'équations étant vide, $\theta = id$ et donc $\xi = \psi$ convient.

Cas $Q = \{\alpha = \alpha\} \cup Q'$.
Soit ψ solution de Q qui préserve les sortes. Nous en déduisons que :

$$\psi(\alpha) = \psi(\alpha) \tag{6.1}$$

$$\psi \text{ est solution de } Q' \tag{6.2}$$

Nous voulons prouver que :

$$\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q') \text{ est défini} \tag{6.3}$$
$$\exists \xi \text{ préservant les sortes telle que } \psi =_V \xi \circ \mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q') \tag{6.4}$$

L'ensemble d'équations Q' étant de complexité inférieure (au sens défini précédemment) à l'ensemble $\{\alpha = \alpha\} \cup Q'$, nous pouvons lui appliquer l'hypothèse de récurrence avec ψ et déduire que $\mathtt{mgu}_V(Q')$ est défini. Ceci implique, par définition de l'algorithme $\mathtt{mgu}_V$ dans le cas présent, que $\mathtt{mgu}_V(Q)$ (c'est-à-dire $\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q')$) l'est également (comme attendu en 6.3).

Cherchons maintenant à établir 6.4. Puisque par l'hypothèse 6.2 ψ est solution de Q', elle peut s'écrire sous la forme $\psi = \xi \circ \mathtt{mgu}_V(Q')$. Or par définition de l'algorithme $\mathtt{mgu}_V$, $\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q') = \mathtt{mgu}_V(Q')$, donc nous déduisons que $\psi = \xi \circ \mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q')$ ce qui était attendu en 6.4.

Cas $Q = \{\alpha = \tau\} \cup Q'$ avec $\tau \neq \alpha$.

Soit ψ solution de Q qui préserve les sortes. Nous en déduisons que :

$$\psi(\alpha) = \psi(\tau) \tag{6.1}$$
$$\psi \text{ est solution de } Q' \tag{6.2}$$

Nous voulons prouver que :

$$\mathtt{mgu}_V(\{\alpha = \tau\} \cup Q') \text{ est défini} \tag{6.3}$$
$$\exists \xi \text{ préservant les sortes telle que } \psi =_V \xi \circ \mathtt{mgu}_V(\{\alpha = \tau\} \cup Q') \tag{6.4}$$

Dans ce cas, l'unificateur retourné par l'algorithme $\mathtt{mgu}_V$ est bien défini et de la forme $\theta = \mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}$ seulement si α n'apparaît pas dans τ. Nous devons donc préalablement prouver ce point. Si α apparaît dans τ, alors α est un sous-terme strict de τ (à moins que $\tau = \alpha$, or nous sommes explicitement pas dans ce cas là). Donc $\psi(\alpha)$ est un sous-terme strict de $\psi(\tau)$ impliquant que $\psi(\alpha)$ est différent de $\psi(\tau)$, ce qui est en contradiction avec notre hypothèse 6.1. Donc effectivement, α n'apparaît pas dans τ ce qui nous permet d'établir la première partie de notre théorème (c'est-à-dire 6.3) dans le cas présent.

Cherchons maintenant à en établir la seconde partie (c'est-à-dire 6.4). L'idée est d'écrire ψ sous la forme $\psi' \circ \{\alpha \leftarrow \tau\}$, de montrer que ψ' est alors solution de $Q'\{\alpha \leftarrow \tau\}$, ce qui nous permettra alors d'appliquer l'hypothèse de récurrence sur ψ' et $Q'\{\alpha \leftarrow \tau\}$ (en effet, l'ensemble $Q'\{\alpha \leftarrow \tau\}$ est de complexité inférieure à l'ensemble $\{\alpha = \tau\} \cup Q'$ au sens défini précédemment). Nous en déduirons donc que ψ' peut s'écrire sous la forme $\psi' = \xi \circ \mathtt{mgu}_V(Q'\{\alpha \leftarrow \tau\})$, et donc que $\psi = \xi \circ \mathtt{mgu}_V(Q'\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\}$. Et enfin, comme par définition de l'algorithme $\mathtt{mgu}_V$, $\mathtt{mgu}_V(Q'\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\} = \mathtt{mgu}_V(\{\alpha = \tau\} \cup Q')$, ceci nous amènera à conclure que $\psi = \xi \circ \mathtt{mgu}_V(\{\alpha = \tau\} \cup Q')$ ce qui est le but attendu en (6.4).

Cherchons donc à écrire ψ sous la forme $\psi' \circ \{\alpha \leftarrow \tau\}$ pour une certaine ψ'.

Prenons ψ' telle que : $\begin{cases} \beta \mapsto \psi(\beta) & \text{pour } \beta \neq \alpha \\ \alpha \mapsto \alpha \end{cases}$

Ainsi, $\psi' \circ \{\alpha \leftarrow \tau\}$ est telle que :

- $\alpha \mapsto \tau \mapsto \psi'(\tau) = \psi(\tau)$ car α n'apparaît pas dans τ.
- $\beta \mapsto \beta \mapsto \psi'(\beta) = \psi(\beta)$ (pour β différente de α)

Donc on a bien $\psi = \psi' \circ \{\alpha \leftarrow \tau\}$.

Prouvons maintenant que ψ' est solution de $Q'\{\alpha \leftarrow \tau\}$. Soit $\{\tau_1 = \tau_2\}$ une équation de Q'. Alors $\{\tau_1\{\alpha \leftarrow \tau\} = \tau_2\{\alpha \leftarrow \tau\}\}$ est une équation de $Q'\{\alpha \leftarrow \tau\}$. Si ψ est solution de Q', alors $\psi(\tau_1) = \psi(\tau_2)$, donc $\psi(\tau_1\{\alpha \leftarrow \tau\}) = \psi(\tau_2\{\alpha \leftarrow \tau\})$.

- $\psi(\tau_1\{\alpha \leftarrow \tau\}) = (\psi \circ \{\alpha \leftarrow \tau\})(\tau_1) = \psi'(\tau_1)$ par la définition de ψ' (cf. ci-dessus)
- $\psi(\tau_2\{\alpha \leftarrow \tau\}) = (\psi \circ \{\alpha \leftarrow \tau\})(\tau_2) = \psi'(\tau_2)$

Or par l'hypothèse 6.2, puisque ψ est solution de Q' et que $\{\tau_1 = \tau_2\}$ est une équation de Q', on a bien $\psi'(\tau_1) = \psi'(\tau_2)$, ce qui nous amène le résultat attendu.
Nous pouvons donc maintenant suivre le chemin de preuve énoncé ci-dessus pour conclure.

Cas $Q = \{\tau = \alpha\} \cup Q'$.
On se convaincra aisément que ce cas se réduit exactement au précédent.

Cas $Q = \{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q'$.
Soit ψ solution de Q qui préserve les sortes. Nous en déduisons que :

$$\psi(\mathtt{int}[\varphi_1]) = \psi(\mathtt{int}[\varphi_2]) \tag{6.1}$$
$$\psi \text{ est solution de } Q' \tag{6.2}$$

Nous voulons prouver que :

$$\mathtt{mgu}_V(\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q') \text{ est défini} \tag{6.3}$$
$$\exists\, \xi \text{ préservant les sortes telle que } \psi =_V \xi \circ \mathtt{mgu}_V(\{\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q') \tag{6.4}$$

Dans ce cas, l'unificateur retourné par l'algorithme $\mathtt{mgu}_V$ est bien défini et de la forme $\theta = \mathtt{mgu}_V(\{\varphi_1 = \varphi_2\} \cup Q')$ ce qui nous permet d'établir la première partie de notre théorème (c'est-à-dire 6.3).

Cherchons maintenant à en établir la seconde partie (c'est-à-dire 6.4). Par définition de l'algorithme $\mathtt{mgu}_V$, $\mathtt{mgu}_V(\{\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q')$ est égal à $\mathtt{mgu}_V(\{\{\varphi_1 = \varphi_2\} \cup Q')$. L'ensemble d'équations $\{\varphi_1 = \varphi_2\} \cup Q'$ étant de complexité inférieure (au sens défini précédemment) à l'ensemble $\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q'$, nous pouvons lui appliquer l'hypothèse de récurrence avec ψ pour déduire directement notre but 6.4 et ainsi conclure le présent cas de la preuve.

Cas $Q = \{\mathtt{exn}[\varphi_1] = \mathtt{exn}[\varphi_2]\} \cup Q'$.
On se convaincra aisément que ce cas est traité similairement au précédent en remplaçant dans la preuve le constructeur de type `int` par `exn`.

Cas $Q = \{\tau_1 \xrightarrow{\varphi_1} \tau_1' = \tau_2 \xrightarrow{\varphi_2} \tau_2'\} \cup Q'$.
Ce cas est traité simplement par l'utilisation de l'hypothèse de récurrence, l'ensemble d'équations $\{\tau_1 = \tau_2; \varphi_1 = \varphi_2; \tau_1' = \tau_2'\} \cup Q'$ étant de complexité inférieure (au sens défini précédemment) à l'ensemble Q initial . Possédant une structure de problème similaire au cas $Q = \{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q'$, afin de ne pas allonger cette preuve, nous nous autorisons à demander au lecteur de s'y référer.

Cas $Q = \{\rho = \rho\} \cup Q'$.
Ce cas se traite comme le cas $Q = \{\alpha = \alpha\} \cup Q'$.

Cas $Q = \{\rho = \varphi\} \cup Q'$ et $Q = \{\varphi = \rho\} \cup Q'$.
Ces cas sont traités comme le cas $Q = \{\alpha = \tau\} \cup Q'$.

Cas $Q = \{\top = \top\} \cup Q'$.
Puisque par définition de l'algorithme $\mathtt{mgu}_V$, $\mathtt{mgu}_V(\{\top = \top\} \cup Q')$ est égal à $\mathtt{mgu}_V(Q')$, nous pouvons directement faire usage de l'hypothèse de récurrence pour démontrer ce cas (l'ensemble d'équations Q' étant de complexité inférieure (au sens défini précédemment) à l'ensemble $\{\top = \top\} \cup Q'$).

Cas $Q = \{(i : \pi;\, \varphi) = \top\} \cup Q'$, $Q = \{\top = (i : \pi;\, \varphi)\} \cup Q'$ et $Q = \{(\epsilon_1;\, \varphi_1) = (\epsilon_2;\, \varphi_2)\} \cup Q'$ si $H(\epsilon_1) = H(\epsilon_2)$.
Ces cas sont traités similairement à $Q = \{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q'$.

Cas $Q = \{(\epsilon_1;\, \varphi_1) = (\epsilon_2;\, \varphi_2)\} \cup Q_1$ si $H(\epsilon_1) \neq H(\epsilon_2)$.
Soit ψ solution de Q qui préserve les sortes. Nous en déduisons que :

$$\psi(\epsilon_1;\, \varphi_1) = \psi(\epsilon_2;\, \varphi_2) \tag{6.5}$$
$$\psi \text{ est solution de } Q_1 \tag{6.6}$$

Dans ce cas, l'unificateur retourné par l'algorithme $\mathtt{mgu}_V$ est bien défini et de la forme $\theta = \mathtt{mgu}_{V \cup \{\rho\}}(\{\varphi_1 = (\epsilon_2;\, \rho);\, \varphi_2 = (\epsilon_1;\, \rho)\} \cup Q_1)$ ce qui nous permet d'établir la première partie de notre théorème (c'est-à-dire 6.3). Cherchons maintenant à en établir la seconde partie (c'est-à-dire 6.4).

Soit $t(S)$ la sorte de $(\epsilon_1;\, \varphi_1)$ et de $(\epsilon_2;\, \varphi_2)$ et ρ une variable de rangée telle que $\rho \notin V$ et soit de sorte $t(S \cup \{H(\epsilon_1);\, H(\epsilon_2)\})$. Enfin, notons $Q' = \{\varphi_1 = (\epsilon_2;\, \rho);\, \varphi_2 = (\epsilon_1;\, \rho)\} \cup Q_1$. Nous remarquons que les équations présentes dans Q' sont bien formées puisque la première est de la sorte $t(S \cup \{H(\epsilon_1)\})$ et la seconde de la sorte $t(S \cup \{H(\epsilon_2)\})$).

Puisque par l'hypothèse 6.5 $\psi(\epsilon_1;\, \varphi_1)$ et $\psi(\epsilon_2;\, \varphi_2)$, sont égales, elles sont toutes deux égales à une rangée de la forme $(\psi(\epsilon_1);\, \psi(\epsilon_2);\, \varphi)$ pour une certaine rangée φ.

Prenons $\psi' = \psi \oplus \{\rho \leftarrow \varphi\}$. Par construction de la rangée φ, la substitution ψ' est une solution de Q'. Par application de l'hypothèse de récurrence, $\theta = \mathtt{mgu}_{V \cup \{\rho\}}(Q')$ est définie, et $\psi' =_{V \cup \{\rho\}} \xi \circ \theta$ pour une certaine substitution ξ.

Ainsi $\mathtt{mgu}_V(Q) = \theta$ est définie, et puisque $\psi' =_V \psi$, nous avons bien $\psi =_V \xi \circ \theta$ comme attendu.

Cas $Q = \{(i : \pi_1) = (i : \pi_2)\} \cup Q'$, $Q = \{(C : \pi_1) = (C : \pi_2)\} \cup Q'$ et $Q = \{D(\tau_1) = D(\tau_2)\} \cup Q'$.

Ces cas se traitent comme le cas $Q = \{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q'$.

Cas $Q = \{\delta = \pi\} \cup Q'$ et $\{\pi = \delta\} \cup Q'$.

Ces deux cas sont traités comme $Q = \{\alpha = \tau\} \cup Q'$.

Cas $Q = \{\mathtt{Pre} = \mathtt{Pre}\} \cup Q'$.

On se convaincra aisément que ce cas est démontré de manière similaire au cas $Q = \{\top = \top\} \cup Q'$.

□

6.4.4 Correction de l'algorithme d'inférence W

Lemme 6.4 (Propriétés structurelles de W) *Supposons que $(\tau, \varphi, \theta, V') = W(E, a, V)$ soit défini, V soit fini, et $FV(E) \subseteq V$. Alors :*

- *$V \subseteq V'$;*
- *V' est fini ;*
- *$FV(\tau) \subseteq V'$ et $FV(\varphi) \subseteq V'$;*
- *toutes les variables non présentes dans V' sont hors de portée de θ ;*
- *$FV(\theta(E)) \subseteq V'$;*
- *θ préserve les sortes.*

Preuve : La preuve est simplement effectuée par examen des cas de l'algorithme W en conjonction avec les propriétés de $\mathtt{mgu}_V$ sur Dom et Rng démontrées dans le théorème 6.2.

Cas a est x.

Ce cas est traité en examinant la fonction auxiliaire `Inst`. Notons $E(x)$ sous la forme $\forall \alpha_i, \beta_j, \delta_k.\tau'$. Nous avons alors que :

$$\tau = \tau'\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\} \tag{6.1}$$
$$V' = V\{\rho\} \cup \{\alpha'_i, \beta'_j, \delta'_k\} \tag{6.2}$$
$$\theta = id \tag{6.3}$$
$$\varphi = \rho \tag{6.4}$$

- Montrons que $V \subseteq V'$, c'est-à-dire par 6.2 que $V \subseteq V \cup \{\rho\} \cup \{\alpha'_i, \beta'_j, \delta'_k\}$. Ceci est trivialement vérifié.
- V étant fini par hypothèse, V' (étant égal à $V \cup \{\rho\} \cup \{\alpha'_i, \beta'_j, \delta'_k\}$ par 6.2) l'est également.

- Montrons que $FV(\tau) \subseteq V'$, c'est-à-dire par 6.1 et 6.2 que $FV(\tau'\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\}) \subseteq V\{\rho\} \cup \{\alpha'_i, \beta'_j, \delta'_k\}$ Nous savons par hypothèse que τ' appartient à E. Puisque par hypothèse $FV(E) \subseteq V$, nous avons que $FV(\tau'\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\}) \subseteq FV(E) \cup \{\alpha'_i, \beta'_j, \delta'_k\}$. Donc par 6.1, nous déduisons que $FV(\tau) \subseteq V \cup \{\alpha'_i, \beta'_j, \delta'_k\}$, et par extension que $FV(\tau) \subseteq V \cup \{\rho\} \cup \{\alpha'_i, \beta'_j, \delta'_k\}$, comme attendu.
  Montrons que $FV(\varphi) = V'$, c'est-à-dire par 6.2 et 6.4 que $FV(\rho) = V\{\rho\} \cup \{\alpha'_i, \beta'_j, \delta'_k\}$. Ceci est trivialement vérifié.
- Par 6.3, θ étant l'identité, toutes les variables non présentes dans V' sont trivialement hors de portée de θ.
- Montrons que $FV(\theta(E)) \subseteq V'$. Puisque θ est l'identité, que par hypothèse $FV(E) \subseteq V$ nous avons que $FV(\theta(E)) \subseteq V$. De plus, par hypothèse $V \subseteq V'$, donc nous avons bien que $FV(\theta(E)) \subseteq V'$.
- θ préserve les sortes trivialement puisque c'est l'identité.

Cas a est i.
Par définition de W, nous avons :

$$\tau = \texttt{int}[i : \texttt{Pre}; \rho] \tag{6.1}$$
$$V' = V \cup \{\rho, \rho'\} \tag{6.2}$$
$$\theta = id \tag{6.3}$$
$$\varphi = \rho \tag{6.4}$$

- Montrons que $V \subseteq V'$, c'est-à-dire par 6.2 que $V \subseteq V \cup \{\rho, \rho'\}$, ce qui est trivialement vérifié.
- V étant fini, $V' = V \cup \{\rho, \rho'\}$ est fini.
- Montrons que $FV(\tau) \subseteq V'$ c'est-à-dire par 6.1 et 6.2 que $FV(\texttt{int}[i : \texttt{Pre}; \rho]) \subseteq V \cup \{\rho, \rho'\}$. Comme $FV(\texttt{int}[i : \texttt{Pre}; \rho]) = \{\rho\}$, ceci est trivialement vérifié.
  Montrons que $FV(\varphi) \subseteq V'$, c'est-à-dire par 6.2 et 6.4 que $FV(\rho) \subseteq V \cup \{\rho, \rho'\}$ ce qui est trivialement vérifié.
- Par 6.3, θ étant l'identité, toutes les variables non présentes dans V' sont trivialement hors de portée de θ.
- Puisque par 6.3 θ est l'identité, que par hypothèse $FV(E) \subseteq V$ et que par 6.2 $V \subseteq V'$, alors nous avons bien $FV(\theta(E)) \subseteq V'$;
- Par 6.3, étant l'identité, θ préserve trivialement les sortes.

Cas a est $\lambda x.a_1$.
Par définition de W, nous avons :

$$\tau = \theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1 \tag{6.1}$$
$$V' = V_1 \cup \{\rho\} \tag{6.2}$$
$$\theta = \theta_1 \tag{6.3}$$
$$\varphi = \rho \tag{6.4}$$
$$\tag{6.5}$$

Par application de l'hypothèse de récurrence sur la seconde étape de l'algorithme W, nous déduisons que $(\tau_1, \varphi_1, \theta_1, V_1)$ satisfont les présentes propriétés structurelles de W à savoir :

$$V \cup \{\alpha\} \subseteq V_1 \tag{6.6}$$
$$V_1 \text{ est fini} \tag{6.7}$$
$$FV(\tau_1) \subseteq V_1 \text{ et } FV(\varphi_1) \subseteq V_1 \tag{6.8}$$
$$\text{toutes les variables non présentes dans } V_1 \text{ sont hors de portée de } \theta_1 \tag{6.9}$$
$$FV(\theta_1(E \oplus \{x : \alpha\})) \subseteq V_1 \tag{6.10}$$
$$\theta_1 \text{ préserve les sortes} \tag{6.11}$$

- Montrons que $V \subseteq V'$, c'est-à-dire par 6.2, que $V \subseteq V_1 \cup \{\rho\}$. Par 6.6, nous avons $V \cup \{\alpha\} \subseteq V_1$, donc $V \subseteq V_1$, donc $V \subseteq V_1 \cup \{\rho\}$ comme attendu.
- Par 6.7, V_1 est fini d'où V' (étant égal par 6.2 à $V_1 \cup \{\rho\}$) l'est également.
- Montrons que $FV(\tau) \subseteq V'$ c'est-à-dire par 6.1 et 6.2, que $FV(\theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1) \subseteq V_1 \cup \{\rho\}$. De 6.10 nous déduisons que $FV(\theta_1(\alpha)) \subseteq V_1$. Comme $FV(\theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1) = FV(\theta_1(\alpha)) \cup FV(\varphi_1) \cup FV(\tau_1)$, de la déduction précédente et de l'hypothèse 6.8 nous avons $FV(\theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1) \subseteq V_1$ et par extension $FV(\theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1) \subseteq V_1\{\rho\}$ comme attendu.
  Montrons que $FV(\varphi) \subseteq V'$, c'est-à-dire par 6.2 et 6.4 que $FV(\rho) \subseteq V_1 \cup \{\rho\}$ ce qui est trivial puisque $FV(\rho) = \{\rho\}$.
- De 6.9 et 6.3 nous déduisons toutes les variables non présentes dans V_1 sont hors de portée de θ. N'ayant pas créé de nouvelle variable, à part ρ qui se trouve dans V', nous avons bien que toutes les variables non présentes dans V' (égal par 6.2 à $V_1 \cup \{\rho\}$) sont hors de portée de θ.
- Montrons que $FV(\theta(E)) \subseteq V'$, c'est-à-dire, par 6.2, que $FV(\theta(E)) \subseteq V_1 \cup \{\rho\}$. Par 6.3 et 6.10, nous déduisons que $FV(\theta(E \oplus \{x : \alpha\})) \subseteq V_1$, donc que $FV(\theta(E)) \subseteq V_1$, et par extension que $FV(\theta(E)) \subseteq V_1 \cup \{\rho\}$ comme attendu.
- θ préserve les sortes puisque par 6.3 $\theta = \theta_1$ et que par 6.11 θ_1 les préserve.

Cas a est $a_1(a_2)$.

Par définition de W, nous avons :

$$\tau = \mu(\alpha) \tag{6.1}$$
$$V' = V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu) \tag{6.2}$$
$$\theta = \mu \circ \theta_2 \circ \theta_1 \tag{6.3}$$
$$\varphi = \mu(\varphi_2) \tag{6.4}$$

Par application de l'hypothèse de récurrence sur les première et seconde étapes de l'algorithme W, nous déduisons que $(\tau_1, \varphi_1, \theta_1, V_1)$ et $(\tau_2, \varphi_2, \theta_2, V_2)$ satisfont les présentes propriétés structurelles de W à savoir :

$$V \subseteq V_1 \tag{6.5}$$
$$V_1 \text{ est fini} \tag{6.6}$$
$$FV(\tau_1) \subseteq V_1 \text{ et } FV(\varphi_1) \subseteq V_1 \tag{6.7}$$
$$\text{toutes les variables non présentes dans } V_1 \text{ sont hors de portée de } \theta_1 \tag{6.8}$$
$$FV(\theta_1(E)) \subseteq V_1 \tag{6.9}$$
$$\theta_1 \text{ préserve les sortes} \tag{6.10}$$

$$V_1 \subseteq V_2 \tag{6.11}$$
$$V_2 \text{ est fini} \tag{6.12}$$
$$FV(\tau_2) \subseteq V_2 \text{ et } FV(\varphi_2) \subseteq V_2 \tag{6.13}$$
$$\text{toutes les variables non présentes dans } V_2 \text{ sont hors de portée de } \theta_2 \tag{6.14}$$
$$FV(\theta_2 \circ \theta_1(E)) \subseteq V_2 \tag{6.15}$$
$$\theta_2 \text{ préserve les sortes} \tag{6.16}$$

- Montrons que $V \subseteq V'$, c'est-à-dire par 6.1 que $V \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. Par 6.5 et 6.11, nous avons $V \subseteq V_2$. Donc par extension, nous avons bien $V \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$.
- Par 6.1, $V' = V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. Puisque V_2 est fini (par 6.12), V' est également fini.

- Montrons que $FV(\tau) \subseteq V'$, c'est-à-dire par les hypothèses 6.1 et 6.2 que $FV(\mu(\alpha)) \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. La substitution μ peut ne pas affecter α, auquel cas, $\alpha \in FV(\mu(\alpha))$. Si par contre μ affecte α alors $FV(\mu(\alpha)) \subseteq \mathrm{Rng}\,(\mu)$. Ainsi, nous avons que $FV(\mu(\alpha)) \subseteq \{\alpha\} \cup \mathrm{Rng}\,(\mu)$ et par extension, nous avons également que $FV(\mu(\alpha)) \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ comme attendu.
  Montrons que $FV(\varphi) \subseteq V'$, c'est-à-dire par les hypothèses 6.2 et 6.4 que $FV(\mu(\varphi_2)) \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. De la même façon que ci-dessus, la substitution μ peut ne pas modifier φ_2, auquel cas $FV(\mu(\varphi_2)) = FV(\varphi_2) \subseteq V_2$ par 6.13. Dans le cas contraire, $FV(\mu(\varphi_2)) \subseteq \mathrm{Rng}\,(\mu)$. Ainsi nous avons que $FV(\mu(\varphi_2)) \subseteq V_2 \cup \mathrm{Rng}\,(\mu)$ et donc par extension que $FV(\mu(\varphi_2)) \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ comme attendu.
- Vérifions que toutes les variables non présentes dans $V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ sont hors de portée de $\mu \circ \theta_2 \circ \theta_1$. Soit v une variable. Si $v \notin V_2$, alors par 6.14 v est hors de portée de θ_2. De plus $v \notin V_1$ puisque par 6.11 $V_1 \subseteq V_2$. Donc v est également hors de portée de θ_1. Si de plus $v \notin \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ alors v est hors de portée de μ. Ainsi, par le lemme 5.8, si $v \notin V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ alors v est hors de portée de $\mu \circ \theta_2 \circ \theta_1$. Et par extension, si $v \notin V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$, alors v est bien hors de portée de $\mu \circ \theta_2 \circ \theta_1$ comme nous l'attendions.
- Montrons que $FV(\theta(E)) \subseteq V'$, c'est-à-dire par 6.2 et par définition de θ dans l'algorithme que $FV(\mu \circ \theta_2 \circ \theta_1(E)) \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. Nous savons par 6.15 que $FV(\theta_2 \circ \theta_1(E)) \subseteq V_2$. Donc nous avons $FV(\mu \circ \theta_2 \circ \theta_1(E)) \subseteq V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ et par extension nous avons bien que $FV(\mu \circ \theta_2 \circ \theta_1(E)) \subseteq V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$.
- Par 6.3 $\theta = \mu \circ \theta_2 \circ \theta_1$. Par 6.10 et 6.16 θ_1 et θ_2 les préserve. De plus, μ étant un unificateur principal, par invariants structurels du théorème 6.2, nous savons qu'elle préserve également les sortes. La composition de substitutions préservant les sortes étant une substitution qui préserve les sortes, nous avons bien que θ les préserve.

Cas a est `let` $x = a_1$ `in` a_2.

Par définition de W, nous avons :

$$\tau = \mu(\tau_2) \tag{6.1}$$
$$V' = V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu) \tag{6.2}$$
$$\theta = \mu \circ \theta_2 \circ \theta_1 \tag{6.3}$$
$$\varphi = \mu(\varphi_2) \tag{6.4}$$

Par application de l'hypothèse de récurrence sur la première étape de l'algorithme W, nous déduisons que $(\tau_1, \varphi_1, \theta_1, V_1)$ satisfont les présentes propriétés structurelles de W à savoir :

$$V \subseteq V_1 \tag{6.5}$$
$$V_1 \text{ est fini} \tag{6.6}$$
$$FV(\tau_1) \subseteq V_1 \text{ et } FV(\varphi_1) \subseteq V_1 \tag{6.7}$$
$$\text{toutes les variables non présentes dans } V_1 \text{ sont hors de portée de } \theta_1 \tag{6.8}$$
$$FV(\theta_1(E)) \subseteq V_1 \tag{6.9}$$
$$\theta_1 \text{ préserve les sortes} \tag{6.10}$$

Appliquons maintenant l'hypothèse de réccurrence sur la seconde étape de l'algorithme W. Nous déduisons alors que $(\tau_2, \varphi_2, \theta_2, V_2)$ satisfont les présentes propriétés structurelles de W à savoir :

$$V_1 \subseteq V_2 \tag{6.11}$$
$$V_2 \text{ est fini} \tag{6.12}$$
$$FV(\tau_2) \subseteq V_2 \text{ et } FV(\varphi_2) \subseteq V_2 \tag{6.13}$$
$$\text{toutes les variables non présentes dans } V_2 \text{ sont hors de portée de } \theta_2 \tag{6.14}$$
$$FV(\theta_2(\theta_1(E) \oplus \{x : \mathrm{Gen}\,(\tau_1, \theta_1(E), \varphi_1)\}) \subseteq V_2 \tag{6.15}$$
$$\theta_2 \text{ préserve les sortes} \tag{6.16}$$

- Montrons que $V \subseteq V'$, c'est-à-dire par 6.2 que $V \subseteq V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. Par 6.5 et 6.11 nous avons $V \subseteq V_1 \subseteq V_2$, donc par extension, nous avons bien $V \subseteq V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$.
- Par 6.12, V_2 étant fini, $\mathrm{Dom}\,(\mu)$ et $\mathrm{Rng}\,(\mu)$ l'étant également, nous avons bien que $V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ (c'est-à-dire V' par 6.2) est fini.
- Montrons que $FV(\tau) \subseteq V'$, c'est-à-dire par les hypothèses 6.1 et 6.2, que $FV(\mu(\tau_2)) \subseteq V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. Par la déduction 6.13, nous savons que $FV(\tau_2) \subseteq V_2$. Ainsi $FV(\mu(\tau_2))$ est bien inclu dans $V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ comme attendu.
- Montrons que $FV(\varphi) \subseteq V'$, c'est-à-dire par les hypothèses 6.4 et 6.2, que $FV(\mu(\varphi_2)) \subseteq V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$, ce qui est vérifié puisque par 6.13, nous savons que $FV(\varphi_2) \subseteq V_2$.
- Vérifions que toutes les variables non présentes dans $V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ sont hors de portée de $\mu \circ \theta_2 \circ \theta_1$. Soit v une variable. Si $v \notin V_2$, alors par 6.14 v est hors de portée de θ_2. De plus $v \notin V_1$ puisque par 6.10 $V_1 \subseteq V_2$. Donc v est également hors de portée de θ_1. Si de plus $v \notin \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ alors v est hors de portée de μ. Ainsi, par le lemme 5.8, si $v \notin V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$ alors v est hors de portée de $\mu \circ \theta_2 \circ \theta_1$, comme nous l'attendions.
- Montrons que $FV(\theta(E)) \subseteq V'$, c'est-à-dire par 6.2 et par définition de θ dans l'algorithme que $FV(\mu \circ \theta_2 \circ \theta_1(E)) \subseteq V_2 \cup \mathrm{Dom}\,(\mu) \cup \mathrm{Rng}\,(\mu)$. De 6.15, par le lemme de commutation entre généralisation et substitution (lemme 5.9), nous déduisons que $FV(\theta_2 \circ \theta_1(E) \oplus \{x : \mathrm{Gen}\,(\theta_2(\tau_1), \theta_2 \circ \theta_1(E), \theta_2(\varphi_1))\}) \subseteq V_2$.

Cas a est `match` a_1 `with` $p \to a_2 \mid x \to a_3$.

Cas a est C.

On se convaincra aisément que ce cas est identique au cas a est i, en remplaçant le constructeur de type `int` par `exn` et nous nous autorisons à ne pas en donner ici la preuve *in extenso* afin de ne pas rallonger inutilement cette démonstration.

Cas a est $D(a_1)$.

Cas a est `try` a_1 `with` $x \to a_2$.

□

Théorème 6.4 (Correction de `Patsubtr`) *Si $(E, \tau', \theta, V') = Patsubtr(p, \tau, V)$ est défini, alors :*
- $\vdash p : \theta(\tau) \Rightarrow E$
- $\vdash \theta(\tau) - p \rightsquigarrow \tau'$

Preuve : La preuve est effectuée par cas sur l'algorithme de la fonction auxiliaire `Patsubtr`.

Cas p est x.

Hypothèses :

$$E = \{x : \tau\} \tag{6.1}$$
$$\tau' = \alpha \tag{6.2}$$
$$\theta = id \tag{6.3}$$

Nous voulons montrer que : $x : id(\tau) \Rightarrow \{x : \tau\}$ et $id(\tau) - p \leadsto \alpha$, c'est-à-dire que :

$$\vdash x : \tau \Rightarrow \{x : \tau\} \tag{6.4}$$
$$\vdash \tau - p \leadsto \alpha \tag{6.5}$$

Le but 6.4 est directement fourni par la règle de typage 19 des motifs. Pour prouver 6.5, nous voulons utiliser une règle de soustraction 25, donc nous devons en établir la prémisse, à savoir que α bf. Or ceci est trivialement vérifié, ce qui clos ce cas de la preuve.

Cas p est i.
Hypothèses :

$$E = \emptyset \tag{6.1}$$
$$\mu = \mathtt{mgu}_{V \cup \{\rho\}}\{\tau = \mathtt{int}[i : \mathtt{Pre};\ \rho]\} \tag{6.2}$$
$$\tau' = \mathtt{int}[i : \delta;\ \mu(\rho)] \tag{6.3}$$
$$\theta = \mu \tag{6.4}$$
$$\tag{6.5}$$

Nous voulons montrer que :

$$i : \mu(\tau) \Rightarrow \emptyset \tag{6.6}$$
$$\mu(\tau) - i \leadsto \mathtt{int}[i : \delta;\ \mu(\rho)] \tag{6.7}$$

Pour prouver le but 6.6, nous voulons utiliser une règle 20. Nous devons donc montrer que nous pouvons écrire $\mu(\tau)$ sous la forme $\mathtt{int}[i : \mathtt{Pre};\ \varphi]$ pour un certain φ. Or nous savons par l'hypothèse 6.2 que μ est un unificateur de $\{\tau = \mathtt{int}[i : \mathtt{Pre};\ \rho]\}$. Donc nous en déduisons que $\mu(\tau) = \mu(\mathtt{int}[i : \mathtt{Pre};\ \rho])$, c'est-à-dire que :

$$\mu(\tau) = \mathtt{int}[i : \mathtt{Pre};\ \mu(\rho)] \tag{6.8}$$

Prenons alors φ égal à $\mu(\rho)$ et nous avons bien réussi à écrire $\mu(\tau)$ sous la forme désirée ce qui établit bien notre but initial 6.6.

Nous devons, pour terminer ce cas de la preuve, démontrer la seconde propriété, à savoir 6.7. Pour y parvenir, nous désirons utiliser une règle de soustraction 23. Nous devons donc montrer à nouveau qu'il est possible d'écrire $\mu(\tau)$ sous la forme $\mathtt{int}[i : \mathtt{Pre};\ \mu(\rho)]$, ce qui a déjà été démontré ci-dessus et clos ainsi ce cas de la preuve.

Cas p est C.
On se convaincra aisément que ce cas est identique au précédent, en remplaçant le constructeur de type `int` par `exn`.

Cas p est $D(p_1)$.
Hypothèses :

$$E = \mu(E_1) \tag{6.1}$$
$$(E_1, \tau_1', \theta_1, V_1) = \mathtt{Patsubtr}(p_1, \tau_1, V) \tag{6.2}$$
$$\mu = \mathtt{mgu}_{V_1 \cup \{\rho\}}\{\theta_1(\tau) = \mathtt{exn}[D(\theta_1(\tau_1)); \rho]\} \tag{6.3}$$
$$\tau' = \mathtt{exn}[D(\mu(\tau_1'));\ \mu(\rho)] \tag{6.4}$$
$$\theta = \mu \circ \theta_1 \tag{6.5}$$

Nous voulons montrer que :

$$D(p_1) : \mu \circ \theta_1(\tau) \Rightarrow \mu(E_1) \tag{6.6}$$
$$\mu \circ \theta_1(\tau) - D(p_1) \leadsto \mathtt{exn}[D(\mu(\tau_1'));\ \mu(\rho)] \tag{6.7}$$

Appliquons l'hypothèse de récurrence à 6.2, il nous vient que :

$$\vdash p_1 : \theta_1(\tau_1) \Rightarrow E_1 \tag{6.8}$$
$$\vdash \theta_1(\tau_1) - p_1 \rightsquigarrow \tau_1' \tag{6.9}$$

Pour prouver 6.6, nous voulons utiliser une règle de typage des motifs 22. Nous devons donc d'abord vérifier que $\mu \circ \theta_1(\tau)$ peut s'écrire sous forme d'un type exception. Puisque par 6.3, μ est un unificateur de $\{\theta_1(\tau) = \mathtt{exn}[D(\theta_1(\tau_1));\, \rho]\}$, nous savons que $\mu(\theta_1(\tau)) = \mu(\mathtt{exn}[D(\theta_1(\tau_1));\, \rho])$, c'est-à-dire que $\mu(\theta_1(\tau)) = \mathtt{exn}[D(\mu(\theta_1(\tau_1)));\, \mu(\rho)]$. Donc nous pouvons en effet écrire $\mu \circ \theta_1(\tau)$ sous forme d'un type exception. Il nous reste donc à montrer les prémisses de notre règle 22 en les instanciant en tenant compte de la forme de $\mu \circ \theta_1(\tau)$. Ainsi nous devons montrer que :

$$\mu(\theta_1(\tau_1)) < \mu(\theta_1(TypeArg(D)) \tag{6.10}$$
$$p : \mu(\theta_1(\tau_1)) \Rightarrow \mu(E_1) \tag{6.11}$$

Nous obtenons 6.10 trivialement par définition de $<$ et de $TypeArg$. Quant à 6.11, nous l'obtenons par application du lemme de stabilité du typage par substitution (partie 1) à 6.8 avec la substitution μ, ce qui nous permet de conclure notre but attendu en 6.6.

Nous nous intéressons maintenant à démontrer 6.9. Pour y parvenir, nous voulons utiliser une dérivation de soustraction de motif 26. Nous avons vu que nous pouvions écrire $\mu \circ \theta_1(\tau)$ sous forme d'un type exception $\mathtt{exn}[D(\mu(\theta_1(\tau_1)));\, \mu(\rho)]$. Nous allons donc montrer la prémisse de notre règle en tenant compte de cette forme pour le type duquel on soustrait. Nous devons donc montrer que :

$$\mu(\theta_1(\tau_1)) - p_1 \rightsquigarrow \mu(\tau_1') \tag{6.12}$$

qui est directement obtenu en appliquant le lemme de stabilité du typage par substitution (partie 2) à 6.9 avec la substitution μ. Ainsi, munis de ces deux prémisses, nous pouvons effectivement reconstruire une dérivation 26 qui établit bien notre but initial 6.7 et clos ainsi complètement cette preuve.

□

Théorème 6.5 (Correction de l'algorithme W) *Si $(\tau, \varphi, \theta, V') = W(E, a, V)$ est défini, alors $\theta(E) \vdash a : \tau/\varphi$. De plus, nous avons $\vdash \tau$ bf et $\vdash \varphi :: \mathtt{EXN}(\emptyset)$, et θ préserve les sortes.*

Preuve : La preuve est effectuée par récurrence sur la forme des expressions a, en utilisant la stabilité des jugements de typage par substitution (lemme 5.12) et la correction de $\mathtt{mgu}_V$ (lemme 6.2). Notons que $\vdash \tau$ bf et $\vdash \varphi :: \mathtt{EXN}(\emptyset)$ découle de $\theta(E) \vdash a : \tau/\varphi$ par le lemme 5.5 (énonçant que les règles de typage des expressions sont bien sortées).

Cas a est x.

Hypothèses :

$$W(E, x, V) = (\mathtt{Inst}(E(x), V \cup \{\rho\}), \rho, id, V \cup \{\rho\}) \tag{6.1}$$
$$\rho \notin V \text{ et } \rho :: \mathtt{EXN}(\emptyset) \tag{6.2}$$

On veut dériver :

$$id(E) \vdash x : \mathtt{Inst}(E(x), V \cup \{\rho\})/\rho \tag{6.3}$$

Ceci revient trivialement à chercher à dériver, $E \vdash x : \mathtt{Inst}(E(x), V \cup \{\rho\})/\rho$. Pour y parvenir, nous voulons utiliser une règle 10. Donc nous devons en établir les prémisses, à savoir :

$$\mathtt{Inst}(E(x), V \cup \{\rho\}) \leq E(x) \tag{6.4}$$
$$\vdash \rho :: \mathtt{EXN}(\emptyset) \tag{6.5}$$

Par définition de $\leq$ et de `Inst` notre prémisse 6.4 est triviale. Quand à la seconde (prémisse 6.5) celle-ci nous est directement fournie par hypothèse 6.2. Ainsi nous pouvons effectivement reconstituer une dérivation 10 et conclure ce cas.

Cas a est i.
Hypothèses :

$$W(E, i, V) = (\mathtt{int}[i : \mathtt{Pre}; \rho], \rho', id, V \cup \{\rho, \rho'\}) \tag{6.1}$$
$$\rho, \rho' \notin V \tag{6.2}$$
$$\rho :: \mathtt{INT}(\{i\}) \text{ et } \rho' :: \mathtt{EXN}(\emptyset) \tag{6.3}$$

On veut dériver :

$$id(E) \vdash i : \mathtt{int}[i : \mathtt{Pre}; \rho]/\rho' \tag{6.4}$$

Ceci revient trivialement à chercher à dériver, $E \vdash i : \mathtt{int}[i : \mathtt{Pre}; \rho]/\rho'$. Pour y parvenir, nous voulons utiliser une règle 11. Donc nous devons en établir les prémisses, à savoir :

$$\vdash \rho' :: \mathtt{INT}(\{i\}) \tag{6.5}$$
$$\vdash \rho :: \mathtt{EXN}(\emptyset) \tag{6.6}$$

Celles-ci nous sont directement fournies par nos hypothèses 6.1 et 6.2, ce qui nous permet de reconstituer effectivement une dérivation 11 et ainsi de conclure ce cas.

Cas a est $\lambda x.a_1$.
Hypothèses :

$$W(E, \lambda x.a_1, V) = (\theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1, \rho, \theta_1, V_1 \cup \{\rho\}) \tag{6.1}$$
$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E \oplus \{x : \alpha\}, a_1, V \cup \{\alpha\}) \tag{6.2}$$
$$\rho \notin V_1 \text{ et } \rho :: \mathtt{EXN}(\emptyset) \tag{6.3}$$

On veut dériver :

$$\theta_1(E) \vdash \lambda x.a_1 : \theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1/\rho \tag{6.4}$$

Pour y parvenir, nous voulons utiliser une règle 12, donc une dérivation de la forme :

$$\frac{\dots \quad \dots \quad \dots}{\theta_1(E) \vdash \lambda x.a_1 : \theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1/\rho}$$

Nous devons donc en établir les trois prémisses, à savoir :

$$\vdash \theta_1(\alpha) \text{ bf} \tag{6.5}$$
$$\theta_1(E) \oplus \{x : \theta_1(\alpha)\} \vdash a_1 : \tau_1/\varphi_1 \tag{6.6}$$
$$\vdash \varphi :: \mathtt{EXN}(\emptyset) \tag{6.7}$$

La prémisse 6.5 nous est donnée par application de l'hypothèse de récurrence. Quant à 6.7, celle-ci fait directement partie de nos hypothèses (6.3). Pour la dernière prémisse restant (6.6), l'utilisation de l'hypothèse de récurrence nous dit que $\theta_1(E \oplus \{x : \alpha\}) \vdash a_1 : \tau_1/\varphi_1$. Or par définition, $\theta_1(E \oplus \{x : \alpha\})$ est égal à $\theta_1(E) \oplus \{x : \theta_1(\alpha)\})$, nous amenant bien à la conclusion 6.6 attendue. Nous pouvons donc effectivement reconstituer une dérivation 12 et ainsi conclure ce cas.

Cas a est $a_1(a_2)$.

Hypothèses :

$$\mu = \mathtt{mgu}_{V_2 \cup \{\alpha\}}\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha, \theta_2(\varphi_1) = \varphi_2\} \tag{6.1}$$
$$W(E, a_1(a_2), V) = (\mu(\alpha), \mu(\varphi_2), \mu \circ \theta_2 \circ \theta_1, V_2 \cup \{\alpha\} \cup \mathrm{Dom}\,(\mu)) \tag{6.2}$$
$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V) \tag{6.3}$$
$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1) \tag{6.4}$$
$$\alpha \notin V_2 \tag{6.5}$$

On veut dériver :

$$\mu \circ \theta_2 \circ \theta_1(E) \vdash a_1(a_2) : \mu(\alpha)/\mu(\varphi_2) \tag{6.6}$$

Pour y parvenir, nous voulons utiliser une règle 13, donc une dérivation de la forme :

$$\frac{\ldots \quad \ldots}{\mu \circ \theta_2 \circ \theta_1(E) \vdash a_1(a_2) : \mu(\alpha)/\mu(\varphi_2)}$$

Nous devons donc en établir les deux prémisses, à savoir :

$$\mu \circ \theta_2 \circ \theta_1(E) \vdash a_1 : ? \xrightarrow{\mu(\varphi)} \mu(\alpha)/\mu(\varphi) \tag{6.7}$$
$$\mu \circ \theta_2 \circ \theta_1(E) \vdash a_2 : ?/\mu(\varphi) \tag{6.8}$$

dans lesquelles nous devrons trouver par quoi instancier les points d'interrogation (?).

Commençons par établir la première de ces prémisses. En appliquant l'hypothèse de récurrence à 6.3, il nous vient que :

$$\theta_1(E) \vdash a_1 : \tau_1/\varphi_1 \tag{6.9}$$

Par application du lemme de stabilité du typage par substitution (lemme 5.12) à 6.9, avec la substitution θ_2 nous déduisons que :

$$\theta_2(\theta_1(E)) \vdash a_1 : \theta_2(\tau_1)/\theta_2(\varphi_1) \tag{6.10}$$

Puisque μ est, par définition de l'algorithme $\mathtt{mgu}_V$, un unificateur de $\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha\}$, nous savons que :

$$\mu(\theta_2(\tau_1)) = \mu(\tau_2 \xrightarrow{\varphi_2} \alpha) \tag{6.11}$$

Appliquons maintenant le lemme de stabilité (lemme 5.12) à 6.10 avec μ, et nous déduisons que :

$$\mu(\theta_2(\theta_1(E))) \vdash a : \mu(\theta_2(\tau_1))/\mu(\theta_2(\varphi_1)) \tag{6.12}$$

Des déductions 6.11 et 6.12 nous déduisons que :

$$\mu(\theta_2(\theta_1(E))) \vdash a : \mu(\tau_2 \xrightarrow{\varphi_2} \alpha)/\mu(\theta_2(\varphi_1)) \tag{6.13}$$

Puisque, par définition de l'algorithme $\mathtt{mgu}_V$, μ est également un unificateur de $\{\theta_2(\varphi_1) = \varphi_2\}$, nous déduisons que :

$$\mu(\theta_2(\varphi_1)) = \mu(\varphi_2) \tag{6.14}$$

Combinons 6.13 et 6.14, nous obtenons :

$$\mu(\theta_2(\theta_1(E))) \vdash a : \mu(\tau_2 \xrightarrow{\varphi_2} \alpha)/\mu(\varphi_2) \tag{6.15}$$

c'est-à-dire

$\mu(\theta_2(\theta_1(E))) \vdash a : \mu(\tau_2) \stackrel{\varphi_2}{\rightarrow} \mu(\alpha))/\mu(\varphi_2)$. Ceci est justement notre but 6.7 dans lequel nous instancions les points d'interrogation par $\mu(\tau_2)$ et nous prenons φ égal à φ_2.

Nous pouvons maintenant établir notre seconde prémisse (6.8) en tenant compte de l'instanciation des points d'interrogation et du choix de φ que nous venons de faire. Nous devons donc maintenant établir que :

$$\mu \circ \theta_2 \circ \theta_1(E) \vdash a_2 : \ \mu(\tau_2)/\mu(\varphi_2) \tag{6.16}$$

En appliquant l'hypothèse de récurrence à 6.4, il nous vient que :

$$\theta_2(\theta_1(E)) \vdash a_2 : \tau_2/\varphi_2 \tag{6.17}$$

Nous appliquons alors simplement le lemme 5.12 à 6.17 avec la substitution μ, et déduisons que $\mu(\theta_2(\theta_1(E))) \vdash a_2 : \mu(\tau_2)/\mu(\varphi_2)$ ce qui était bien le but attendu en 6.16.

Ainsi, munis de ces deux prémisses, nous pouvons effectivement reconstruire une dérivation 13 et clore ce cas de notre preuve.

Cas a est $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$.

Hypothèses :

$$\mu = \mathtt{mgu}_{V_2}\{\theta_2(\varphi_1) = \varphi_2\} \tag{6.1}$$

$$W(E, \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2, V) = (\mu(\tau_2), \mu(\varphi_2), \mu \circ \theta_2 \circ \theta_1, V_2 \cup \mathrm{Dom}\,(\mu)) \tag{6.2}$$

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V_1) \tag{6.3}$$

$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E) \oplus \{x : \mathrm{Gen}\,(\tau_1, \theta_1(E), \varphi_1)\}, a_2, V_2) \tag{6.4}$$

On veut dériver :

$$\mu \circ \theta_2 \circ \theta_1(E) \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \mu(\tau_2)/\mu(\varphi_2) \tag{6.5}$$

Pour y parvenir, nous voulons utiliser une règle 14, donc une dérivation de la forme :

$$\frac{\ldots \quad \ldots}{\mu \circ \theta_2 \circ \theta_1(E) \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \mu(\tau_2)/\mu(\varphi_2)}$$

Nous devons donc en établir les deux prémisses, à savoir :

$$\mu \circ \theta_2 \circ \theta_1(E) \vdash a_1 : ?/\mu(\varphi_2) \tag{6.6}$$

$$\mu \circ \theta_2 \circ \theta_1(E) \oplus \{x : \mathrm{Gen}\,(\,?, \mu \circ \theta_2 \circ \theta_1(E), \mu(\varphi_2))\} \vdash a_2 : \mu(\tau_2)/\mu(\varphi_2) \tag{6.7}$$

dans lesquelles nous devrons trouver par quoi instancier les points d'interrogation (?).

Commençons par établir la première de ces prémisses. En appliquant l'hypothèse de récurrence à 6.3, il nous vient que :

$$\theta_1(E) \vdash a_1 : \tau_1/\varphi_1 \tag{6.8}$$

Par application du lemme de stabilité (lemme 5.12) à 6.8 avec la substitution $\mu \circ \theta_2$, il nous vient que :

$$\mu(\theta_2(\theta_1(E))) \vdash a_1 : \mu(\theta_2(\tau_1)/\mu(\theta_2(\varphi_1) \tag{6.9}$$

Puisque μ est un unificateur de $\{\theta_2(\varphi_1) = \varphi_2\}$, (et qu'il est bien défini pour des raisons similaires à celles énoncées dans le cas précédent), nous savons que :

$$\mu(\theta_2(\varphi_1)) = \mu(\varphi_2) \tag{6.10}$$

Combinant 6.9 avec 6.10, nous déduisons que :

$$\mu(\theta_2(\theta_1(E))) \vdash a_1 : \mu(\theta_2(\tau_1))/\mu(\varphi_2) \tag{6.11}$$

ce qui est notre but 6.6 dans lequel on instancie les points d'interrogation par $\mu(\theta_2(\tau_1))$.

Nous pouvons maintenant établir notre seconde prémisse (6.7) en tenant compte de l'instanciation des points d'interrogation. Nous devons donc maintenant établir que :

$$\mu(\theta_2(\theta_1(E))) \oplus \{x : \mathrm{Gen}\,(\mu(\theta_2(\tau_1)), \mu(\theta_2(\theta_1(E))), \mu(\varphi_2))\} \vdash a_2 :\ \mu(\tau_2)/\mu(\varphi_2) \quad (6.12)$$

En appliquant l'hypothèse de récurrence à 6.4, il nous vient que :

$$\theta_2(\theta_1(E) \oplus \{x : \mathrm{Gen}\,(\tau_1, \theta_1(E), \varphi_1)\}) \vdash a_2 : \tau_2/\varphi_2 \quad (6.13)$$

c'est-à-dire que :

$$\theta_2(\theta_1(E)) \oplus \theta_2(\{x : \mathrm{Gen}\,(\tau_1, \theta_1(E), \varphi_1)\}) \vdash a_2 : \tau_2/\varphi_2 \quad (6.14)$$

En appliquant le lemme de commutation entre généralisation et substitution (lemme 5.10) à 6.14, nous déduisons que :

$$\theta_2(\theta_1(E)) \oplus \{x : \mathrm{Gen}\,(\theta_2(\tau_1), \theta_2(\theta_1(E)), \theta_2(\varphi_1))\} \vdash a_2 : \tau_2/\varphi_2 \quad (6.15)$$

Nous pouvons maintenant appliquer le lemme de stabilité du typage par substitution avec la substitution μ sur 6.15 pour déduire que :

$$\mu(\theta_2(\theta_1(E))) \oplus \mu(\{x : \mathrm{Gen}\,(\theta_2(\tau_1), \theta_2(\theta_1(E)), \theta_2(\varphi_1))\}) \vdash a_2 : \mu(\tau_2)/\mu(\varphi_2) \quad (6.16)$$

Enfin, il nous reste à appliquer à nouveau le lemme de commutation en entre généralisation et substitution à 6.16 pour parvenir à la conclusion 6.12 attendue.

Ainsi, munis des deux prémisses attendues, nous pouvons effectivement reconstruire une dérivation 14 et clore ce cas de notre preuve.

Cas a est `match` a_1 `with` $p \to a_2 \mid x \to a_3$.

Hypothèses :

$$\begin{aligned}
\mu &= \mathtt{mgu}_{V_3}\{\theta_3(\tau_2) = \tau_3, \theta_3(\varphi_2) = \varphi_3, \theta_3(\theta_2(\xi(\varphi_1))) = \varphi_3\} && (6.1)\\
W(E, a, V) &= (\mu(\tau_3), \mu(\varphi_3), \mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1, V_3 \cup \mathrm{Dom}\,(\mu)) && (6.2)\\
(\tau_1, \varphi_1, \theta_1, V_1) &= W(E, a_1, V) && (6.3)\\
(E', \tau', \xi, V_1') &= \mathtt{Patsubtr}(p, \tau_1, V_1) && (6.4)\\
(\tau_2, \varphi_2, \theta_2, V_2) &= W(\xi(\theta_1(E)) \oplus E', a_2, V_1') && (6.5)\\
(\tau_3, \varphi_3, \theta_3, V_3) &= W(\theta_2(\xi(\theta_1(E))) \oplus \{x : \theta_2(\tau')\}, a_3, V_2) && (6.6)
\end{aligned}$$

On veut dériver :

$$\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) \vdash \mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3 : \mu(\tau_3)/\mu(\varphi_3) \quad (6.7)$$

Pour y parvenir, nous voulons utiliser une règle 15, donc une dérivation de la forme :

$$\frac{\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots}{\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) \vdash \mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3 : \mu(\tau_3)/\mu(\varphi_3)}$$

Nous devons donc en établir les cinq prémisses, à savoir :

$$\begin{aligned}
\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) &\vdash a_1 : ?/\mu(\varphi_3) && (6.8)\\
&\vdash p : ? \Rightarrow ??? && (6.9)\\
&\vdash ? - p \rightsquigarrow ?? && (6.10)\\
\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) \oplus ??? &\vdash a_2 : \mu(\tau_3)/\mu(\varphi_3) && (6.11)\\
\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) \oplus \{x : ??\} &\vdash a_3 : \mu(\tau_3)/\mu(\varphi_3) && (6.12)
\end{aligned}$$

dans lesquelles nous devrons trouver par quoi instancier les simples, doubles et triples points d'interrogation (?), (? ?) et (? ? ?).

Commençons par la première prémisse. L'application de l'hypothèse de récurrence à 6.3 nous permet de déduire que :

$$\theta_1(E) \vdash a_1 : \tau_1/\varphi_1 \tag{6.13}$$

sur quoi nous appliquons le lemme de stabilité par substitution (lemme 5.12) avec la substitution $\mu \circ \theta_3 \circ \theta_2 \circ \xi$ pour déduire que :

$$\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) \vdash a_1 : \mu \circ \theta_3 \circ \theta_2 \circ \xi(\tau_1)/\mu \circ \theta_3 \circ \theta_2 \circ \xi(\varphi_1) \tag{6.14}$$

Or nous savons que μ est défini et est un unificateur de $\{\theta_3(\theta_2(\xi(\varphi_1))) = \varphi_3\}$, donc nous avons

$$\mu(\theta_3(\theta_2(\xi(\varphi_1)))) = \mu(\varphi_3) \tag{6.15}$$

En combinant 6.14 et 6.15, nous déduisons que :

$$\mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1(E) \vdash a_1 : \mu \circ \theta_3 \circ \theta_2 \circ \xi(\tau_1)/\mu(\varphi_3) \tag{6.16}$$

ce qui est notre but 6.8 dans lequel nous avons instancié les simples points d'interrogation (?) par $\mu \circ \theta_3 \circ \theta_2 \circ \xi(\tau_1)$.

Intéressons-nous maintenant à notre quatrième prémisse, à savoir 6.11. L'application de l'hypothèse de récurrence à 6.5 nous permet de déduire que : $\theta_2(\xi(\theta_1(E)) \oplus\ E') \vdash a_2 : \tau_2/\varphi_2$ c'est-à-dire que :

$$\theta_2(\xi(\theta_1(E))) \oplus\ \theta_2(E')) \vdash a_2 : \tau_2/\varphi_2 \tag{6.17}$$

Appliquons le lemme de stabilité du typage par substitution à 6.17 avec la substitution $\mu \circ \theta_3$, et nous en déduisons que :

$$\mu(\theta_3(\theta_2(\xi(\theta_1(E))))) \oplus\ \mu(\theta_3(\theta_2(E'))) \vdash a_2 : \mu(\theta_3(\tau_2)/\mu(\theta_3(\varphi_2) \tag{6.18}$$

Puisque μ est un unificateur de $\{\theta_3(\tau_2) = \tau_3\}$ et de $\{\theta_3(\varphi_2) = \varphi_3\}$, nous savons que :

$$\mu(\theta_3(\tau_2)) = \mu(\tau_3) \tag{6.19}$$

$$\mu(\theta_3(\varphi_2)) = \mu(\varphi_3) \tag{6.20}$$

Combinons alors 6.18 avec nos deux égalités 6.19 et 6.20, nous obtenons que :

$$\mu(\theta_3(\theta_2(\xi(\theta_1(E))))) \oplus\ \mu(\theta_3(\theta_2(E'))) \vdash a_2 : \mu(\tau_3)/\mu(\varphi_3) \tag{6.21}$$

qui est notre but de départ 6.11 dans lequel les triples points d'interrogation (? ? ?) ont été instanciés par $\mu(\theta_3(\theta_2(E')))$.

Nous essayons maintenant d'établir notre cinquième prémisse (prémisse 6.12). L'application de l'hypothèse de récurrence sur 6.6 nous permet de déduire que : $\theta_3(\theta_2(\xi(\theta_1(E)))) \oplus \theta_3(\{x : \theta_2(\tau')\}) \vdash a_3 : \tau_3/\varphi_3$, c'est-à-dire que :

$$\theta_3(\theta_2(\xi(\theta_1(E)))) \oplus \{x : \theta_3(\theta_2(\tau'))\} \vdash a_3 : \tau_3/\varphi_3 \tag{6.22}$$

Appliquons le lemme de stabilité par substitution à 6.22 avec la substitution μ et il vient que :

$$\mu(\theta_3(\theta_2(\xi(\theta_1(E))))) \oplus \{x : \mu(\theta_3(\theta_2(\tau')))\} \vdash a_3 : \mu(\tau_3)/\mu(\varphi_3) \tag{6.23}$$

ce que nous attendions en tant que but 6.12, dans lequel nous avons instancié le type inconnu dénoté par les doubles points d'interrogation (? ?) par $\mu(\theta_3(\theta_2(\tau')))$.

En résumé, nous savons maintenant par quoi remplacer tous les points d'interrogation :

- $? \leftarrow \mu \circ \theta_3 \circ \theta_2 \circ \xi(\tau_1)$
- $?? \leftarrow \mu(\theta_3(\theta_2(\tau')))$
- $??? \leftarrow \mu(\theta_3(\theta_2(E')))$

Nous pouvons désormais vérifier qu'il est possible d'établir les prémisses 6.9 et 6.10 dont nous avons besoin pour reconstituer une règle 15, qui sont respectivement devenues :

$$\vdash p : \mu \circ \theta_3 \circ \theta_2 \circ \xi(\tau_1) \Rightarrow \mu(\theta_3(\theta_2(E'))) \tag{6.24}$$

$$\vdash \mu \circ \theta_3 \circ \theta_2 \circ \xi(\tau_1) - p \rightsquigarrow \mu(\theta_3(\theta_2(\tau'))) \tag{6.25}$$

En appliquant l'hypothèse de récurrence (via `Patsubtr`) sur 6.4, il nous vient que :

$$p : \xi(\tau_1) \Rightarrow E' \tag{6.26}$$

$$\xi(\tau_1) - p \rightsquigarrow \tau' \tag{6.27}$$

Nous pouvons alors appliquer le lemme de stabilité du typage par substitution (lemme 5.12, partie 1) à 6.26 et ce même lemme (partie 2) à 6.27 avec pour chacun la substitution $\mu \circ \theta_3 \circ \theta_2$ pour établir les résultats attendus en 6.24 et 6.25.

Ainsi nous avons réussi à réunir toutes les prémisses nécessaire à la reconstruction de la règle 15 attendue ce qui nous permet de conclure la preuve dans le cas présent.

□

Conclusion : Nous concluons cette section par un corollaire établissant que toute exception pouvant s'échapper de l'évaluation de a est correctement prédite par notre analyse.

Théorème 6.6 (Correction de l'analyse) *Si $(\tau, \varphi, \theta, V') = W(\emptyset, a, \emptyset)$ est défini alors pour toute exception v telle que $a \xrightarrow{*}$ `raise` v, v appartient à φ.*

Preuve : Nous avons $\emptyset \vdash a : \tau/\varphi$ par le théorème de correction de l'algorithme W (6.5) et donc que v appartient à φ par le théorème de correction de l'analyse d'exceptions (5.2). □

6.4.5 Complétude de l'algorithme W

Théorème 6.7 (Complétude de `Inst`) *Soit V un ensemble fini de variables et θ' une substitution préservant les sortes. Si $\tau \leq \sigma$ alors $(\tau', V') =$ `Inst`(σ, V) est défini et il existe une substitution ξ préservant les sortes telle que $\tau = \xi(\tau')$.*

Preuve : Notons σ sous la forme $\forall \alpha_i, \rho_j, \delta_k. \tau_\sigma$. Par définition de $\leq$, si $\tau \leq \sigma$ alors :

$$\tau = \tau_\sigma\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\} \tag{6.1}$$

$$\tau_i \text{ bf} \tag{6.2}$$

$$\varphi_j :: K(\rho_j) \tag{6.3}$$

Par définition de `Inst`, celle-ci est trivialement définie et retourne un type τ' de la forme :

$$\tau' = \tau_\sigma\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\} \tag{6.4}$$

$$\text{où } \alpha'_i, \rho'_j, \delta'_k \text{ fraîches } \notin V \tag{6.5}$$

$$\rho'_j \text{ et } \rho_j \text{ ont la même sorte pour tout } j \tag{6.6}$$

Il nous reste alors à montrer l'existence de la substitution ξ annoncée. Prenons alors $\xi = \{\alpha'_i \leftarrow \tau_i, \rho'_j \leftarrow \varphi_j, \delta'_k \leftarrow \pi_k\}$. Cette substitution convient car nous avons bien $\tau = \xi(\tau')$ et elle préserve les sortes. En effet, la sorte de ρ'_j est égale à celle de φ_j qui est elle-même égale (par 6.3) à celle de ρ_j, ce qui nous garantit bien la préservation des sortes.

□

Théorème 6.8 (Complétude de `Patsubtr`) *Soit V un ensemble fini de variables (variables non utilisables commes variables « fraîches ») et θ' une substitution préservant les sortes. Si $p : \theta'(\tau') \Rightarrow E'$ et $\theta'(\tau') - p \rightsquigarrow \tau_s$ et $FV(\tau') \subseteq V$ alors $(E, \tau, \theta, V') =$ `Patsubtr`(p, τ', V) est défini et il existe une substitution ψ préservant les sortes tel que $\tau_s = \psi(\tau)$ et $E' = \psi(E)$ et $\theta' =_V \psi \circ \theta$.*

Preuve : La preuve s'effectue par récurrence sur la forme du motif p en montrant, d'une part qu'aucune étape de l'algorithme n'échoue, et d'autre part que la substitution ψ annoncée existe bien.

Cas p est x.

Hypothèses :

$$x : \theta'(\tau') \Rightarrow E' \tag{6.1}$$
$$\theta'(\tau') - p \rightsquigarrow \tau_s \tag{6.2}$$

Nous voulons prouver que :

$$(E, \tau, \theta, V') = \texttt{Patsubtr}(x, \tau', V) \text{ est défini} \tag{6.3}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau_s = \psi(\tau) \text{ et } E' = \psi(E) \text{ et } \theta' =_V \psi \circ \theta \tag{6.4}$$

De 6.1 en utilisant une règle de typage 19, nous déduisons que :

$$E' = \{x : \theta'(\tau')\} \tag{6.5}$$

De 6.2 en utilisant une règle de soustraction 25, nous déduisons que :

$$\tau_s \text{ est quelconque mais bien formé} \tag{6.6}$$

$\diamondsuit$ L'algorithme *Patsubtr* est trivialement défini dans le cas présent puisqu'il est toujours possible de choisir une variable α fraîche hors de V, ce dernier étant fini par hypothèse.

Montrons maintenant, pour satisfaire notre but 6.4, l'existence de la substitution ψ qui doit satisfaire les conditions suivantes :

$$\tau_s = \psi(\alpha) \tag{6.7}$$
$$E' = \psi(\{x : \tau'\}) \tag{6.8}$$
$$\theta' =_V \psi \circ id \tag{6.9}$$
$$\psi \text{ préserve les sortes} \tag{6.10}$$

Prenons $\psi = \theta' \oplus \{\alpha \leftarrow \tau_s\}$ et vérifions qu'elle satisfait nos exigences.

- $\psi(\alpha) = (\theta' \oplus \{\alpha \leftarrow \tau_s\})(\alpha) = \tau_s$ comme attendu en 6.7.
-

$$\begin{array}{rcll}
\psi(\{x : \tau'\}) & = & \{x : (\theta' \oplus \{\alpha \leftarrow \tau_s\})(\tau') & \text{par choix de } \psi \\
& = & \{x : \theta'(\tau)\} & \text{puisque } \alpha \notin V \text{ et que } FV(\tau') \subseteq V, \\
& & & \alpha \text{ n'apparaît pas dans } \tau' \\
& = & E' & \text{par 6.5 comme attendu en 6.8}
\end{array}$$

- Soit une variable $v \in V$.

$$\begin{array}{rcll}
\psi \circ id(v) & = & \psi(v) & \\
& = & \theta' \oplus \{\alpha \leftarrow \tau_s\} & \text{par choix de } \psi \\
& = & \theta'(v) & \text{car } v \neq \alpha \text{ par hypothèse, puisque } \alpha \notin V, \\
& & & \text{comme attendu en 6.9}
\end{array}$$

- Nous savons par hypothèse que θ' préserve les sortes. De plus $\{\alpha \leftarrow \tau_s\}$ les préserve également puisque τ_s est bien formé. Ainsi la composition de substitutions préservant les sortes étant une substitution qui les préserve, nous avons bien que $\psi = \theta' \oplus \{\alpha \leftarrow \tau_s\}$ préserve les sortes comme attendu en 6.10.

Cas p est i.

Hypothèses :

$$i : \theta'(\tau') \Rightarrow E' \tag{6.1}$$
$$\theta'(\tau') - p \rightsquigarrow \tau_s \tag{6.2}$$

Nous voulons prouver que :

$$(E, \tau, \theta, V') = \mathtt{Patsubtr}(i, \tau', V) \text{ est défini} \tag{6.3}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau_s = \psi(\tau) \text{ et } E' = \psi(E) \text{ et } \theta' =_V \psi \circ \theta \tag{6.4}$$

Commençons par démontrer que l'algorithme *Patsubtr* est défini. De 6.1 en utilisant une règle de typage 20, nous déduisons que :

$$E' = \{\} \tag{6.5}$$
$$\theta'(\tau') \text{ est de la forme } \mathtt{int}[i : \mathtt{Pre};\ \varphi] \tag{6.6}$$

De 6.2 et 6.6 en utilisant une règle de soustraction 23, nous déduisons que :

$$\tau_s = \mathtt{int}[i : \pi;\ \varphi] \tag{6.7}$$

$\diamondsuit$ La première étape de l'algorithme *Patsubtr* est trivialement définie puisqu'il est toujours possible de choisir une variable de type fraîche hors de V. Prouvons maintenant qu'il existe un unificateur ψ_1 de $\{\tau' = \mathtt{int}[i : \mathtt{Pre};\ \rho]\}$.

Prenons $\psi_1 = \theta' \oplus \{\rho \leftarrow \varphi;\ \delta \leftarrow \pi\}$.

- Puisque ρ et δ n'appartiennent pas à V et que $FV(\tau') \subseteq V$, alors ρ et δ n'apparaissent pas dans τ'. Ainsi, $\psi_1(\tau') = (\theta' \oplus \{\rho \leftarrow \varphi;\ \delta \leftarrow \pi\})(\tau') = \theta'(\tau') \stackrel{6.6}{=} \mathtt{int}[i : \mathtt{Pre};\ \varphi]$
- $\psi_1(\mathtt{int}[i : \mathtt{Pre};\ \rho]) = \mathtt{int}[i : \mathtt{Pre};\ \psi_1(\rho)] = \mathtt{int}[i : \mathtt{Pre};\ (\theta' \oplus \{\rho \leftarrow \varphi;\ \delta \leftarrow \pi\})(\rho)] = \mathtt{int}[i : \mathtt{Pre};\ \varphi]$ également.
- Par hypothèse, θ' préserve les sortes. De plus $\{\rho \leftarrow \varphi;\ \delta \leftarrow \pi\}$ les préserve également puisque φ est bien formé. Ainsi ψ_1, composition de ces deux substitutions, préserve les sortes.

Donc il existe bien un tel unificateur. L'algorithme $\mathtt{mgu}_V$ étant complet (théorème 6.3), il trouve alors effectivement μ, l'unificateur principal. De plus ψ_1 étant unificateur et μ étant *principal*, nous déduisons que pouvons écrire ψ_1 sous la forme :

$$\psi_1 =_{V \cup \{\rho\}} \psi_2 \circ \mu \tag{6.8}$$

$\diamondsuit$ La seconde étape de notre algorithme n'échoue ainsi pas. Les étapes restant ne posent pas de problème particulier, ainsi l'algorithme est-il bien défini dans le cas présent.

Nous devons maintenant, pour démontrer x6.4, trouver une substitution ψ satisfaisant les critères suivants :

$$\mathtt{int}[i : \pi;\ \varphi] = \psi(\mu(\mathtt{int}[i : \delta;\ \rho])) \tag{6.9}$$
$$\{\} = \psi(\{\}) \tag{6.10}$$
$$\theta' =_V \psi \circ \mu \tag{6.11}$$
$$\psi \text{ préserve les sortes} \tag{6.12}$$

Prenons $\psi = \psi_2$ trouvée ci-dessus et vérifions son comportement vis-à-vis de nos exigences.

- $$\begin{array}{rcll} \psi(\mu(\mathtt{int}[i:\delta;\,\rho])) & = & \psi_2(\mu(\mathtt{int}[i:\delta;\,\rho])) & \text{par choix de } \psi \\ & = & \psi_1(\mathtt{int}[i:\delta;\,\rho]) & \text{par 6.8} \\ & = & (\theta' \oplus \{\rho \leftarrow \varphi;\, \delta \leftarrow \pi\})(\mathtt{int}[i:\delta;\,\rho]) & \text{par définition de } \psi_1 \\ & = & \mathtt{int}[i:\pi;\,\varphi] & \text{comme attendu en 6.9} \end{array}$$
- $\{\} = \psi(\{\})$ est trivialement satisfait pour toute substitution ψ, comme attendu en 6.10
- Soit une variable $v \in V$.

$$\begin{array}{rcll} \psi \circ \mu(v) & = & \psi_2 \circ \mu(v) & \text{par choix de } \psi \\ & = & \psi_1(v) & \text{par définition de } \psi_2 \\ & = & (\theta' \oplus \{\rho \leftarrow \varphi;\, \delta \leftarrow \pi\})(v) & \text{car } v \neq \rho \text{ et } v \neq \delta \text{ par hypothèse,} \\ & & & \text{puisque } \rho, \delta \notin V \\ & = & \theta'(v) & \text{comme attendu en 6.11} \end{array}$$

- ψ préserve les sortes car elle est égale à ψ_2 qui les préserve par complétude et principalité de l'algorithme d'unification (théorème 6.3). C'est bien le résultat attendu en 6.12.

Cas p est C.

On se convaincra aisément, afin de ne pas rallonger inutilement cette démonstration, que ce cas se prouve de la même façon que le précédent, en remplaçant le constructeur de type `int` par `exn`.

Cas p est $D(p_1)$.

Hypothèses :

$$D(p_1) : \theta'(\tau') \Rightarrow E' \tag{6.1}$$
$$\theta'(\tau') - D(p_1) \rightsquigarrow \tau_s \tag{6.2}$$
$$FV(\tau') \subseteq V \tag{6.3}$$

Nous voulons prouver que :

$$(E, \tau, \theta, V') = \mathtt{Patsubtr}(D(p_1), \tau', V) \text{ est défini} \tag{6.4}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau_s = \psi(\tau) \text{ et } E' = \psi(E) \text{ et } \theta' =_V \psi \circ \theta \tag{6.5}$$

Commençons par démontrer que l'algorithme *Patsubtr* est défini. De 6.1 par application d'une règle de typage 22, nous déduisons que :

$$\theta'(\tau') \text{ est de la forme } \mathtt{exn}[D(\tau_1'');\, \varphi] \tag{6.6}$$
$$\tau_1'' \leq \text{TypeArg}(D) \tag{6.7}$$
$$\vdash p_1 : \tau_1'' \Rightarrow E' \tag{6.8}$$

De 6.2 en utilisant une règle de soustraction 26, nous déduisons :

$$\tau_1'' - p_1 \rightsquigarrow \tau_s' \tag{6.9}$$
$$\tau_s \text{ est de la forme } \mathtt{exn}[D(\tau_s');\, \varphi] \tag{6.10}$$

Ainsi, par 6.7, nous déduisons que :

$$(\tau_1, V_0) = \mathtt{Inst}(\mathit{TypeArg}(D), V) \text{ est défini} \tag{6.11}$$
$$\tau_1'' = \xi(\tau_1) \tag{6.12}$$

♢ La déduction 6.11 nous montre que la première étape de l'algorithme *Patsubtr* est effectivement définie. Nous devons maintenant vérifier qu'il en est de même pour sa seconde étape. Nous voudrions appliquer l'hypothèse de récurrence à 6.8 et 6.9, non pas avec avec la substitution ξ comme 6.12 nous le permet d'ores et déjà, mais avec une substitution θ'' telle que $\theta''(\tau_1) = \xi(\tau_1)$ et $\theta''(\tau') = \theta'(\tau')$. Construisons une telle substitution par :

$$\theta''(\alpha) = \begin{cases} \theta'(\alpha) & \text{si } \alpha \in V \\ \xi(\alpha) & \text{si } \alpha \notin V \end{cases} \tag{6.13}$$

Puisque τ_1 est une instance triviale d'un schéma de types clos (plus précisément de *TypeArg*(D)), toutes ses variables sont fraîches, choisies hors de V_1, donc à plus forte raison hors de V. Ainsi on a bien :

$$\theta''(\tau_1) = \xi(\tau_1) \tag{6.14}$$

D'autre part, par l'hypothèse 6.3, $FV(\tau') \subseteq V$, donc on a bien :

$$\theta''(\tau') = \theta'(\tau') \tag{6.15}$$

Ainsi la substitution θ'' que nous avons construite correspond bien à nos attentes. De 6.12 et 6.14 nous pouvons alors déduire que :

$$\tau_1'' = \theta''(\tau_1) \tag{6.16}$$

Nous pouvons maintenant appliquer l'hypothèse de récurrence à 6.8 et 6.9, avec avec la substitution θ'' grâce à 6.16 et nous en déduisons qu'il existe une substitution ψ_1 et que :

$$\begin{aligned} (E_1, \tau_1', \theta_1, V_1) = Patsubr(p_1, \tau_1, V_0) \text{ est défini} && (6.17)\\ \tau_s' = \psi_1(\tau_1') && (6.18)\\ E' = \psi_1(E_1) && (6.19)\\ \theta'' =_{V_0} \psi_1 \circ \theta_1 && (6.20)\\ \psi_1 \text{ préserve les sortes} && (6.21)\\ && (6.22) \end{aligned}$$

$\diamondsuit$ La déduction 6.17 nous montre que la seconde étape de l'algorithme *Patsubtr* est définie. Sa troisième étape ne pose pas de problème puisqu'il est toujours possible de choisir une variable ρ hors de V_1. Il nous faut donc maintenant montrer qu'il existe un unificateur ψ_2 de $\{\theta_1(\tau') = \texttt{exn}[D(\theta_1(\tau_1));\ \rho]\}$.

Prenons $\psi_2 = \psi_1 \oplus \{\rho \leftarrow \varphi\}$.

•

$$\begin{array}{rcll} \psi_2(\theta_1(\tau')) & = & (\psi_1 \oplus \{\rho \leftarrow \varphi\})(\theta_1(\tau')) & \text{par choix de } \psi_2 \\ & = & \psi_1(\theta_1(\tau')) & \text{car } \rho \text{ n'apparaît pas dans } \tau' \\ & & & \text{et est hors de portée de } \theta_1 \\ & = & \theta''(\tau') & \text{par 6.20 car } FV(\tau') \subseteq V \text{ donc } \subseteq V_0 \\ & = & \theta'(\tau') & \text{par 6.15} \\ & = & \texttt{exn}[D(\tau_1'');\ \varphi] & \text{par 6.6} \end{array}$$

•

$$\begin{array}{rcll} \psi_2(\texttt{exn}[D(\theta_1(\tau_1));\ \rho]) & = & (\psi_1 \oplus \{\rho \leftarrow \varphi\})(\texttt{exn}[D(\theta_1(\tau_1));\ \rho]) & \text{par choix de } \psi_2 \\ & = & \texttt{exn}[D((\psi_1 \oplus \{\rho \leftarrow \varphi\})(\theta_1(\tau_1)));\ \varphi] & \\ & = & \texttt{exn}[D(\psi_1(\theta_1(\tau_1)));\ \varphi] & \text{car } \rho \text{ n'apparaît pas} \\ & & & \text{dans } \tau_1 \text{ et est hors} \\ & & & \text{de portée de } \theta_1 \\ & = & \texttt{exn}[D(\theta''(\tau_1));\ \varphi] & \text{par 6.20} \\ & = & \texttt{exn}[D(\xi(\tau_1));\ \varphi] & \text{par 6.14} \\ & = & \texttt{exn}[D(\tau_1'');\ \varphi] & \text{par 6.12} \end{array}$$

- Par 6.21, ψ_1 préserve les sortes. La rangée φ étant bien formée, et par choix ψ_2 étant la composition de ces deux substitutions, nous avons bien que ψ_2 préserve les sortes.

Donc il existe bien un tel unificateur. L'algorithme $\mathtt{mgu}_V$ étant complet (théorème 6.3), il trouve alors effectivement μ, l'unificateur principal. De plus ψ_2 étant unificateur et μ étant *principal*, nous déduisons que pouvons écrire ψ_2 sous la forme :

$$\psi_2 =_{V_1 \cup \{\rho\}} \psi_3 \circ \mu \tag{6.23}$$

♢ Ainsi la quatrième étape de l'algorithme *Patsubr* est bien définie, montrant que ce dernier l'est totalement dans le cas présent.

Montrons maintenant, pour satisfaire notre but 6.5, l'existence de la substitution ψ qui doit satisfaire les conditions suivantes :

$$\tau_s = \psi(\mathtt{exn}[D(\mu(\tau_1')); \mu(\rho)]) \tag{6.24}$$
$$E' = \psi(\mu(E_1)) \tag{6.25}$$
$$\theta' =_V \psi \circ \mu \circ \theta_1 \tag{6.26}$$
$$\psi \text{ préserve les sortes} \tag{6.27}$$

Prenons $\psi = \psi_3$ et vérifions qu'elle satisfait bien nos hypothèses.

- $\psi(\mathtt{exn}[D(\mu(\tau_1')); \mu(\rho)]) \overset{\text{choix de } \psi}{=} \psi_3(\mathtt{exn}[D(\mu(\tau_1')); \mu(\rho)]) = \mathtt{exn}[D(\psi_3 \circ \mu(\tau_1')); \psi_3 \circ \mu(\rho)]$ $\overset{6.23}{=} \mathtt{exn}[D(\psi_2(\tau_1')); \psi_2(\rho)] \overset{\text{déf de } \psi_2}{=} \mathtt{exn}[D((\psi_1 \oplus \{\rho \leftarrow \varphi\})(\tau_1')); (\psi_1 \oplus \{\rho \leftarrow \varphi\})(\rho)] =$ $\mathtt{exn}[D(\psi_1(\tau_1')); \varphi] \overset{6.18}{=} \mathtt{exn}[D(\tau_s'); \varphi] \overset{6.10}{=} \tau_s'$ comme attendu en 6.24.
- $\psi(\mu(E_1)) \overset{\text{choix de } \psi}{=} \psi_3(\mu(E_1)) \overset{6.23}{=} \psi_2(E_1) \overset{\text{déf de } \psi_2}{=} (\psi_1 \oplus \{\rho \leftarrow \varphi\})(E_1) \overset{\text{car } \rho \notin E_1}{=} \psi_1(E_1)$ $\overset{6.19}{=} E'$ comme attendu en 6.25.
- Soit une variable $v \in V$.

$$\begin{array}{rcll}
\psi \circ \mu(v) & = & \psi_3 \circ \mu \circ \theta_1(v) & \text{par choix de } \psi \\
 & = & \psi_2 \circ \theta_1(v) & \text{par 6.23} \\
 & = & (\psi_1 \oplus \{\rho \leftarrow \varphi\}) \circ \theta_1(v) & \text{par définition de } \psi_2 \\
 & = & \psi_1 \circ \theta_1(v) & \text{car } v \text{ hors de portée de } \theta_1 \text{ et } v \neq \rho \text{ par} \\
 & & & \text{hypothèse, puisque } \rho \notin V \\
 & = & \theta''(v) & \text{par 6.20} \\
 & = & \theta'(v) & \text{par 6.13 comme attendu en 6.26.}
\end{array}$$

- ψ préserve les sortes car elle est égale à ψ_3 qui les préserve par complétude et principalité de l'algorithme d'unification (théorème 6.3). C'est bien le résultat attendu en 6.27.

□

Théorème 6.9 (Complétude de l'algorithme W) *Soit E un environnement bien formé, a une expression, et V un ensemble fini de variables de type tel que $FV(E) \subseteq V$. S'il existe une substitution θ' préservant les sortes et des types τ', φ' tels que $\theta'(E) \vdash a : \tau'/\varphi'$, alors $(\tau, \varphi, \theta, V') = W(E, a, V)$ est défini et il existe une substitution ψ préservant les sortes tel que $\tau' = \psi(\tau)$ et $\varphi' = \psi(\varphi)$ et $\theta' =_V \psi \circ \theta$.*

Preuve : La preuve s'effectue par récurrence sur la forme de l'expression a, en démontrant d'une part que chaque étape de l'algorithme est définie, et d'autre part que la substitution ψ annoncée existe bien.

Cas a est x.

Hypothèses :

$$\theta'(E) \vdash x : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, x, V) \text{ est défini} \tag{6.2}$$

$$\exists \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. À partir l'hypothèse 6.1, nous pouvons utiliser une règle de typage 10 de la forme :

$$\frac{\tau \leq \theta'(E)(x) \qquad \vdash \varphi' :: \mathtt{EXN}(\emptyset)}{\theta'(E) \vdash x : \tau'/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\tau \leq \theta'(E)(x) \tag{6.4}$$

$$\varphi' :: \mathtt{EXN}(\emptyset) \tag{6.5}$$

$\diamondsuit$ La première étape de l'algorithme est bien définie puisqu'il est toujours possible de choisir une variable de rangée fraîche et de sorte $\mathtt{EXN}(\emptyset)$.

$\diamondsuit$ Quant à la seconde étape, nous devons nous assurer que $\mathtt{Inst}(E(x), V \cup \{\rho\})$ est bien définie. Puisque $\theta'(E) \vdash x : \tau'/\varphi'$, on sait par 6.4 que $x \in \mathrm{Dom}\,(\theta'(E))$, donc $x \in \mathrm{Dom}\,(E)$ et $\tau' \leq \theta'(E(x))$ ce qui implique bien, par définition de $\leq$ et de $\mathtt{Inst}$ que cette dernière est définie. Ainsi, dans le cas présent l'algorithme W est bien défini comme attendu en 6.2 puisque chacune de ses étapes l'est.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\tau' = \psi(\mathtt{Inst}(E(x))) \tag{6.6}$$

$$\varphi' = \psi(\mu(\rho)) \tag{6.7}$$

$$\theta' =_V \psi \circ id \tag{6.8}$$

Notons $E(x)$ sous la forme $\forall \alpha_i, \rho_j, \delta_k. \tau_x$ avec $\alpha_i, \rho_j, \delta_k$ choisies dans V' hors de portée de θ'. On a donc $\theta'(E(x)) = \forall \alpha_i, \rho_j, \delta_k. \theta'(\tau_x)$.

Soit ψ_x la substitution $\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\}$ qui a été appliquée à $E(x)$ par $\mathtt{Inst}$ lors de l'instanciation (ainsi, l'algorithme étant défini, il nous renvoie un type τ de la forme $\psi_x(\tau_x)$).

On note ψ' la substitution sur les $\alpha_i, \rho_j, \delta_k$ telle que $\tau' = \psi'(\theta'(\tau_x))$ et $\varphi' = \psi'(\theta'(\rho))$. Prenons alors $\psi = \psi' \circ \theta' \circ \psi_x$. et vérifions qu'elle satisfait nos exigences.

- On a $\psi(\tau) = \psi'(\theta'(\tau_x)) = \tau'$, ce qui était attendu en 6.6
- On a $\psi(\rho) = \psi' \circ \theta' \circ \psi_x(\rho)$. Comme ρ est une variable fraîche hors de portée de ψ_x, c'est égal à $\psi' \circ \theta'(\rho)$ qui par définition de ψ' est bien égal à φ' comme attendu en 6.7.
- Soit une variable $v \in V$.

$$\begin{array}{rcll} \psi \circ \theta(v) & = & \psi \circ id(v) & \text{par définition de } \theta \text{ dans l'algorithme} \\ & = & \psi(v) & \\ & = & \psi' \circ \theta' \circ \psi_x(v) & \text{par choix de } \psi \\ & = & \psi' \circ \theta'(v) & \text{car } v \text{ n'est pas } \alpha_i, \rho_j \text{ ni } \delta_k \end{array}$$

De plus, $\alpha_i, \rho_j, \delta_k$ étant les seules variables variables modifiées par ψ' et ces mêmes variables étant choisies hors de portée de θ',

$$= \quad \theta'(v) \quad \text{comme attendu en 6.8}$$

- Par hypothèse, θ' préserve les sortes. Par construction, comme il est toujours possible de choisir des variables ρ'_j de la sorte adéquate, $\psi_x = \{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\}$ préserve également les sortes. Enfin, par le fait que τ_x soit bien formé, qu'il soit toujours possible de choisir ρ de sorte adéquate, et par construction de ψ', cette dernière les préserve. Ainsi, ψ étant la composition de ces trois substitutions qui préservent les sortes, elle les préserve aussi.

Cas a est i.
Hypothèses :

$$\theta'(E) \vdash i : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, i, V) \text{ est défini} \tag{6.2}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

L'algorithme W est trivialement défini et nous avons donc

$$\tau = \mathtt{int}[i : \mathtt{Pre};\, \rho] \tag{6.4}$$
$$\varphi = \rho \tag{6.5}$$
$$\theta = id \tag{6.6}$$

À partir l'hypothèse 6.1, nous pouvons utiliser une règle de typage 11 de la forme :

$$\frac{\vdash \varphi_1 :: \mathtt{INT}(\{i\}) \qquad \vdash \varphi' :: \mathtt{EXN}(\emptyset)}{\theta'(E) \vdash i : \mathtt{int}[i : \mathtt{Pre};\, \varphi_1]/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\tau' \text{ est de la forme } \mathtt{int}[i : \mathtt{Pre};\, \varphi_1] \tag{6.7}$$

Nous allons maintenant nous concentrer sur l'existence de la substitution ψ afin de démontrer 6.3. Celle-ci doit donc satisfaire les propriétés suivantes :

$$\tau' = \psi(\mathtt{int}[i : \mathtt{Pre};\, \rho]) \tag{6.8}$$
$$\varphi' = \psi(\rho') \tag{6.9}$$
$$\theta' =_V \psi \circ id \tag{6.10}$$
$$\psi \text{ préserve les sortes} \tag{6.11}$$

Prenons $\psi = \{\rho \leftarrow \varphi_1;\, \rho' \leftarrow \varphi'\}$. Clairement ψ satisfait ainsi 6.8 et 6.9. Soit une variable $v \in V$. On a $\psi \circ id(v) = \{\rho \leftarrow \varphi_1;\, \rho' \leftarrow \varphi'\}(v)$. Comme ρ et ρ' n'appartiennent pas à V, ceci est égal à $id(v)$ c'est-à-dire v. De plus, $\theta'(v) = v$, et donc nous obtenons bien le résultat attendu en 6.10. Pour terminer, étant toujours possible de choisir ρ et ρ' de sorte adéquate, la substitution ψ préserve bien les sortes comme attendu en 6.11.

Cas a est $a_1(a_2)$.
Hypothèses :

$$\theta'(E) \vdash a_1(a_2) : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, a_1(a_2), V) \text{ est défini} \tag{6.2}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. À partir l'hypothèse 6.1, nous pouvons utiliser une règle de typage 13 de la forme :

$$\frac{\theta'(E) \vdash a_1 : (\tau'' \xrightarrow{\varphi'} \tau')/\varphi' \qquad \theta'(E) \vdash a_2 : \tau''/\varphi'}{\theta'(E) \vdash a_1(a_2) : \tau'/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\theta'(E) \vdash a_1 : (\tau'' \xrightarrow{\varphi} \tau')/\varphi' \tag{6.4}$$
$$\theta'(E) \vdash a_2 : \tau''/\varphi' \tag{6.5}$$

Appliquons l'hypothèse de récurrence sur 6.4, nous déduisons alors qu'il existe une substitution ψ_1 et que :

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V) \text{ est défini} \tag{6.6}$$
$$\tau'' \xrightarrow{\varphi'} \tau' = \psi_1(\tau_1) \tag{6.7}$$
$$\varphi' = \psi_1(\varphi_1) \tag{6.8}$$
$$\theta' =_V \psi_1 \circ \theta_1 \tag{6.9}$$
$$\psi_1 \text{ préserve les sortes} \tag{6.10}$$

$\diamondsuit$ La déduction 6.6 nous permet de dire que la première étape de l'algorithme W dans le cas présent est bien définie. Maintenant, en combinant 6.5 avec 6.9, nous déduisons que :

$$\psi_1 \circ \theta_1(E) \vdash a_2 : \tau''/\varphi' \tag{6.11}$$

Appliquons l'hypothèse de récurrence (en utilisant le lemme de propriétés structurelles 6.4) sur 6.11, (licite car $\psi_1 \circ \theta_1$ préserve les sortes par construction) il vient alors qu'il existe une substitution ψ_2 et que :

$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1) \text{ est défini} \tag{6.12}$$
$$\tau'' = \psi_2(\tau_2) \tag{6.13}$$
$$\varphi' = \psi_2(\varphi_2) \tag{6.14}$$
$$\psi_1 =_{V_1} \psi_2 \circ \theta_2 \tag{6.15}$$
$$\psi_2 \text{ préserve les sortes} \tag{6.16}$$

$\diamondsuit$ La déduction 6.12 nous permet de dire que la seconde étape de l'algorithme W dans le cas présent est bien définie.

$\diamondsuit$ La troisième étape de l'algorithme est définie puisqu'il est toujours possible de choisir une variable de type fraîche hors de V_2, donc il nous reste à vérifier la dernière étape, à savoir l'existence d'un unificateur ψ_3 pour l'ensemble d'équations $\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha;\ \theta_2(\varphi_1) = \varphi_2\}$. Nous essayons donc maintenant d'en exhiber un. Prenons :

$$\psi_3 = \psi_2 \oplus \{\alpha \leftarrow \tau'\} \tag{6.17}$$

Tout d'abord, ψ_3 préserve les sortes puisque τ' est bien formé et que par 6.16 ψ_2 les préserve. La variable α n'apparaissant pas dans τ_2 et par les hypothèses 6.13 et 6.14, nous avons que $\psi_3(\tau_2 \xrightarrow{\varphi_2} \alpha) = \tau'' \xrightarrow{\varphi'} \tau'$. De plus, α n'apparaissant pas dans τ_1 ni dans $Rng(\theta_2)$, par 6.17 nous avons $\psi_3(\theta_2(\tau_1)) = \psi_2(\theta_2(\tau_1)) =$ (par hypothèse 6.15) à $\psi_1(\tau_1) =$ (par 6.7) à $\tau'' \xrightarrow{\varphi'} \tau'$. Ainsi ψ_3 est bien un unificateur de la première équation de notre ensemble. Vérifions qu'il l'est également pour le seconde. α étant hors de portée de ψ_2, n'apparaissant pas dans φ_2 et par l'hypothèse 6.14, nous avons que $\psi_3(\varphi_2) = \varphi'$. De plus, α n'apparaissant pas dans $Rng(\theta_2)$ et étant hors de porté de ψ_2, nous avons que $\psi_3(\theta_2(\varphi_1)) = (\psi_2 \oplus \{\alpha \leftarrow \tau'\})(\theta_2(\varphi_1)) = \psi_2(\theta_2(\varphi_1)) =$ (par 6.15) à $\psi_1(\varphi_1) =$ (par 6.8) à φ'. Ainsi ψ_3 est également solution de notre seconde équation, et donc ψ_3 est un unificateur de l'ensemble d'équations complet. Puisque l'algorithme $\mathtt{mgu}_V$ est complet (théorème 6.3), il trouve alors effectivement μ, unificateur principal. De plus ψ_3 étant unificateur et μ étant *principal*, nous déduisons que pouvons écrire ψ_3 sous la forme :

$$\psi_3 =_{V_2 \cup \{\alpha\}} \psi_4 \circ \mu \tag{6.18}$$

$\diamondsuit$ Ainsi la dernière étape de W n'ayant pas échoué, nous venons de prouver que l'algorithme est effectivement défini dans le cas présent, but attendu en 6.2.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\tau' = \psi(\mu(\alpha)) \tag{6.19}$$
$$\varphi' = \psi(\mu(\varphi_2)) \tag{6.20}$$
$$\theta' =_V \psi \circ \mu \circ \theta_2 \circ \theta_1 \tag{6.21}$$
$$\psi \text{ préserve les sortes} \tag{6.22}$$

Prenons $\psi = \psi_4$ trouvée ci-dessus (6.18) et vérifions qu'elle satisfait nos exigences.

- Avec ce choix, $\psi(\mu(\alpha)) = \psi_4(\mu(\alpha)) \overset{6.18}{=} \psi_3(\alpha) \overset{6.17}{=} (\psi_2 \oplus \{\alpha \leftarrow \tau'\})(\alpha) = \tau'$. Ainsi notre propriété 6.19 est bien vérifiée par notre substitution.
- Avec ce choix nous avons $\psi(\mu(\varphi_2)) = \psi_4(\mu(\varphi_2)) \overset{6.18}{=} \psi_3(\varphi_2) \overset{6.17}{=} (\psi_2 \oplus \{\alpha \leftarrow \tau'\}) \circ \varphi_2$. Comme α est hors de porté de φ_2, ceci est égal à $\psi_2 \circ \varphi_2$ qui est bien égal (par 6.14) à φ', comme attendu en 6.20.
- Soit une variable $v \in V$ (*a fortiori*, nous avons $v \in V_1$, $v \in V_2$, $v \neq \alpha$).

$$\begin{array}{rcll}
\psi(\theta(v)) & = & \psi(\mu(\theta_2(\theta_1(v)))) & \text{par définition de } \theta \text{ dans l'algorithme} \\
& = & \psi_4(\mu(\theta_2(\theta_1(v)))) & \text{par choix de } \psi \\
& = & \psi_3(\theta_2(\theta_1(v))) & \text{car } v \in V_2 \cup \{\alpha\} \text{ et par 6.18} \\
& = & (\psi_2 \oplus \{\alpha \leftarrow \tau'\})(\theta_2(\theta_1(v))) & \text{par construction de } \psi_3 \\
& = & \psi_2(\theta_2(\theta_1(v))) & \text{car } \theta_2(\theta_1(v)) \neq \alpha \text{ (explication ci-dessous)}
\end{array}$$

En effet, supposons que $\theta_2(\theta_1(v)) = \alpha$. De l'hypothèse que $\alpha \notin V_2$, nous déduisons par propriétés structurelles de W (lemme 6.4) que α est hors de portée de θ_2. Donc ce serait que $\theta_1(v) = \alpha$. De l'hypothèse que $\alpha \notin V_1$, nous déduisons maintenant (à nouveau par le lemme 6.4) que α est hors de portée de θ_1. Ce serait donc que $v = \alpha$. Or c'est en contradiction avec notre hypothèse de départ disant qu'*a fortiori* nous avons $v \neq \alpha$.

$$= \quad \psi_1(\theta_1(v)) \quad \text{car } FV(\theta_1(v)) \subseteq V_1 \text{ (explication ci-dessous) et par 6.15}$$

En effet, supposons que nous n'ayons pas $FV(\theta_1(v)) \subseteq V_1$. Alors il existerait $\gamma \in FV(\theta_1(v))$ telle que $\gamma \notin V_1$. Ainsi, γ serait hors de portée de θ_1 puisque par le lemme de propriétés structurelles de W (lemme 6.4) toute variable non présente dans V_1 est hors de portée de θ_1. D'où on aurait $v = \gamma$ (par le lemme 5.6), ce qui est une contradiction car par hypothèse $v \in V \subseteq V_1$ et nous venons de voir que $\gamma \notin V_1$.

$$= \quad \theta'(v) \quad \text{car } v \in V$$

ce qui est le résultat attendu en 6.21, nous permettant de conclure la preuve de 6.3.
- Par complétude et principalité de l'algorithme d'unification (théorème 6.3), ψ_4 préserve les sortes donc ψ également comme attendu en 6.22.

Cas a est $\lambda x.a_1$.

Hypothèses :

$$\theta'(E) \vdash \lambda x.a_1 : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, \lambda x.a_1, V) \text{ est défini} \tag{6.2}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. À partir l'hypothèse 6.1, nous pouvons utiliser une règle de typage 12 de la forme :

$$\frac{\vdash \tau_2' \text{ bf} \qquad \theta'(E) \oplus \{x : \tau_2'\} \vdash a_1 : \tau_1'/\varphi_1' \qquad \vdash \varphi' :: \mathtt{EXN}(\emptyset)}{\theta'(E) \vdash \lambda x.a_1 : (\tau_2' \xrightarrow{\varphi_1'} \tau_1')/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\vdash \tau_2' \text{ bf} \tag{6.4}$$
$$\theta'(E) \oplus \{x : \tau_2'\} \vdash a_1 : \tau_1'/\varphi_1' \tag{6.5}$$
$$\vdash \varphi' :: \mathtt{EXN}(\emptyset) \tag{6.6}$$
$$\tau' \text{ est de la forme } \tau_2' \xrightarrow{\varphi_1'} \tau_1' \tag{6.7}$$

Prenons une variable $\alpha \notin V$. Ainsi, α n'appartenant pas à $FV(E)$, la substitution $\theta' \oplus \{\alpha \leftarrow \tau_2'\}$ est telle que :

- $(\theta' \oplus \{\alpha \leftarrow \tau_2'\})(E) = \theta'(E)$
- $(\theta' \oplus \{\alpha \leftarrow \tau_2'\})(\alpha) = \tau_2'$, donc $(\theta' \oplus \{\alpha \leftarrow \tau_2'\})\{x : \alpha\} = \{x : \tau_2'\}$

Donc, combiné avec 6.5 nous avons :

$$(\theta' \oplus \{\alpha \leftarrow \tau_2'\})(E \oplus \{x : \alpha\}) \vdash a_1 : \tau_1'/\varphi_1' \tag{6.8}$$

Appliquons alors l'hypothèse de récurrence sur 6.8, avec notre substitution $\theta' \oplus \{\alpha \leftarrow \tau_2'\}$ qui préserve les sortes (car τ_2' est bien formé et θ' préserve les sortes par hypothèse). Nous déduisons donc qu'il existe une substitution ψ_1 et que :

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E \oplus \{x : \alpha\}, a_1, V \cup \{\alpha\}) \text{ est défini} \tag{6.9}$$
$$\tau_1' = \psi_1(\tau_1) \tag{6.10}$$
$$\varphi_1' = \psi_1(\varphi_1) \tag{6.11}$$
$$\theta' \oplus \{\alpha \leftarrow \tau_2'\} =_{V \cup \{\alpha\}} \psi_1 \circ \theta_1 \tag{6.12}$$
$$\psi_1 \text{ préserve les sortes} \tag{6.13}$$

$\diamondsuit$ La déduction 6.9 nous montre que la première étape de notre algorithme est bien définie. Les autres étapes ne posant pas de problème particulier, nous déduisons donc que l'algorithme est bien défini dans le cas présent, comme attendu en 6.2.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\tau_2' \xrightarrow{\varphi_1'} \tau_1' = \psi(\theta_1(\alpha \xrightarrow{\varphi_1} \tau_1)) \tag{6.14}$$
$$\varphi' = \psi(\rho) \tag{6.15}$$
$$\theta' =_V \psi \circ \theta_1 \tag{6.16}$$
$$\psi \text{ préserve les sortes} \tag{6.17}$$

Prenons $\psi = \psi_1 \oplus \{\rho \leftarrow \varphi'\}$ et vérifions qu'elle satisfait nos exigences.

- Avec ce choix, $\psi(\theta_1(\alpha \xrightarrow{\varphi_1} \tau_1)) = (\psi_1 \oplus \{\rho \leftarrow \varphi'\})(\theta_1(\alpha \xrightarrow{\varphi_1} \tau_1))$. Puisque $\rho \notin V_1$, ρ n'apparaît ni dans τ_1 ni dans φ_1 et donc ceci est égal à $\psi_1(\theta_1(\alpha \xrightarrow{\varphi_1} \tau_1))$. Par 6.10 et 6.11, ceci est égal à $\psi_1(\theta_1(\alpha)) \xrightarrow{\varphi_1'} \tau_1'$ qui est égal, par 6.12, à $((\theta' \oplus \{\alpha \leftarrow \tau_2'\})(\alpha)) \xrightarrow{\varphi_1'} \tau_1'$, c'est-à-dire $\tau_2' \xrightarrow{\varphi_1'} \tau_1'$ comme attendu en 6.14.
- Avec ce choix, $\psi(\rho) = \psi_1 \oplus \{\rho \leftarrow \varphi'\}(\rho) = \varphi'$, comme attendu en 6.10.
- Soit une variable $v \in V$.

$$\begin{array}{rcll} \psi \circ \theta(v) & = & \psi \circ \theta_1(v) & \text{par définition de } \theta \text{ dans l'algorithme} \\ & = & (\psi_1 \oplus \{\rho \leftarrow \varphi'\}) \circ \theta_1(v) & \text{par définition de } \psi \end{array}$$

On sait que $v \neq \rho$ puisque, v appartenant à V elle appartient également à V_1, or ρ est choisie hors de V_1. Donc :

$$\begin{array}{lll} = & \psi_1 \circ \theta_1(v) & \text{par la remarque ci-dessus} \\ = & \theta' \oplus \{\alpha \leftarrow \tau_2'\}(v) & \text{par 6.12, et le fait que } v \in V \text{ donc } v \in V \cup \{\alpha\} \end{array}$$

On sait également que $v \neq \alpha$ puisque $v \in V$, or α est explicitement choisie hors de V. Donc :

$$= \quad \theta'(v) \quad \text{comme attendu en 6.16.}$$

- Par 6.13 ψ_1 préserve les sortes. De plus φ' étant bien sorté et étant toujours possible de choisir ρ de cette sorte, la substitution $\{\rho \leftarrow \varphi'\}$ préserve également les sortes. Ainsi il en va de même pour ψ comme attendu en 6.17.

Cas a est `let` $x = a_1$ `in` $a_2 : \tau/\varphi$.
Hypothèses :

$$\theta'(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, \texttt{let } x = a_1 \texttt{ in } a_2, V) \text{ est défini} \tag{6.2}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. Sa première étape est trivialement définie puisqu'il est, par définition, toujours possible de choisir une nouvelle variable fraîche. À partir l'hypothèse 6.1, nous pouvons utiliser une règle de typage 14 de la forme :

$$\frac{\theta'(E) \vdash a_1 : \tau_1'/\varphi' \qquad \theta'(E) \oplus \{x : \mathrm{Gen}\,(\tau_1', \theta'(E), \varphi')\} \vdash a_2 : \tau'/\varphi'}{\theta'(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau'/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\theta'(E) \vdash a_1 : \tau_1'/\varphi' \tag{6.4}$$
$$\theta'(E) \oplus \{x : \mathrm{Gen}\,(\tau_1', \theta'(E), \varphi')\} \vdash a_2 : \tau'/\varphi' \tag{6.5}$$

Appliquons l'hypothèse de récurrence sur 6.4, nous déduisons alors qu'il existe une substitution ψ_1 et que :

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(a_1, E, a_1, V) \text{ est défini} \tag{6.6}$$
$$\tau_1' = \psi_1(\tau_1) \tag{6.7}$$
$$\varphi' = \psi_1(\varphi_1) \tag{6.8}$$
$$\theta' =_V \psi_1 \circ \theta_1 \tag{6.9}$$
$$\psi_1 \text{ préserve les sortes} \tag{6.10}$$

En combinant 6.5 avec 6.9 nous avons :

$$\psi_1 \circ \theta_1(E) \oplus \{x : \mathrm{Gen}\,(\tau_1', \psi_1 \circ \theta_1(E), \varphi')\} \vdash a_2 : \tau'/\varphi' \tag{6.11}$$

De 6.11, 6.7 et 6.8 nous déduisons que :

$$\psi_1 \circ \theta_1(E) \oplus \{x : \mathrm{Gen}\,(\psi_1(\tau_1), \psi_1 \circ \theta_1(E), \psi_1(\varphi_1))\} \vdash a_2 : \tau'/\varphi' \tag{6.12}$$

On démontre alors facilement (omis ici afin de ne pas rallonger outre mesure cette preuve) que $\psi_1(\mathrm{Gen}\,(\tau_1, \theta_1(E), \varphi_1))$ est plus général que $\mathrm{Gen}\,(\psi_1(\tau_1), \psi_1 \circ \theta_1(E), \psi_1(\varphi_1)$. Donc par le lemme de stabilité du typage sous hypothèses renforcées (lemme 5.13), appliqué à 6.11 nous avons :

$$\psi_1 \circ \theta_1(E) \oplus \psi_1(\{x : \mathrm{Gen}\,(\tau_1), \theta_1(E), \varphi_1\}) \vdash a_2 : \tau'/\varphi' \tag{6.13}$$

$$\text{c'est-à-dire } \psi_1(\theta_1(E) \oplus \{x : \mathrm{Gen}\,(\tau_1), \theta_1(E), \varphi_1\}) \vdash a_2 : \tau'/\varphi' \tag{6.14}$$

Appliquons alors l'hypothèse de récurrence sur l'expression a_2 et l'environnement de typage $\theta_1(E) \oplus \{x : \mathrm{Gen}\,(\tau_1), \theta_1(E), \varphi_1\}$ avec la substitution ψ_1 (qui préserve les sortes par 6.10). Il vient alors qu'il existe une substitution ψ_2 et que :

$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1) \text{ est défini} \tag{6.15}$$

$$\tau' = \psi_2(\tau_2) \tag{6.16}$$

$$\varphi' = \psi_2(\varphi_2) \tag{6.17}$$

$$\psi_1 =_{V_1} \psi_2 \circ \theta_2 \tag{6.18}$$

$$\psi_2 \text{ préserve les sortes} \tag{6.19}$$

$\diamondsuit$ La déduction 6.15 nous montre que la seconde étape de notre algorithme est bien définie. Il nous reste maintenant à vérifier la dernière, c'est-à-dire prouver qu'il existe un unificateur ψ_3 de $\{\theta_2(\varphi_1) = \varphi_2\}$. Prenons $\psi_3 = \psi_2$. Nous avons $\psi_2 \circ \theta_2(\varphi_1) \overset{6.18}{=} \psi_1(\varphi_1) \overset{6.8}{=} \varphi'$. D'autre part, $\psi_2(\varphi_2)$ est aussi égal à φ' par 6.16. Donc il existe bien un tel unificateur, et l'algorithme $\mathtt{mgu}_V$ étant complet (théorème 6.3), il trouve effectivement l'unificateur principal μ. De plus ψ_3 étant unificateur et μ étant *principal*, nous déduisons que pouvons écrire ψ_3 sous la forme :

$$\psi_3 =_{V_2} \psi_4 \circ \mu \tag{6.20}$$

avec ψ_4 qui préserve les sortes (par complétude et principalité de l'algorithme d'unification (théorème 6.3).

$\diamondsuit$ Ainsi la dernière étape de W n'ayant pas échoué, nous venons de prouver que l'algorithme est effectivement défini dans le cas présent, comme attendu en 6.2.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\tau' = \psi(\mu(\tau_2)) \tag{6.21}$$

$$\varphi' = \psi(\mu(\varphi_2)) \tag{6.22}$$

$$\theta' =_V \psi \circ \mu \circ \theta_2 \circ \theta_1 \tag{6.23}$$

$$\psi \text{ préserve les sortes} \tag{6.24}$$

Prenons $\psi = \psi_4$ trouvée ci-dessus (6.20) et vérifions qu'elle satisfait nos exigences.

- Avec ce choix, $\psi(\mu(\tau_2)) = \psi_4(\mu(\tau_2)) \overset{6.20}{=} \psi_3(\tau_2) \overset{\text{déf de } \psi_3}{=} \psi_2(\tau_2) \overset{6.16}{=} \tau'$, comme attendu en 6.21.
- Avec ce choix, $\psi(\mu(\varphi_2)) = \psi_4(\mu(\varphi_2)) \overset{6.20}{=} \psi_3(\varphi_2) \overset{\text{déf de } \psi_3}{=} \psi_2(\varphi_2) \overset{6.17}{=} \varphi'$, comme attendu en 6.22.
- Soit une variable $v \in V$.

$$\begin{array}{rcll} \psi(\theta(v)) & = & \psi(\mu(\theta_2(\theta_1(v)))) & \text{par définition de } \theta \text{ dans l'algorithme} \\ & = & \psi_4(\mu(\theta_2(\theta_1(v))) & \text{par définition de } \psi \\ & = & \psi_3(\theta_2(\theta_1(v)) & \text{par 6.20, car } FV(\theta_2(\theta_1(v))) \subseteq V_2 \text{ (cf. ci-dessous)} \end{array}$$

En effet, supposons que nous n'ayons pas $FV(\theta_2(\theta_1(v))) \subseteq V_2$. Alors il existerait $\gamma \in FV(\theta_2(\theta_1(v)))$ telle que $\gamma \notin V_2$. Ainsi, γ serait hors de portée de θ_2 puisque par le lemme de propriétés structurelles de W (lemme 6.4) toute variable non présente dans V_2 est hors de portée de θ_2. D'où on aurait $v = \theta_1(\gamma)$. Mais si $\gamma \notin V_2$, alors forcément $\gamma \notin V_1$ puisque par propriétés structurelles de W $V_1 \subseteq V_2$. Or, à nouveau par les propriétés structurelles de W, toute variable non présente dans V_1 est hors de portée de θ_1. Donc, par le lemme 5.6, ça voudrait dire que $v = \gamma$, ce qui est une contradiction car par hypothèse $v \in V \subseteq V_1 \subseteq V_2$.

$$\begin{array}{lll} = & \psi_2(\theta_2(\theta_1(v)) & \text{par définition de } \psi_3 \\ = & \psi_1(\theta_1(v)) & \text{par 6.18 car } FV(\theta_1(v)) \subseteq V_1 \text{ (cf. ci-dessous)} \end{array}$$

En effet, supposons que nous n'ayons pas $FV(\theta_1(v)) \subseteq V_1$. Alors il existerait $\gamma \in FV(\theta_1(v))$ telle que $\gamma \notin V_1$. Ainsi, γ serait hors de portée de θ_1 puisque par le lemme de propriétés structurelles de W (lemme 6.4) toute variable non présente dans V_1 est hors de portée de θ_1. D'où on aurait $v = \gamma$ (par le lemme 5.6), ce qui est une contradiction car par hypothèse $v \in V \subseteq V_1$ et nous venons de voir que $\gamma \notin V_1$.

$$= \quad \theta'(v) \quad \text{comme attendu en 6.23, par 6.9 puisque } v \in V.$$

- Comme attendu en 6.24, ψ préserve les sortes puisque égale à ψ_4 par construction qui les préserve (6.20).

Cas a est `match` a_1 `with` $p \to a_2 \mid x \to a_3$.
Hypothèses :

$$\theta'(E) \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3, V) \text{ est défini} \tag{6.2}$$

$$\exists\, \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. À partir de l'hypothèse 6.1, nous pouvons utiliser une règle de typage 15 de la forme :

$$\frac{\begin{array}{c}\theta'(E) \vdash a_1 : \tau_1'/\varphi' \qquad \vdash p : \tau_1' \Rightarrow E_s \qquad \vdash \tau_1' - p \rightsquigarrow \tau_2' \\ \theta'(E) \oplus E_s \vdash a_2 : \tau'/\varphi' \qquad \theta'(E) \oplus \{x : \tau_2'\} \vdash a_3 : \tau'/\varphi'\end{array}}{\theta'(E) \vdash \texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3 : \tau'/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\theta'(E) \vdash a_1 : \tau_1'/\varphi' \tag{6.4}$$

$$\vdash p : \tau_1' \Rightarrow E_s \tag{6.5}$$

$$\vdash \tau_1' - p \rightsquigarrow \tau_2' \tag{6.6}$$

$$\theta'(E) \oplus E_s \vdash a_2 : \tau'/\varphi' \tag{6.7}$$

$$\theta'(E) \oplus \{x : \tau_2'\} \vdash a_3 : \tau'/\varphi' \tag{6.8}$$

Appliquons l'hypothèse de récurrence sur 6.4, nous déduisons alors qu'il existe une substitution ψ_1 et que :

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(a_1, E, a_1, V) \text{ est défini} \tag{6.9}$$

$$\tau_1' = \psi_1(\tau_1) \tag{6.10}$$

$$\varphi' = \psi_1(\varphi_1) \tag{6.11}$$

$$\theta' =_V \psi_1 \circ \theta_1 \tag{6.12}$$

$$\psi_1 \text{ préserve les sortes} \tag{6.13}$$

$\diamondsuit$ La déduction 6.9 nous montre que la première étape de l'algorithme W est bien définie. Maintenant, prouvons que la seconde étape est également définie. Par 6.10 et 6.5 nous avons : $p : \psi_1(\tau_1) \Rightarrow E_s$. De plus, par 6.10 et 6.6 nous avons : $\psi_1(\tau_1) - p \rightsquigarrow \tau_2'$. Pour finir, par invariants structurels de W (lemme 6.4), nous savons que $FV(\tau_1) \subseteq V_1$. Nous pouvons donc appliquer le théorème de complétude de `Patsubtr` (théorème 6.8) avec la substitution ψ_1 et déduire que :

$$(E', \tau', \xi, V_1') = \texttt{Patsubtr}(p, \tau_1, V_1) \text{ est défini} \tag{6.14}$$

$$\tau_2' = \eta(\tau') \tag{6.15}$$

$$E_s = \eta(E') \tag{6.16}$$

$$\psi_1 =_{V_1} \eta \circ \xi \tag{6.17}$$

$$\eta \text{ et } \xi \text{ préservent les sortes} \tag{6.18}$$

♢ La déduction 6.14 nous montre que la seconde étape de l'algorithme W est bien définie. Maintenant, prouvons que la troisième étape l'est également. En combinant 6.7 et 6.12, nous déduisons que :

$$\psi_1 \circ \theta_1(E) \oplus E_s \vdash a_2 : \tau'/\varphi' \tag{6.19}$$

De 6.17 et 6.19 nous déduisons alors que

$$\eta \circ \xi \circ \theta_1(E) \oplus E_s \vdash a_2 : \tau'/\varphi' \tag{6.20}$$

Or, par 6.16, $E_s = \eta(E')$, donc nous avons :

$$\eta \circ \xi \circ \theta_1(E) \oplus \eta(E') \vdash a_2 : \tau'/\varphi' \tag{6.21}$$

c'est-à-dire :

$$\eta(\xi \circ \theta_1(E) \oplus E') \vdash a_2 : \tau'/\varphi' \tag{6.22}$$

Nous pouvons alors appliquer l'hypothèse de récurrence sur 6.22 avec la substitution η, et déduire qu'il existe une substitution ψ_2 et que :

$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\xi \circ \theta_1(E) \oplus E', a_2, V_1') \text{ est défini} \tag{6.23}$$

$$\tau' = \psi_2(\tau_2) \tag{6.24}$$

$$\varphi' = \psi_2(\varphi_2) \tag{6.25}$$

$$\eta =_{V_1'} \psi_2 \circ \theta_2 \tag{6.26}$$

$$\psi_2 \text{ préserve les sortes} \tag{6.27}$$

♢ La déduction 6.23 nous montre que la troisième étape de l'algorithme W est effectivement définie. Il nous reste maintenant à prouver que sa dernière étape est elle aussi définie. En combinant 6.8 et 6.12, nous déduisons que :

$$\psi_1 \circ \theta_1(E) \oplus \{x : \tau_2'\} \vdash a_3 : \tau'/\varphi' \tag{6.28}$$

Combinant 6.28 et 6.17 nous avons alors :

$$\eta \circ \xi \circ \theta_1(E) \oplus \{x : \tau_2'\} \vdash a_3 : \tau'/\varphi' \tag{6.29}$$

De 6.29 et 6.15, il vient alors :

$$\eta \circ \xi \circ \theta_1(E) \oplus \{x : \eta(\tau')\} \vdash a_3 : \tau'/\varphi' \tag{6.30}$$

qui devient, par 6.26 :

$$\psi_2 \circ \theta_2 \circ \xi \circ \theta_1(E) \oplus \{x : \psi_2 \circ \theta_2(\tau')\} \vdash a_3 : \tau'/\varphi' \tag{6.31}$$

c'est-à-dire :

$$\psi_2(\theta_2 \circ \xi \circ \theta_1(E) \oplus \{x : \theta_2(\tau')\}) \vdash a_3 : \tau'/\varphi' \tag{6.32}$$

Nous pouvons alors appliquer l'hypothèse de récurrence sur 6.32 avec la substitution ψ_2 et nous déduisons qu'il existe une substitution ψ_3 et que :

$$(\tau_3, \varphi_3, \theta_3, V_3) = W(\theta_2 \circ \xi \circ \theta_1(E) \oplus \{x : \theta_2(\tau')\}, a_3, V_2) \text{ est défini} \tag{6.33}$$
$$\tau' = \psi_3(\tau_3) \tag{6.34}$$
$$\varphi' = \psi_3(\varphi_3) \tag{6.35}$$
$$\psi_2 =_{V_2} \psi_3 \circ \theta_3 \tag{6.36}$$
$$\psi_3 \text{ préserve les sortes} \tag{6.37}$$

◊ La déduction 6.33 nous montre que la quatrième étape l'algorithme W est effectivement définie. Pour terminer la preuve que celui-ci est effectivement défini, il nous reste maintenant à montrer que sa dernière étape n'échoue pas, c'est-à-dire qu'il existe un unificateur ψ_4 de $\{\theta_3(\tau_2) = \tau_3;\ \theta_3(\varphi_2) = \varphi_3;\ \theta_3(\theta_2(\xi(\varphi_1))) = \varphi_3\}$.

Prenons $\psi_4 = \psi_3$ trouvée ci-dessus. Nous avons alors :

- $\psi_4(\theta_3(\tau_2)) = \psi_3(\theta_3(\tau_2)) \overset{6.36}{=} \psi_2(\tau_2) \overset{6.24}{=} \tau'$. De plus $\psi_4(\tau_3) = \psi_3(\theta_3(\tau_3) \overset{6.34}{=} \tau'$ également. Donc ce choix pour ψ_4 satisfait la première équation.
- $\psi_4(\theta_3(\varphi_2)) = \psi_3(\theta_3(\varphi_2)) \overset{6.36}{=} \psi_2(\varphi_2)) \overset{6.25}{=} \varphi'$. De plus $\psi_4(\varphi_3) = \psi_3(\varphi_3) \overset{6.35}{=} \varphi'$ également. Donc ce choix pour ψ_4 satisfait la seconde équation.
- $\psi_4(\theta_3(\theta_2(\xi(\varphi_1)))) = \psi_3(\theta_3(\theta_2(\xi(\varphi_1)))) \overset{6.36}{=} \psi_2(\theta_2(\xi(\varphi_1))) \overset{6.26}{=} \eta(\xi(\varphi_1)) \overset{6.17}{=} \psi_1(\varphi_1) \overset{6.11}{=} \varphi'$. De plus $\psi_4(\varphi_3) = \psi_3(\varphi_3) \overset{6.35}{=} \varphi'$ également. Donc ce choix pour ψ_4 satisfait aussi la dernière équation de notre ensemble.

Donc il existe bien un tel unificateur, et l'algorithme $\mathtt{mgu}_V$ étant complet (théorème 6.3), il trouve effectivement l'unificateur principal μ. De plus ψ_4 étant unificateur et μ étant *principal*, nous déduisons que pouvons écrire ψ_4 sous la forme :

$$\psi_4 =_{V_3} \psi_5 \circ \mu \tag{6.38}$$

◊ Ainsi, la dernière étape de l'algorithme W étant effectivement définie, celui-ci n'échoue pas dans le cas présent, comme attendu en 6.2.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\tau' = \psi(\mu(\tau_3)) \tag{6.39}$$
$$\varphi' = \psi(\mu(\varphi_3)) \tag{6.40}$$
$$\theta' =_V \psi \circ \mu \circ \theta_3 \circ \theta_2 \circ \xi \circ \theta_1 \tag{6.41}$$
$$\psi \text{ préserve les sortes} \tag{6.42}$$

Prenons $\psi = \psi_5$ et vérifions qu'elle satisfait nos exigences.

- Avec ce choix, $\psi(\mu(\tau_3)) = \psi_5 \circ \mu(\tau_3) \overset{6.38}{=} \psi_4(\tau_3) = \psi_3(\tau_3)$ par définition de ψ_4. Or par 6.34 $\psi_3(\tau_3) = \tau'$ comme attendu en 6.39.
- Avec ce choix, $\psi(\mu(\varphi_3)) = \psi_5 \circ \mu(\varphi_3) \overset{6.38}{=} \psi_4(\varphi_3) = \psi_3(\varphi_3)$ par définition de ψ_4. Or par 6.35 $\psi_3(\varphi_3) = \varphi'$ comme attendu en 6.40.
- Soit une variable $v \in V$.

$$\begin{array}{rcll}
\psi(\theta(v)) & = & \psi(\mu(\theta_3(\theta_2(\xi(\theta_1(v)))))) & \text{par définition de } \theta \text{ dans l'algorithme} \\
& = & \psi_5(\mu(\theta_3(\theta_2(\xi(\theta_1(v)))))) & \text{par définition de } \psi \\
& = & \psi_4(\theta_3(\theta_2(\xi(\theta_1(v))))) & \text{par 6.38 car } FV(\theta_3(\theta_2(\xi(\theta_1(v))))) \subseteq V_3 \\
& = & \psi_3(\theta_3(\theta_2(\xi(\theta_1(v))))) & \text{par définition de } \psi_4 \\
& = & \psi_2(\theta_2(\xi(\theta_1(v)))) & \text{par 6.36 car } FV(\theta_2(\xi(\theta_1(v)))) \subseteq V_2 \\
& = & \eta(\xi(\theta_1(v))) & \text{par 6.26 car } FV(\xi(\theta_1(v))) \subseteq V_1' \\
& = & \psi_1(\theta_1(v)) & \text{par 6.17 car } FV(\theta_1(v)) \subseteq V_1 \\
& = & \theta'(v) & \text{comme attendu en 6.41, par 6.12 puisque } v \in V.
\end{array}$$

- Par construction, ψ étant égale à ψ_5 et ψ_5 préservant les sortes par complétude et principalité de l'algorithme d'unification (théorème 6.3), nous avons bien que ψ préserve les sortes comme attendu en 6.42.

Cas a est C.

On se convaincra aisément que ce cas est traité de la même façon que a est i, en remplaçant le constructeur de type `int` par `exn`.

Cas a est $D(a_1)$.

Hypothèses :

$$\theta'(E) \vdash D(a_1) : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, D(a_1), V) \text{ est défini} \tag{6.2}$$
$$\exists\, \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. Le seul moyen de dériver 6.1, est d'utiliser une règle de typage 17 de la forme :

$$\frac{\tau'_1 \leq \text{TypeArg}(D) \qquad \theta'(E) \vdash a_1 : \tau'_1/\varphi' \qquad \vdash \varphi'_1 :: \texttt{EXN}(\{D\}) \qquad \vdash \varphi' :: \texttt{EXN}(\emptyset)}{\theta'(E) \vdash D(a_1) : \texttt{exn}[D(\tau'_1);\ \varphi'_1]/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\tau'_1 \leq \text{TypeArg}(D) \tag{6.4}$$
$$\theta'(E) \vdash a_1 : \tau'_1/\varphi' \tag{6.5}$$
$$\vdash \varphi'_1 :: \texttt{EXN}(\{D\}) \tag{6.6}$$
$$\vdash \varphi' :: \texttt{EXN}(\emptyset) \tag{6.7}$$
$$\vdash \tau' \text{ est de la forme } \texttt{exn}[D(\tau'_1);\ \varphi'_1] \tag{6.8}$$

Appliquons l'hypothèse de récurrence sur 6.5. Nous en déduisons alors qu'il existe une substitution ψ_1 et que :

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V) \text{ est défini} \tag{6.9}$$
$$\tau'_1 = \psi_1(\tau_1) \tag{6.10}$$
$$\varphi' = \psi_1(\varphi_1) \tag{6.11}$$
$$\theta' =_V \psi_1 \circ \theta_1 \tag{6.12}$$
$$\psi_1 \text{ préserve les sortes} \tag{6.13}$$

♢ La déduction 6.9 montre que la première étape de l'algorithme W est définie. Nous devons maintenant nous assurer que $\texttt{Inst}(\mathit{TypeArg}(D), V_1)$ est bien définie. Puisque par 6.4 $\tau'_1 \leq \text{TypeArg}(D)$, nous pouvons appliquer le théorème de complétude de `Inst` (théorème 6.7) et déduire que :

$$(\tau_2, V_2) = \texttt{Inst}(\text{TypeArg}(D)) \text{ est défini} \tag{6.14}$$
$$\tau'_1 = \xi(\tau_2) \tag{6.15}$$
$$\xi = \{\alpha \leftarrow \alpha'\} \text{ où } \alpha' \notin V_1 \tag{6.16}$$

♢ Ainsi la seconde étape de l'algorithme n'échoue pas et nous devons maintenant montrer qu'il existe un unificateur ψ_2 de $\{\tau_2 = \tau_1\}$. Prenons $\psi_2 = \psi_1 \oplus \xi \oplus \{\rho \leftarrow \varphi'_1\}$. Cette substitution ψ_2 préserve bien les sortes car, par 6.13, ψ_1 les préserve et comme il est toujours possible de choisir ρ de sorte adéquate, la substitution $\{\rho \leftarrow \varphi'_1\}$ les préserve également.

Puisque, par 6.16 $\alpha' \notin V_1$ et que par le lemme de propriétés structurelles de W (lemme 6.4) $FV(\tau_1) \subseteq V_1$, nous déduisons que :

$$\alpha' \notin \tau_1 \tag{6.17}$$

De même, puisque $\rho \notin V_2$, alors $\rho \notin V_1$. Nous savons également que $FV(\tau_1) \subseteq V_1$, $FV(\varphi_1) \subseteq V_1$ et $FV(\tau_2) \subseteq V_2$, $FV(\varphi_2) \subseteq V_2$ (par lemme de propriétés structurelles de W 6.4) donc :

$$\rho \notin \tau_1 \text{ et } \rho \notin \varphi_1 \tag{6.18}$$
$$\rho \notin \tau_2 \text{ et } \rho \notin \varphi_2 \tag{6.19}$$

Vérifions maintenant le comportement de ψ_2 sur l'ensemble d'équations à unifier :

- $\psi_2(\tau_1) = (\psi_1 \oplus \xi)(\tau_1) \overset{6.16+6.17+6.18}{=} \psi_1(\tau_1) \overset{6.10}{=} \tau_1'$
- $\psi_2(\tau_2) = (\psi_1 \oplus \xi)(\tau_2)$. Les variables de τ_2 appartenant toutes à Dom (ξ) et ρ n'appartenant pas à τ_2 (par 6.19) , nous en déduisons que c'est égal à $\xi(\tau_2) \overset{6.15}{=} \tau_1'$

Donc il existe bien un unificateur ψ_2 de notre ensemble d'équations et l'algorithme $\mathtt{mgu}_V$ étant complet, il trouve effectivement μ comme attendu. Qui plus est, μ étant l'unificateur principal, nous pouvons écrire :

$$\psi_2 =_{V_2} \psi_3 \circ \mu \tag{6.20}$$

avec ψ_3 préservant les sortes (par complétude et principalité de l'algorithme d'unification).

$\diamondsuit$ Ainsi la troisième étape de notre algorithme W est bien définie. Les étapes restant ne peuvent pas échouer puisqu'il est toujours possible de choisir des variables fraîches et que le rendu du résultat est trivialement défini. Aussi nous avons prouvé que W est totalement défini dans le cas présent.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\mathtt{exn}[D(\tau_1'); \varphi_1'] = \psi(\mu(\mathtt{exn}[D(\tau_1); \rho])) \tag{6.21}$$
$$\varphi' = \psi \circ \mu(\varphi_1) \tag{6.22}$$
$$\theta' =_V \psi \circ \mu \circ \theta_1 \tag{6.23}$$
$$\psi \text{ préserve les sortes} \tag{6.24}$$

Prenons $\psi = \psi_3$ et vérifions qu'elle satisfait nos exigences.

- $\psi(\mu(\mathtt{exn}[D(\tau_1)); \rho]) = \psi_3(\mu(\mathtt{exn}[D(\tau_1)); \rho]) \overset{6.20}{=} \psi_2(\mathtt{exn}[D(\tau_1)); \rho]) = (\psi_1 \oplus \xi \oplus \{\rho \leftarrow \varphi_1'\})(\mathtt{exn}[D(\tau_1)); \rho]) \overset{6.18}{=} \mathtt{exn}[D(\psi_1(\tau_1)); \varphi_1'] \overset{6.10}{=} \mathtt{exn}[D(\tau_1'); \varphi_1']$ comme attendu en 6.21.
- $\psi \circ \mu(\varphi_1) = \psi_3 \circ \mu(\varphi_1) \overset{6.20}{=} \psi_2(\varphi_1) = (\psi_1 \oplus \xi \oplus \{\rho \leftarrow \varphi_1'\})(\varphi_1) \overset{6.18}{=} \psi_1(\varphi_1) \overset{6.11}{=} \varphi'$ comme attendu en 6.22.
- Soit une variable $v \in V$.

$$\begin{array}{rcll} \psi(\theta(v)) & = & \psi(\mu(\theta_1(v))) & \text{par définition de } \theta \text{ dans l'algorithme} \\ & = & \psi_3(\mu(\theta_1(v))) & \text{par choix de } \psi \\ & = & \psi_2(\theta_1(v)) & \text{par 6.20 car } FV(\theta_1(v)) \subseteq V_2 \\ & = & (\psi_1 \oplus \xi \oplus \{\rho \leftarrow \varphi_1'\}) \circ \theta_1(v) & \text{par définition de } \psi_2 \end{array}$$

Par le lemme de propriétés structurelles de W (lemme 6.4), nous savons que toutes les variables n'appartenant pas à V_1 sont hors de portée de θ_1. De plus, v n'étant égale ni à ρ, ni à un α du domaine de ξ, nous en déduisons alors que :

$$\begin{array}{cll} = & \psi_1 \circ \theta_1(v) & \text{par la remarque ci-dessus} \\ = & \theta' & \text{par 6.12 comme attendu en 6.23.} \end{array}$$

- ψ étant égale par construction à ψ_3 et cette dernière préservant les sortes (par 6.20), nous déduisons que ψ les préserve bien comme attendu en 6.24.

Cas a est `try` a_1 `with` $x \to a_2$.

Hypothèses :

$$\theta'(E) \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau'/\varphi' \tag{6.1}$$

Nous voulons prouver que :

$$(\tau, \varphi, \theta, V') = W(E, \texttt{try}\ a_1\ \texttt{with}\ x \to a_2, V) \text{ est défini} \tag{6.2}$$

$$\exists \psi \text{ préservant les sortes tq } \tau' = \psi(\tau) \text{ et } \varphi' = \psi(\varphi) \text{ et } \theta' =_V \psi \circ \theta \tag{6.3}$$

Commençons par démontrer que l'algorithme W est défini. À partir l'hypothèse 6.1, nous pouvons utiliser une règle de typage 18 de la forme :

$$\frac{\theta'(E) \vdash a_1 : \tau'/\varphi_1' \qquad \theta'(E) \oplus \{x : \texttt{exn}[\varphi_1']\} \vdash a_2 : \tau'/\varphi'}{\theta'(E) \vdash \texttt{try}\ a_1\ \texttt{with}\ x \to a_2 : \tau'/\varphi'}$$

Nous en déduisons donc que nous avons :

$$\theta'(E) \vdash a_1 : \tau'/\varphi_1' \tag{6.4}$$

$$\theta'(E \oplus \{x : \texttt{exn}[\varphi_1']\} \vdash a_2 : \tau'/\varphi' \tag{6.5}$$

Appliquons l'hypothèse de récurrence sur 6.4. Nous en déduisons alors qu'il existe une substitution ψ_1 et que :

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V) \text{ est défini} \tag{6.6}$$

$$\tau' = \psi_1(\tau_1) \tag{6.7}$$

$$\varphi_1' = \psi_1(\varphi_1) \tag{6.8}$$

$$\theta' =_V \psi_1 \circ \theta_1 \tag{6.9}$$

$$\psi_1 \text{ préserve les sortes} \tag{6.10}$$

$\diamondsuit$ La déduction 6.6 montre que la première étape de l'algorithme W est définie. Nous devons maintenant nous assurer que la seconde étape l'est également.

Combinons 6.5 et 6.8, nous déduisons alors que :

$$\theta'(E) \oplus \{x : \texttt{exn}[\psi_1(\varphi_1)]\} \vdash a_2 : \tau'/\varphi' \tag{6.11}$$

De 6.11 et 6.9, nous avons :

$$\psi_1 \circ \theta_1(E) \oplus \{x : \texttt{exn}[\psi_1(\varphi_1)]\} \vdash a_2 : \tau'/\varphi' \tag{6.12}$$

c'est-à-dire :

$$\psi_1(\theta_1(E) \oplus \{x : \texttt{exn}[\varphi_1]\}) \vdash a_2 : \tau'/\varphi' \tag{6.13}$$

Appliquons alors l'hypothèse de récurrence avec la substitution ψ_1 et il vient qu'il existe une substitution ψ_2 et que :

$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E) \oplus \{x : \texttt{exn}[\varphi_1]\}), a_2, V_1) \text{ est défini} \tag{6.14}$$

$$\tau' = \psi_2(\tau_2) \tag{6.15}$$

$$\varphi' = \psi_2(\varphi_2) \tag{6.16}$$

$$\psi_1 =_{V_1} \psi_2 \circ \theta_2 \tag{6.17}$$

$$\psi_2 \text{ préserve les sortes} \tag{6.18}$$

$\diamondsuit$ La déduction 6.14 montre que la seconde étape de l'algorithme W est définie. Pour terminer, nous devons désormais montrer qu'il existe un unificateur ψ_3 de l'ensemble d'équations $\{\theta_2(\tau_1) = \tau_2\}$. Prenons $\psi_3 = \psi_2$ trouvée ci-dessus. Vérifions-en alors le comportement sur notre équation.

- Avec ce choix, $\psi_3(\theta_2(\tau_1)) = \psi_2(\theta_2(\tau_1)) \overset{6.17}{=} \psi_1(\tau_1) \overset{6.7}{=} \tau'$
- Avec ce choix, $\psi_3(\tau_2) = \psi_2(\tau_2) \overset{6.15}{=} \tau'$

Donc il existe bien un tel unificateur, et l'algorithme $\texttt{mgu}_V$ étant complet (théorème 6.3), il trouve effectivement l'unificateur principal μ. De plus ψ_3 étant unificateur et μ étant *principal*, nous déduisons que pouvons écrire ψ_3 sous la forme :

$$\psi_3 =_{V_2} \psi_4 \circ \mu \tag{6.19}$$

avec ψ_4 préservant les sortes (par complétude et principalité de l'algorithme d'unification). $\diamondsuit$ Ainsi la troisième étape de notre algorithme W est bien définie. La dernière n'échoue pas trivialement, montrant ainsi que W est bien défini dans le cas présent.

Nous devons maintenant, pour satisfaire le but 6.3, montrer l'existence, de la substitution ψ qui doit alors satisfaire les propriétés suivantes :

$$\tau' = \psi \circ \mu(\tau_2) \tag{6.20}$$
$$\varphi' = \psi \circ \mu(\varphi_2) \tag{6.21}$$
$$\theta' =_V \psi \circ \mu \circ \theta_2 \circ \theta_1 \tag{6.22}$$
$$\psi \text{ préserve les sortes} \tag{6.23}$$

Prenons $\psi = \psi_4$ et vérifions qu'elle satisfait nos exigences.

- $\psi \circ \mu(\tau_2) = \psi_4 \circ \mu(\tau_2) = \psi_3(\tau_2) = \psi_2(\tau_2) \overset{6.15}{=} \tau'$
- $\psi \circ \mu(\varphi_2) = \psi_4 \circ \mu(\varphi_2) = \psi_3(\varphi_2) = \psi_2(\varphi_2) \overset{6.16}{=} \varphi'$
- Soit une variable $v \in V$.

$$\begin{array}{rcll}
\psi(\theta(v)) & = & \psi(\mu(\theta_2(\theta_1(v)))) & \text{par définition de } \theta \text{ dans l'algorithme} \\
& = & \psi_4(\mu(\theta_2(\theta_1(v)))) & \text{par choix de } \psi \\
& = & \psi_3(\theta_2(\theta_1(v))) & \text{par 6.19 car } FV(\theta_2(\theta_1(v))) \subseteq V_2 \\
& = & \psi_2(\theta_2(\theta_1(v))) & \text{par définition de } \psi_3 \\
& = & \psi_1(\theta_1(v)) & \text{par 6.17 car } FV(\theta_1(v)) \subseteq V_1 \\
& = & \theta' & \text{comme attendu en 6.22, par 6.9 puisque } v \in V.
\end{array}$$

- ψ étant égale par construction à ψ_4 et cette dernière préservant les sortes (par 6.19), nous déduisons que ψ les préserve bien comme attendu en 6.23.

$\square$

Conclusion : Nous concluons cette section par un corollaire établissant que l'algorithme W que nous avons décrit prédit un ensemble minimal d'exceptions vis-à-vis de la spécification de notre analyse.

Théorème 6.10 (Complétude de l'analyse vis-à-vis du système de types) *Si $\emptyset \vdash a : \tau'/\varphi'$ alors $(\tau, \varphi, \theta, V') = W(\emptyset, a, \emptyset)$ est défini et toute exception appartenant à la rangée φ appartient également à φ'.*

Preuve : Par le théorème de complétude de l'algorithme W (6.9), nous avons $\varphi' = \psi \circ (\varphi)$ pour une certaine substitution ψ. Il est clair alors que toute exception v appartenant à φ appartient aussi à $\psi(\varphi)$ (qui est égal à φ'). $\square$

En l'absence de cette propriété de complétude, notre analyse donnerait quand même des résultats corrects. Toutefois, il est satisfaisant de voir que le système d'inférence que nous avons décrit reflète exactement notre analyse. Autrement dit, l'implémentation de l'analyse et ses spécifications formelles coïncident exactement.

Chapitre 7

Extension à un langage complet

Au cours des chapitres précédents, nous avons étudié le principe de notre analyse et montré sa correction ainsi que sa complétude vis-à-vis des règles de typage et de la sémantique dynamique d'un petit langage ML. Cette analyse étant destinée à opérer sur le langage *Objective Caml*, il est maintenant nécessaire de l'étendre de façon à gérer toutes les constructions de ce langage. En particulier nous allons étudier cinq extensions importantes, à savoir la gestion des définitions de types utilisateur avec les cas délicats des types récursifs (types cycliques), l'analyse des n-uplets et des enregistrements, la prise en compte des valeurs mutables (références), l'intégration au système de modules et enfin l'analyse des aspects objets du langage.

Comme nous le verrons, certaines de ces extensions s'intègrent aisément dans notre formalisme, d'autres nécessitent un réel effort d'implantation, mais toutes entrent élégamment dans le cadre de notre analyse.

7.1 Définitions de types utilisateur

Dans cette section, nous allons examiner les différentes stratégies possibles d'approximation des types concrets permettant la recherche d'un compromis entre efficacité et précision de notre analyse.

7.1.1 Types concrets

Caml fournit un certain nombre de types de base :

- `int` pour les valeurs entières,
- `bool` pour les valeurs de vérité,
- `float` pour les flottants,
- `string` pour les chaînes de caractères,
- `char` pour les caractères,

ainsi que les types composés n-uplets (constructeur $\times$) et les fonctions (constructeur $\to$).

Par ailleurs, il offre à l'utilisateur la possibilité de définir de nouveaux types dont il spécifiera les valeurs : les *types concrets* ou *types sommes*. Ainsi est-il possible de définir un type `t` dont les valeurs seront `A`, `B` ou `C` colportant une valeur entière précisant cette valeur `C`, par la définition suivante :

```
type t = A | B | C of int
```

De telles définitions de types peuvent également être récursives comme l'illustre le cas suivant des listes d'entiers :

```
type intlist = Nil | Cons of (int * intlist)
```

et éventuellement paramétrées comme la définition des listes polymorphes :

```
type α list = [] | :: of (α * α list)
```

Ces définitions de types peuvent être approximées de différentes manières selon le compromis désiré entre rapidité et précision de l'analyse. Nous avons considéré quatre approches dans ce travail, approches que nous allons étudier ci-dessous dans un ordre de précision décroissante (et donc de vitesse d'analyse croissante) et qui se trouvent illustrées par le schéma 7.1. Dans ce qu'il suit, nous supposons que la fonction TypeArg servant initialement à retrouver le type de l'argument d'un constructeur d'exceptions (cf. section 4.2) est désormais étendue pour englober également les constructeurs de types concrets.

7.1.2 Approximation complète

La première approche consiste à appliquer à ces structures de données le même traitement que celui utilisé pour approximer le type `exn` des exceptions. De même que ce type était annoté par une rangée dont les éléments étaient soit des constructeurs d'exceptions constants avec annotation de présence, soit des constructeurs paramétrés colportant le type annoté de leurs arguments, nous annoterons le nom d'un type concret par une rangée φ composée de constructeurs de ce type. Les constructeurs constants porteront également une information de présence, et les constructeurs paramétrés seront accompagnés des types de leurs arguments.

Considérons le texte source de la définition de type :

$$\texttt{type}\ \vec{\alpha}\ \texttt{t} = C_1 \mid \ldots \mid C_n \mid D_1 \texttt{of}\ \mu_1 \mid \ldots \mid D_m \texttt{of}\ \mu_m$$

où μ_i représentent des types ML non annotés. La propagation des exceptions, et plus généralement des valeurs, est capturée en assignant les schémas de types suivants aux constructeurs C_i and D_i :

$$\begin{array}{lcl} C_i & : & \forall \vec{\alpha}, \rho'.\, \vec{\alpha}\ \texttt{t}[C_i : \texttt{Pre};\ \rho'] \\ D_i & : & \forall \vec{\alpha}, \vec{\rho}, \rho', \rho''.\, \tau_i \xrightarrow{\rho''} \vec{\alpha}\ \texttt{t}[D_i(\tau_i);\ \rho'] \end{array}$$

où τ_i sont les types annotés obtenus à partir des μ_i par ajout d'une variable de rangée fraîche et distincte à tout constructeur de type présent dans μ_i *et pouvant porter une annotation de rangée*. En effet, nous verrons que certains constructeurs de types (c'est-à-dire noms de types) peuvent ne pas se trouver annotés si nécessaire, soit dans le cas de types abstraits, soit par choix d'approximation. Ainsi la déclaration de type des listes d'entiers vue précédemment (c'est-à-dire `type intlist = Nil | Cons of (int * intlist)` générera deux constructeurs `Nil` et `Cons` auxquels nous assignerons les schémas de types suivants :

$$\begin{array}{rcl} \texttt{Nil} & : & \forall \rho.\, \texttt{intlist}[\texttt{Nil} : \texttt{Pre};\ \rho] \\ \texttt{Cons} & : & \forall \rho_1, \rho_2, \rho_3, \rho_4.\, (\texttt{int}[\rho_1] \times \texttt{intlist}[\rho_2]) \xrightarrow{\rho_3} \\ & & \qquad \texttt{intlist}[\texttt{Cons}(\texttt{int}[\rho_1] \times \texttt{intlist}[\rho_2]);\ \rho_4] \end{array}$$

Les structures de données récursives telles que `intlist` donnent naturellement lieu à des expressions de type récursives. Considérons :

```
let tail x =
  match x with
  | Cons (hd, tl) → tl
  | l → l
```

Durant l'inférence de types, `tl` et `l` vont respectivement recevoir les types $\texttt{intlist}[\rho_1]$ et $\texttt{intlist}[\texttt{Cons}(\texttt{int}[\rho_2] \times \texttt{intlist}[\rho_1]);\ \rho_3]$. Si seules les expressions de types finies sont autorisées, ces deux types n'auront pas d'unificateur principal (échec de l'unification par *occur-check*, c'est-à-dire unification d'une variable de type avec un terme contenant cette même variable) et le programme sera rejeté par l'analyse. Ceci n'est pas acceptable pour au moins deux raisons : tout d'abord ce programme est bien typé dans le système de ML conventionnel, ensuite il est inconcevable que notre analyse ne puisse pas traiter des programmes de cette simplicité. Aussi avons-nous trouvé une seconde raison d'étendre notre

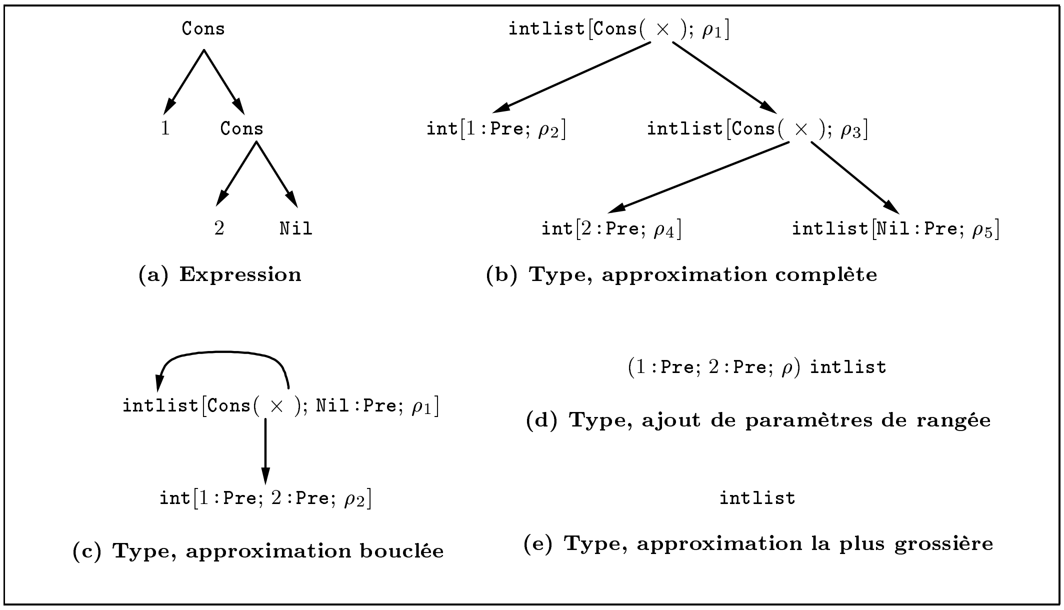


FIG. 7.1 – Exemples d'approximations du type `intlist = Nil | Cons of int * intlist`

système de types par des expressions de types récursives (cycliques, infinies, régulières). Sur l'exemple ci-dessus, nous obtenons alors $\mu\,\alpha.\,\mathtt{intlist}[\mathtt{Cons}(\mathtt{int}[\rho_2] \times \alpha);\ \rho_3]$.

Rappelons, comme mentionné au chapitre 4.4.1, que l'extension de notre algèbre par les types cycliques implique principalement le remplacement de l'unification entre termes par une unification entre graphes au sein de l'algorithme d'inférence.

Nous pouvons désormais remarquer que le type des exceptions peut être vu comme un cas particulier de type concret *extensible* (l'utilisateur peut rajouter un nouveau constructeur à tout moment par déclaration d'une nouvelle exception) dont la signature est toutefois finie. En effet, il est possible de déterminer statiquement le nombre de constructeurs de ce type par lecture du programme tout entier. Ce type est également récursif puisque nous pouvons définir un constructeur d'exceptions dont l'argument est lui-même de type `exn` (par exemple `exception Foo of exn`).

7.1.3 Approximation bouclée

Le schéma d'approximation que nous avons étudié ci-dessus possède l'indésirable effet d'enregistrer, dans l'approximation d'un type, la structure complète de la valeur analysée. Dans le cas d'une structure de donnée donnée *in extenso* ceci peut aboutir à une approximation de très grande taille, tout particulièrement dans le cas de structures de données récursives.

En nous appuyant sur l'exemple du type `intlist` vu précédemment, considérons l'expression : $\ell_n = \mathtt{Cons}(i_1, \mathtt{Cons}(i_2, \ldots, \mathtt{Cons}(i_n, \mathtt{Nil}) \ldots))$. Avec le schéma de types donné ci-dessus au constructeur `Cons`, cette expression va recevoir un type annoté dont la profondeur sera égale à n et enregistra non seulement le fait que la liste contient les entiers $i_1 \ldots i_n$ (une information pouvant en effet être très utile à notre analyse d'exceptions), mais aussi le fait que la liste est de longueur n, que son premier élément est i_1, le second i_2, etc...(cf. schéma 7.1) Ces dernières informations sont de toute évidence inutiles à notre analyse d'exceptions, voire même préjudiciable puisque d'aussi grandes approximations ont pour effet de ralentir l'analyse, la quantité d'informations dans les rangées demandant un surcroît de travail lors des traitements sur les types.

Une solution à ce problème vient de la remarque suivante : dès que ces énormes structures de données sont communiquées à une fonction suffisamment complexe (c'est-à-dire à peu près n'importe quelle fonction d'un programme réel : le cas de la fonction identité et de ses dérivés triviaux étant couramment utilisés à titre d'exemples pour leur simplicité mais ne représentant pas la classe des fonctions que l'on peut trouver dans des applications de taille réelle), leur type annoté « déplié » va se retrouver rapidement unifié avec un terme de type récursif, forçant ainsi toute l'information se trouvant expansée dans le gros type de départ à être repliée au sein d'un type cyclique ainsi plus petit. Par exemple, si nous transmettons la liste ℓ_n à la fonction `tail` vue précédemment, le type de cette liste va se retrouver unifié en :

$$\tau_n = \mu\alpha.\, \mathtt{intlist}[\mathtt{Cons}(\mathtt{int}[i_1 : \mathtt{Pre};\, \ldots;\, i_n : \mathtt{Pre};\, \rho_1] \times \alpha);\, \mathtt{Nil} : \mathtt{Pre};\, \rho_2]$$

L'idée est ainsi de forcer ce repliage en un type récursif lorsque la structure de donnée est créée en donnant un type récursif déjà enroulé aux constructeurs de ce type. Ceci peut être aisément réalisé en unifiant, dans le type des constructeurs, toutes les occurrences du type défini récursivement se trouvant en position d'argument avec une occurrence du type défini récursivement en position résultat. Par exemple, dans le cas du constructeur `Cons` du type `intlist`, nous partons du type :

$$\mathtt{int}[\rho_1] \times \underline{\mathtt{intlist}[\rho_2]} \xrightarrow{\rho_3} \underline{\mathtt{intlist}[\mathtt{Cons}(\mathtt{int}[\rho_1] \times \mathtt{intlist}[\rho_2]);\, \rho_4]}$$

comme dans la stratégie précédente, puis nous unifions les deux types `intlist` soulignés et enfin généralisons les variables libres. Nous obtenons alors :

$$\mathtt{Cons} : \forall \rho_1, \rho_3, \rho_4.\, \mathtt{int}[\rho_1] \times \tau \xrightarrow{\rho_3} \tau$$

où τ est $\mu\alpha.\, \mathtt{intlist}[\mathtt{Cons}(\mathtt{int}[\rho_1] \times \alpha);\, \rho_4]$. Avec ce schéma de types pour `Cons`, la liste ℓ_n possède maintenant le type τ_n raisonnablement compact annoncé précédemment.

Cette technique d'approximation *bouclée* (ou repliée) fonctionne également dans le cadre des définitions de types paramétrés tant que celles-ci restent *régulières*, c'est-à-dire tant que le type défini récursivement est utilisé à l'intérieur de sa définition avec *les mêmes arguments que ceux fournis pour cette définition.* Ainsi la définition des listes polymorphes présentée précédemment est-elle régulière alors que :

```
type 'a nreg = Leaf of 'a | Node of 'a list nreg
```

ne l'est pas puisque le type `nreg`, défini avec α comme argument, est récursivement utilisé avec α `list` à l'intérieur de sa propre définition (cas du constructeur `Node`).

La présence de structures de données non régulières peut mener à deux cas de figures possibles. Dans l'exemple ci-dessus le processus de bouclage forçant les unifications de `t` dans le type du constructeur `Node` rendra essentiellement ce constructeur inutilisable en pratique (impossibilité de créer des valeurs de ce type), lui donnant le type $\tau \xrightarrow{\rho_0} \tau$ où $\tau = \tau'\ \mathtt{nreg}[\mathtt{Leaf}(\tau');\, \mathtt{Node}(\tau);\, \rho_1]$ et $\tau' = \tau'\ \mathtt{list}[\rho_2]$.

Il est également possible de construire des exemples dans lesquels il sera simplement impossible d'effectuer ce processus. Par exemple :

```
type 'a clist = Foo of (int clist) clist
```

Lorsque nous allons tenter d'unifier ensemble toutes les occurrences du type `clist` pour ce constructeur `Foo`, nous devrons unifier leurs arguments, et plus précisément `int` et `int clist` ce qui résultera bien évidemment en une erreur. Ainsi une telle définition de type sera rejetée par notre analyse.

Fort heureusement de telles structures de données non régulières sont extrêmement rares dans les programmes réels. Nous pouvons ainsi décider de les approximer de façon complète, comme étudié dans la section précédente, sans pour autant pénaliser les performances de notre analyse.

7.1.4 Ajout de paramètres de rangée

Une autre alternative à l'annotation des types concrets par des rangées consiste à rajouter des paramètres de rangée à ces types afin de refléter les annotations de rangées des types `exn`, `int` et des types fonctionnels contenus dans les structures de données représentées par les types concrets. Cette technique est effectivement utilisée par Fähndrich *et al* [9] dans leur analyse d'exceptions. Par exemple, la définition de type ML suivante :

```
type t = A of int | B of exn | C of t | D of t
```

doit être transformée en :

$\texttt{type}\ (\rho_1, \rho_2)\ \texttt{t = A of int}[\rho_1]\ \texttt{| B of exn}[\rho_2]\ \texttt{| C of}\ (\rho_1, \rho_2)\ \texttt{t | D of}\ (\rho_1, \rho_2)\ \texttt{t}$

Deux paramètres ρ_1 et ρ_2 sont ajoutés afin de refléter dans le type `t`, les valeurs possibles des types `int` et `exn` contenus dans ce type (via les constructeurs `A` et `B`).

Le type `t` n'est ainsi plus lui-même annoté par une rangée mémorisant quels constructeurs `A`, `B`, `C` ou `D` sont présents dans les valeurs de ce type. Ceci présente la claire caractéristique d'oublier la structure des termes de type `t` tout en se rappelant correctement les valeurs d'entiers et d'exceptions pouvant se trouver encapsulées dans cette structure.

Dans la pratique, cette solution semble être légèrement moins précise et un peu plus efficace que l'approximation complète des structures de données non récursives combinée avec l'approximation « bouclée » de celles qui le sont. En effet, les expressions de types deviennent alors plus petites, mais dans le cas du type `t` ci-dessus, des approximations bouclées seraient capables d'exprimer le fait qu'une valeur de type `t` peut ne contenir ni `C` ni `D`, alors que ceci ne serait pas capturé par une approche basée sur l'ajout de paramètres de rangée.

Il devient maintenant impossible d'appliquer notre opération de soustraction lors des filtrages sur des types non annotés par des rangées puisque nous n'avons plus d'approximation de la structure des valeurs de ces types. Ainsi, dans une telle approche nous considérerions simplement que notre opération de soustraction se réduit à la relation d'identité sur ces types.

7.1.5 Approximation la plus grossière

Dans le cas où nous désirons une rapidité maximale de l'analyse, au détriment de sa précision, il est possible de choisir de ne pas annoter du tout les types concrets. Dans un tel cas il n'existe plus ni rangées, ni paramètres supplémentaires. De cette façon, nous oublions non seulement la structure des valeurs de types concrets, mais aussi les exceptions, les fonctions et les valeurs de base pouvant être encapsulées à l'intérieur de ces types concrets. Ceci a pour conséquence la nécessité pour l'analyse de faire des suppositions très pessimistes sur les valeurs extraites des structures de données sans approximation. Par exemple, si nous extrayons un entier par filtrage d'une telle structure de donnée, nous devons impérativement lui donner le type $\texttt{int}[\top]$ puisque nous avons perdu toute information sur sa valeur : il peut réellement représenter n'importe quel entier. Ceci est reflété dans le type des constructeurs en mettant $\top$ comme annotation à tous les types annotés dans l'argument des constructeurs. Dans l'exemple `intlist` ci-dessus, nous devons ainsi donner les schémas de types suivants aux constructeurs :

$$\begin{array}{lcl} \texttt{Nil} & : & \texttt{intlist} \\ \texttt{Cons} & : & \forall \rho.\, \texttt{int}[\top] \times \texttt{intlist} \xrightarrow{\rho} \texttt{intlist} \end{array}$$

Cette approche présuppose qu'il existe une annotation $\top$ pour les les types alors que le système de types que nous avons décrit en section 4.1.8 ne possède $\top$ que pour le type `int`. Néanmoins, nous pouvons utiliser $\top$ seulement pour annoter les types de base comme `string` ou `int`. Pour les exceptions et les autres structures de données, puisque leur nombre de constructeurs est fini, nous pouvons utiliser une rangée (potentiellement récursive) énumérant tous les constructeurs du type plutôt qu'une constante prédéfinie $\top$ (cf. 4.1.8). Ainsi, par exemple, dans le cas des listes polymorphes, nous pouvons utiliser la *rangée*-$\top$ suivante que nous notons : $T_{\texttt{list}}(\alpha,\rho)$:

$$T_{\texttt{list}}(\alpha,\rho) = \mu\rho'.\,\texttt{Nil} : \texttt{Pre};\ \texttt{Cons}(\alpha \times \alpha\ \texttt{list}[\rho']);\ \rho$$

Le type annoté $\tau\ \texttt{list}[T_{\texttt{list}}(\tau,\rho)]$ représente alors correctement toutes les listes d'éléments de type τ.

L'approche « aucune approximation » présentée dans cette section peut sembler excessivement grossière, mais s'avère en fait réellement efficace dans le cadre des structures de données n'introduisant ni types de base, ni exceptions, ni types fonctionnels. Deux exemples représentatifs sont les types ML prédéfinis α `list` et α `array`, dans lesquels le paramètre α mémorise déjà toute l'information dont nous avons besoin à propos des éléments de la liste ou du tableau. Par exemple, une liste de fonctions des entiers vers les valeurs de vérité possède le type $(\texttt{int}[\varphi_1] \xrightarrow{\varphi_2} \texttt{bool}[\varphi_3])\ \texttt{list}$, où φ_2 dénote l'union des effets de toutes les fonctions présentes dans la liste. Une fonction extraite de cette liste puis appliquée a donc pour effet φ_2 et non « n'importe quelle exception » comme on pourrait le croire naïvement.

Il est particulièrement intéressant de noter que notre analyse utilise cette stratégie d'approximation pour les valeurs flottantes (c'est-à-dire de type `float`). En effet, il est bien connu que l'égalité entre flottants n'a de sens qu'à ϵ près. Autrement dit, tester l'égalité de deux nombres flottants en utilisant directement la primitive `=` du langage donne en règle général des résultats incorrects. En effet, l'espace de représentation des flottants n'étant pas continu en machine (puisqu'il faut représenter une infinité de valeurs sur un nombre fini de bits), il est nécessaire d'effectuer des arrondis et des normalisations lors des calculs, pouvant induire de légères variations (voire très importantes dans certains cas de nombres et opérations mal conditionnés) affectant ainsi l'adéquation d'une égalité structurelle des valeurs. Le filtrage étant une généralisation de l'égalité, on se convaincra aisément qu'il ne possède pas plus de sens que l'égalité sur une telle structure de données et donc que notre opération de soustraction ne peut que se réduire en l'identité. Pour cette raison nous prétendons qu'il est inutile d'annoter le type `float` par une rangée qui nous sera manifestement inutile en pratique. Il est toutefois possible d'opposer à ce point de vue l'argument voulant qu'un constructeur paramétré par un flottant sera alors approximé de manière moins précise, mais l'expérience montre que de tels cas sont rarissimes, d'autant plus que ces valeurs flottantes sont en règle générale utilisées de concert avec des opérateurs arithmétiques, ayant eux, un type de la forme $\ldots \to \texttt{float}[\top]$ qui force tôt ou tard l'unification avec $\texttt{float}[\top]$.

7.1.6 Le choix de la stratégie

Le choix d'une stratégie parmi les quatre examinées précédemment peut être effectué au cas par cas selon la nature et la forme de la structure à analyser. Nous avons déjà vu la politique choisie pour le cas des flottants. D'autre part nous avons considéré plusieurs heuristiques simples afin de réaliser ce choix. Notre premier prototype d'analyseur utilisait une approximation complète pour les structures de données non paramétrées et aucune approximation dans le cas contraire. L'implémentation actuelle de l'analyse utilise désormais une approximation complète pour les structures de données non récursives ou non régulières (cf. 7.1.3), une approximation « bouclée » pour les structures récursives et aucune approximation (cf. 7.1.5) pour le cas des types prédéfinis sans structure propre intéressante du point de vue de notre analyse (par exemple pour les tableaux et les flottants comme nous l'avons vu).

Un autre facteur, qui pourrait être inclu dans cette heuristique, consisterait à vérifier si la structure de données introduit des types exception, fonction, ou des types de base pouvant paramétrer des constructeurs d'exceptions (`string` ou `int` essentiellement). Si tel n'est pas le cas, alors nous pourrions favoriser l'approche « sans approximation ».

7.1.7 Les structures de données paramétrées

Indépendamment de la stratégie d'approximation adoptée pour une définition de types utilisateur, dans le cas où cette dernière est paramétrée, il nous devient impossible d'appliquer notre opération de soustraction sauf dans le cas où le motif est totalement absorbant (c'est-à-dire est une variable) . En effet, considérons la définition et le programme suivants :

```
type α t = A of α | B of α
let v = if ... then (A 1) else (B 1) in
match v with
 | A 1 → ...
 | B x → x
```

La valeur v va avoir pour type $\texttt{int}[1:\texttt{Pre};\ \rho_0]\ t[A(\texttt{int}[1:\texttt{Pre};\ \rho_0]);\ B(\texttt{int}[1:\texttt{Pre};\ \rho_0]);\ \rho_1]$. Considérer que $A(1)$ est supprimé après le passage dans le premier cas du filtrage, revient à étiqueter son argument entier 1 par une variable de présence fraîche. Or ceci revient également à supprimer 1 des valeurs possibles de l'argument du constructeur B, ce qui est clairement faux.

Aussi l'opération de soustraction sur les structures de données paramétrées doit-elle se résumer à l'identité.

Remarquons qu'il est toutefois possible d'introduire un raffinement évitant de laisser la soustraction sans effet. En effet, nous pourrions dire que le type obtenu après soustraction est de la forme :

$$\texttt{int}[1:\texttt{Pre};\ \rho_0]\ t[A(\texttt{int}[1:\delta;\ \rho_0]);\ B(\texttt{int}[1:\texttt{Pre};\ \rho_0]);\ \rho_1]$$

Autrement dit, on ne soustrait pas dans l'argument du constructeur de type `t`, mais seulement dans la rangée qui l'étiquette. Ceci permet de mettre en évidence que dans la rangée, le constructeur `A` n'est pas paramétré par la valeur 1. On en déduit alors que c'est seulement le constructeur `B` qui l'est.

Mais en réalité cette précision est illusoire. Effectivement, si l'on cherche à extraire les valeurs arguments du constructeur `A`, nous devrons utiliser un filtrage, et ceci va forcer à nouveau l'unification du type trouvé ci-dessus avec le type du constructeur `A` (à savoir, $\alpha \xrightarrow{\rho_3} \alpha\ \texttt{t}[A(\alpha);\ \rho_4]$). À partir de ce moment-là, le lien entre la variable de présence δ et l'annotation `Pre` est rétabli, et donc on perd à nouveau le fait que 1 ne paramétrait pas le constructeur `A`. Autrement dit, on n'a rien gagné !

Aussi, étant donné le peu de gain en précision qui découlerait d'un tel raffinement, nous préférons laisser la soustraction sans effet dans de tels cas, au profit d'un gain en vitesse de notre analyse.

7.2 N-uplets et enregistrements

7.2.1 Présentation

Les *n-uplets* (en Anglais *tuples*) sont une généralisation des couples et peuvent posséder une arité quelconque. Syntaxiquement, leur introdution s'effectue en étendant la grammaire des expressions du langage par :

Expressions :

$$\begin{aligned} a \ ::= &\ \ldots \\ \mid &\ (a_1, \ldots, a_n) \end{aligned}$$

Le type d'un n-uplet dépend bien entendu du son arité et est défini par :

$$\begin{aligned} \tau \ ::= & \ \dots \\ & \mid (\tau_1 \times \ \dots \times \ \tau_n) \end{aligned}$$

Les enregistrements, quant à eux, se présentent comme une généralisation des n-uplets en ce sens qu'il s'agit de n-uplets dont chaque champ est nommé par une *étiquette*. Contrairement à SML, les enregistrements de Caml sont déclarés et filtrés par nom. Chaque étiquette ne peut appartenir qu'à un type d'enregistrement permettant ainsi l'inférence du type d'une telle expression à partir des étiquettes qui la composent. La syntaxe des enregistrements se présente comme suit :

Expressions :

$$\begin{aligned} a \ ::= & \ \dots \\ & \mid \{ \ \mathtt{lbl}_1 = a_1; \dots, \mathtt{lbl}_n = a_n \ \} \end{aligned}$$

où `lbl` est une étiquette identifiant le champ de l'enregistrement. L'ordre d'apparition étiquettes n'a pas d'importance, tant au niveau syntaxique qu'au niveau du type de l'enregistrement, type engendré par la grammaire suivante :

$$\begin{aligned} \tau \ ::= & \ \dots \\ & \mid \{ \ \mathtt{lbl}_1 : \tau_1 \dots \mathtt{lbl}_n : \tau_n \ \} \end{aligned}$$

Lorsqu'il s'agit d'inférer le type d'une expression enregistrement, il suffit alors de vérifier dans un *environnement d'étiquettes* (similaire à la fonction TypeArg utilisée précédemment en section 4.2 pour déterminer le type des arguments d'un constructeur d'exceptions) le type « hôte » de chaque étiquette de cette expression et de s'assurer que celui est bien le même pour toutes ces étiquettes. Le type de l'expression est alors ce type « hôte ». Dans le cas de construction d'une valeur enregistrement, le compilateur vérifiera également que la valeur globale (c'est-à-dire la valeur d'enregistrement dans son ensemble) comporte bien une valeur pour chaque champ de l'enregistrement afin de ne pas créer des enregistrements partiels.

Les étiquettes sont mémorisées dans l'environnement lors de la définition du type par l'utilisateur. Par exemple la définition de type suivante :

```
type personne = {
  nom : string ;
  age : int }
```

générera deux nouvelles étiquettes `nom` de type chaîne de caractères et `age` de type entier, toutes deux « hébergées » par le type `personne`.

7.2.2 Approximation

Étant donné la similitude entre ces deux structures de données, les structures d'approximation des n-uplets et des enregistrements seront très proches.

Dans le même ordre d'idée que ce que nous avons vu pour la 0-CFA (cf. 3.1), il est inconcevable de conserver la combinatoire inhérente à de telles structures. Ainsi nous n'approximons pas les n-uplets de manière particulière : chaque composante du n-uplet sera un type colportant sa propre annotation.

Pour ce qui est des enregistrements, la même remarque quant à la combinatoire s'impose. Toutefois nous devons étendre notre algèbre d'éléments de rangée afin de garder trace des étiquettes présentes dans les enregistrements. Ainsi, nous analysons ces derniers en annotant le nom du type enregistrement par une rangée d'une forme particulière et ne comportant qu'une seule valeur (ce afin d'éviter l'explosion de cas).

Nous pouvons donc résumer notre algèbre comme il suit :

Expressions de type :

$$
\begin{array}{lll}
\tau & ::= \ldots & \\
& \mid \vec{\tau}\ \mathtt{t}[\varphi] & \text{constructeur de type approximé} \\
& \mid \vec{\tau}\ \mathtt{t} & \text{constructeur de type non approximé} \\
& \mid (\tau_1 \times \ldots \times \tau_n) & \text{type n-uplet}
\end{array}
$$

Éléments de rangée :

$$\epsilon \ ::= \ldots \mid \{\mathtt{lbl}_1 : \tau_1; \ldots; \mathtt{lbl}_n : \tau_n\}$$

Ainsi, l'expression de n-uplet `if ...then (1, 2) else (3, 4)` aura pour type dans notre système :

$$\mathtt{int}[1 : \mathtt{Pre};\ 3 : \mathtt{Pre};\ \rho_0] \times \mathtt{int}[2 : \mathtt{Pre};\ 4 : \mathtt{Pre};\ \rho_1]$$

Pour une expression enregistrement de la forme : `{ nom = "Foo" ; age = 65535 }`, nous obtiendrons le type :

$$\mathtt{personne}[\{\mathtt{nom} : \mathtt{string}[\text{"}Foo\text{"} : \mathtt{Pre};\ \rho_0]\ ;\mathtt{age} : \mathtt{int}[65535 : \mathtt{Pre};\ \rho_1]\};\ \rho_3]$$

(en s'appuyant sur la définition donnée précédemment pour le type `personne`).

7.3 Structures de données mutables

Comme tout langage de taille réelle, Objective Caml fournit à l'utilisateur, outre le paradigme de la programmation fonctionnelle, des traits impératifs par l'intermédiaire de structures de données modifiables en place. Les structures de base fournies par le langage sont au nombre de deux : les *enregistrements à champs mutables* et les *tableaux*. Une troisième construction : la *référence* est également disponible et couramment utilisée lors de la programmation impérative, toutefois celle-ci ne repose pas sur une construction interne du langage mais sur enrobage syntaxique autour des enregistrements à champs mutables.

7.3.1 Un panorama des structures mutables

Les enregistrements à champ mutables

Nous avons étudié précédemment les enregistrements dont les valeurs contenues dans les différents champs ne pouvaient pas changer en place. Ceux que nous voyons maintenant diffèrent seulement des précédents dans la syntaxe de leur déclaration, par préfixage des noms de champs avec le mot clef `mutable`. Par exemple :

```
type fichier = {
 nom : string;
 mutable taille : int }
```

définit un enregistrement comportant deux champs : un « nom » ne pouvant pas changer et une « taille » sujette à modification, donc affectable en place. Nous voyons ainsi qu'il est possible de panacher les deux types de champs au sein d'un même enregistrement.

La lecture de la valeur mémorisée dans ce champ est effectué comme précédemment. Une nouvelle opération est maintenant disponible : l'affectation d'une nouvelle valeur dans le champ. Pour ce faire on écrit : $\mathtt{expr}_0\mathtt{.etiq} \texttt{ <- } \mathtt{expr}_1$. Avec notre déclaration ci-dessus, nous pourrions donc écrire quelque chose de la forme : `f.taille <- ...` où `f` est une valeur de type `fichier` et les points de suspension une valeur entière. Notons qu'une vérification statique est effectuée par le compilateur afin d'assurer que les champs affectés sont bien déclarés comme *mutables*.

Les tableaux

Les tableaux représentent la seconde structure de données polymorphe mutable du langage. À l'instar de ceux trouvés dans les autres langages de programmation, il consistent en une zone linéaire et bornée de stockage de valeurs d'un même type. Une valeur de type tableau est définie comme suit : `[|` v_1 `; ... ;` v_n `|]` où $v_1 \dots v_n$ représentent des valeurs d'un même type τ. Une telle expression aura alors le type τ `array`.

Chaque case du tableau est accessible à l'indice i aussi bien en lecture (`t.(i)`) qu'en écriture (`t.(i) <- ...`). Un indice représente donc un déplacement dans le tableau et peut varier de 0 jusqu'à la taille du tableau -1 sous peine de provoquer un accès hors limites et ainsi déclencher une exception. Nous verrons que ce dernier point est important pour le schéma de types que nous allons donner aux primitives d'accès aux tableaux.

Les références

Objective Caml définit les références sous forme d'un enregistrement de la forme suivante : `type 'a ref = { mutable contents : 'a }`. Les références sont donc l'équivalent des « variables » à la C. Dans un tel langage, les identificateurs de variables peuvent avoir deux significations différentes : dans l'expression `c = 1 + 3 ;` l'identificateur `c` représente une *adresse mémoire* où stocker le résultat du calcul de l'expression de droite. Dans l'expression `if (c == 4) printf ("...") ;` ce même identificateur désigne la *valeur* contenue dans l'adresse désignée par `c`.

En ML la distinction entre *adresse* et *valeur* est beaucoup plus tranchée. L'identificateur `c` sera lié à une *référence* donc à un enregistrement, dont la valeur pourra être accédée en consultant le champ `contents` de cet enregistrement.

Afin d'alléger l'écriture des programmes utilisant ce trait impératif, Caml fournit en standard trois notations plus compactes pour la création, l'accès aux références et leur affectation. Ainsi, pour créer une référence sur une valeur v, on utilisera la syntaxe `let c = ref v`. Pour accéder à la valeur contenue dans une référence, l'opérateur `!` permet une écriture concise sous la forme `!c`. Quant à l'affectation, plutôt que d'écrire `c.contents <- 1 + 3`, on écrira `c := 1 + 3`.

Nous voyons ainsi que cette structure de donnée sera traitée, du point de vue de notre analyse, exactemement comme les enregistrements à champs mutables.

7.3.2 L'analyse des données mutables

L'idée sous-jacente à l'analyse de telles structures de données est d'accumuler toutes les valeurs qui leur sont affectées. Pour y parvenir, il suffit d'introduire dans notre système de types, la restriction de généralisation aux valeurs [64] comme couramment utilisée en ML. Ainsi, les valeurs mutables ne pouvant être généralisées, aucune instanciation de leurs schémas ne sera effectuée et les contraintes de typage rencontrées dans tout le programme pourront s'accumuler au sein des rangées étiquetant ces valeurs.

Rappelons qu'en ML, cette restriction est rendue nécessaire afin d'éviter les typages incohérents qui pourraient survenir en cas de généralisation de valeurs mutables polymorphes. Considérons la définition `let v = ref []`. Si l'on attribue le schéma de types $\forall\alpha.\,\alpha$ `list` à v, alors il est possible d'utiliser cette dernière de façon polymorphe, par exemple dans :

```
v := [1]
v := [true]
1 + head !v
```

ce qui provoquera bien sûr une erreur à l'exécution du programme.

Grâce à ce mécanisme, notre analyse bénéficie d'une approximation relativement précise des structures de données mutables. Par exemple, un tableau de fonctions aura un type $(\tau_1 \xrightarrow{\varphi} \tau_2)$ `array`, où φ est l'union des effets latents de toutes les fonctions mémorisées dans ce tableau. Contrairement à une simple analyse de flot, qui aurait perdu trace des fonctions stockées dans le tableau et donc de leurs effets, nous pouvons ainsi éviter de déclarer que n'importe quelle exception peut être levée par l'application d'une fonction prise au hasard dans ce tableau : seules les exceptions accumulées via le type flèche argument de `array` seront concernées.

Cette caractéristique de notre analyse prend tout son intérêt par exemple dans le cadre de programmes utilisant des *callbacks*, c'est-à-dire des fonctions mémorisées dans des tables et appelées sur événement.

7.4 Modules et foncteurs

Il est souvent souhaitable de pouvoir considérer un programme comme la juxtaposition de composantes séparées coopérant entre elles. Ceci permet d'identifier plus aisément les fonctionnalités, structures de données et algorithmes requis par les programmes et permet une meilleure réutilisabilité des composants logiciels ainsi écrits. Ce besoin de modularité conduit ainsi généralement à regrouper au sein d'une même entité une structure de données et ses opérations associées et à n'en laisser filtrer à l'extérieur qu'une vue abstraite, indépendante de l'implémentation interne. Objective Caml autorise un tel mécanisme d'encapsulation par l'intermédiaire d'un système de modules d'ordre supérieur fournissant deux sortes d'expressions de modules : les *structures* et les *foncteurs*.

De même que les expressions du langage de base sont typées, les expressions du langage de module le sont également. Le type d'une expression de modules, appelé sa *signature*, permet de refléter, outre tout ou partie des composantes qui le constituent, s'il s'agit d'une structure ou d'un foncteur comme nous allons de voir.

7.4.1 Une présentation du langage de module de Caml

Nous ne prétendons pas décrire ici *in extenso* le système de modules de Caml, ceci sortant du cadre de cette thèse. Une présentation plus complète de la théorie sous-jacente peut être trouvée dans [31]. Une description particulièrement complète de l'implémentation d'un tel système est également disponible dans [30] et pourra servir de base sérieuse au lecteur désireux de se lancer dans l'implantation d'un système de modules.

Toutefois, afin de conférer à cette thèse un caractère auto-suffisant en terme pédagogique, et d'aider le lecteur débutant dans sa compréhension, nous pensons utile de nous attarder un peu sur une présentation informelle mais concise d'un tel système de modules.

Le lecteur déjà familier avec de tels systèmes peut s'affranchir de cette présentation pour se concentrer directement sur les problèmes soulevés par l'analyse elle-même et présentés en section 7.4.2.

Les structures

Les structures représentent les briques de base de ce système et sont composées d'une séquence de définitions de types, de valeurs, d'exceptions mais aussi de modules. Considérons l'exemple :

```
module IntSet =
  struct
    type t = int list
    let equal elt1 elt2 =
      elt1 = elt2
    let rec mem elt = function
      | [] → false
      | h :: q → equal h elt || mem elt q
    let add elt set =
      if mem elt set then set else elt :: set
  end
```

qui implante une structure d'ensemble de valeurs entières sous forme d'une liste d'entiers. Ce module fournit alors de façon centralisée le type `t` de ces ensembles ainsi que différentes primitives opérant dessus : `equal` pour tester l'égalité de deux éléments de l'ensemble, `add` pour l'ajout d'un élément à un ensemble et `mem` pour le test d'appartenance d'un élément à un tel ensemble.

Informellement, la signature d'une structure est composée de l'énumération des définitions de types et d'exceptions, des types des valeurs, et des signatures des sous-modules présents dans cette structure. Ainsi l'exemple ci-dessus possède la signature :

```
module IntSet :
  sig
    type t = int list
    val equal : α → α → bool
    val mem : α → α list → bool
    val add : α → α list → α list
  end
```

Les foncteurs

Les foncteurs peuvent être assimilés à des fonctions des modules dans les modules. Un foncteur prend donc un module en argument pour rendre en résultat un nouveau module. Ce mécanisme permet ainsi la génération de modules particuliers à partir d'un schéma général factorisant les comportements communs à tous ces modules. Partant de l'exemple précédent, nous pouvons remarquer que l'implémentation les fonctions `mem` et `add` ne dépend pas du type des éléments mémorisés dans l'ensemble. En effet, celles-ci ne font jamais explicitement référence à la structure propre des éléments à stocker dans l'ensemble : les élements sont vus comme des entités abstraites. De plus si nous désirons maintenant créer un ensemble de nombre rationnels encodés sous forme d'une paire d'entiers, nous devons réécrire une nouvelle structure et dupliquer inutilement les définitions de `mem` et `add`. Aussi, l'idée est de factoriser le comportement de ces fonctions dans un foncteur et d'obtenir ces deux ensembles par application de ce foncteur à deux structures fournissant chacune le type des élements voulus et l'égalité entre deux de ces éléments. Ainsi nous obtenons le foncteur suivant :

```
module Set (Elem : sig type t val equal : t → t → bool end) =
  struct
    type t = Elem.t list
    let equal = Elem.equal
    let rec mem elt = function
      | [] → false
      | h :: q → equal h elt || mem elt q
    let add elt set =
      if mem elt set then set else elt :: set
  end
```

duquel nous pouvons dériver un ensemble d'entiers et un ensemble de rationnels respectivement par :

```
module IntSet = Set (struct type t = int let equal e1 e2 = e1 = e2) end
```

et

```
module RatSet =
  Set (struct
         type t = (int × int)
         let equal (num1, denom1) (num2, denom2) = ...
       end)
```

dont les signatures seront :

```
module IntSet :
  sig
    type t = int list
```

```
    val equal : int → int → bool
    val mem : int → int list → bool
    val add : int → int list → int list
  end
```

et

```
module RatSet :
  sig
    type t = (int × int) list
    val equal : int × int → int × int → bool
    val mem : int × int → (int × int) list → bool
    val add : int × int → (int × int) list → (int × int) list
  end
```

Syntaxe et type des expressions de modules

Afin de pouvoir y faire plus aisément référence, nous donnons ici la syntaxe des expressions de modules fournies par le langage ainsi que l'algèbre de types y faisant référence.

Chemins d'accès :

p	$::= x_i$	identificateur de module
	$\mid p.x$	accès à une composante de module

Expressions de valeur :

e	$::= v_i$	identificateur de valeur
	$\mid p.v$	composante de valeur d'une structure
	$\mid \dots$	expressions de valeurs de base

Expressions de type :

τ	$::= t_i$	identificateur de type
	$\mid p.t$	composante de type d'une structure
	$\mid \dots$	expressions de types de base

Expressions de module :

m	$::= x_i$	identificateur de module
	$\mid p.x$	composante de module d'une structure
	$\mid \texttt{struct}\ s\ \texttt{end}$	structure
	$\mid \texttt{functor}(x_i : M) = m$	foncteur
	$\mid m_1(m_2)$	application de foncteur
	$\mid (m : M)$	contrainte de signature

Types de module :

M	$::= \texttt{sig}\ S\ \texttt{end}$	type de structure
	$\mid \texttt{functor}(x_i : M) \to M'$	type de foncteur

Corps de structure :

s	$::= \emptyset \mid d; s$	

Composantes de structure :

d	$::= \texttt{val}\ v_i = e$	définition de valeur
	$\mid \texttt{type}\ t_i = \tau$	définition de type
	$\mid \texttt{module}\ x_i = m$	définition de module

Corps de signature :

S	$::= \emptyset \mid D; S$	

Composantes de signature :

D	$::= \texttt{val}\ v_i : \tau$	spécification de valeur
	$\mid \texttt{type}\ t_i$	spécification de type abstrait
	$\mid \texttt{type}\ t_i = \tau$	spécification de type manifeste
	$\mid \texttt{module}\ x_i : M$	spécification de module

Modules et abstraction

Nous venons de voir comment le système de modules permettait le regroupement, sous forme de composants logiciels bien délimités, de structures de données et de leurs opérations. Nous avons également vu que les foncteurs permettaient de factoriser automatiquement les comportements communs à plusieurs modules. Toutefois l'implémentation interne de ces modules reste encore visible à l'extérieur de leur définition, permettant ainsi à un utilisateur indélicat de s'appuyer explicitement sur la structure de cette implémentation sans passer par les primitives offertes par le module.

Ceci soulève deux problèmes : dans un premier temps il devient alors impossible de changer la structure interne (l'implémentation) des ensembles sans modifier également des portions de programmes dans lesquelles le programmeur a explicitement adressé cette structure. Il s'agit donc dans ce cas d'une entrave à la réutilisabilité. D'autre part laisser l'utilisateur manipuler les structures d'ensemble de son propre chef ne garantit pas que ce dernier respectera les éventuels invariants nécessaires au bon fonctionnement de ce composant logiciel, alors que les primitives fournies par le modules sont *a priori* sensées les garantir. Ce dernier aspect s'avère beaucoup plus grave que le précédent puisqu'il entache directement la sûreté du programme.

Afin d'éviter de tels écueils, un mécanisme *d'abstraction* vient compléter le système de modules, permettant ainsi de masquer tout ou partie de la structure interne d'un module (structure ou foncteur) en contraignant explicitement une expression de module à avoir une certaine signature. La signature ainsi imposée, restreignant la signature naturellement inférée par le système, doit alors satisfaire une relation de sous-typage vis-à-vis de cette dernière. Informellement, une signature S_1 est sous-type d'une signature S_2 si :

- S_1 « oublie » de mentionner une ou des composantes présentes dans S_2,
- ou S_1 exporte une ou des composantes de type « moins précisément » que S_2,
- ou si récursivemement la signature de composantes de modules de S_1 est sous-type de la signature de composantes de même nom dans S_2.

Cette notion de sous-typage est de plus entendue modulo les équivalences de types induites par les définitions de types de ces signatures. Ainsi, soit une structure :

```
struct
  type t = bool
  let f x = A x
  type u = int
  type v = V of (u * u)
  let g x = V x
end
```

dont la signature est :

```
  sig
    type t = A of bool
    val f :  bool → t
    type u = int
    type v = B of (u × u)
    val g : u × u → v
  end
```

Nous pouvons alors la contraindre par sous-typage en la nouvelle signature :

```
  sig
    type t
    val f : bool → t
    type v = V of (int × int)
  end
```

dans laquelle nous avons oublié la structure du type `t`, la composante de type `u`, ainsi que la composante de valeur `g`. Nous avons également fait jouer l'équivalence entre `u` et `int` afin de spécifier le type de l'argument du constructeur `V`.

Typage du langage de modules

Puisque notre analyse, basée sur un système de types, doit être étendue au langage de modules, nous allons maintenant présenter rapidement le typage de telles constructions de façon à compléter la présentation informelle du contexte que nous allons étendre.

Nous donnons maintenant successivement les règles de typage des expressions de modules, des chemins d'accès et enfin des expressions de structures.

$$\frac{E(x_i) = M}{E \vdash x_i : (M/x_i)} \ (27) \qquad \frac{E \vdash m : M' \quad E \vdash M' <: M}{E \vdash m : M} \ (28)$$

$$\frac{E \vdash s : S}{E \vdash (\texttt{struct}\ s\ \texttt{end}) : (\texttt{sig}\ S\ \texttt{end}\ )} \ (29)$$

$$\frac{x_i \notin \mathrm{Dom}\,(E) \quad E \oplus (\texttt{module}\ x_i : M) \vdash m : M'}{E \vdash \texttt{functor}(x_i : M) = m : \texttt{functor}(x_i : M) \to M'} \ (30)$$

$$\frac{E \vdash m_1 : \texttt{functor}(x_i : M) \to M' \quad E \vdash m_2 : M}{E \vdash m_1\ (m_2) : M'\{x_i \leftarrow m_2\}} \ (31)$$

$$\frac{E \vdash p : (\texttt{sig}\ S_1;\ \texttt{module}\ x_i : M;\ S_2\ \texttt{end})}{E \vdash p.x : M\{n_i \leftarrow p.n \mid n_i \in \mathrm{Dom}\,(S_1)\}} \ (32)$$

$$E \vdash \emptyset : \emptyset \ (33) \qquad \frac{E \vdash e : \tau \quad v_i \notin \mathrm{Dom}\,(E) \quad E \oplus (\texttt{val}\ v_i : \tau) \vdash s : S}{E \vdash (\texttt{val}\ v_i = e;\ s) : (\texttt{val}\ v_i : \tau;\ S)} \ (34)$$

$$\frac{t_i \notin \mathrm{Dom}\,(E) \quad E \oplus (\texttt{type}\ t_i = \tau) \vdash s : S}{E \vdash (\texttt{type}\ t_i = \tau;\ s) : (\texttt{type}\ t_i = \tau;\ S)} \ (35)$$

$$\frac{E \vdash m : M \quad x_i \notin \mathrm{Dom}\,(E) \quad E \oplus (\texttt{module}\ x_i) : M \vdash s : S}{E \vdash (\texttt{module}\ x_i : M = m;\ s) : (\texttt{module}\ x_i : M;\ S)} \ (36)$$

Définissons maintenant l'équivalence entre types induite par leurs définitions ainsi que la relation de sous-typage entre signatures :

$$E_1;\ \texttt{type}\ t_i = \tau;\ E_2 \vdash t_i \equiv \tau \ (37) \qquad \frac{E \vdash p : \texttt{sig}\ S_1;\ \texttt{type}\ t_i = \tau;\ S_2\ \texttt{end}}{E \vdash p.t \equiv \tau\{n_i \leftarrow p.n \mid n_i \in \mathrm{Dom}\,(S_1)\}} \ (38)$$

$$\frac{E \vdash M_2 <: M_1 \quad E \oplus (\texttt{module}\ x_i : M_2) \vdash M_1' <: M_2'}{E \vdash \texttt{functor}(x_i : M_1) \to M_1' <: \texttt{functor}(x_i : M_2) \to M_2'} \ (39)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \quad E \oplus C_1, \ldots, C_n \vdash C_{\sigma(i)} <: C_i' \text{ pour } i \in \{1, \ldots, m\}}{E \vdash \texttt{sig}\ C_1; \ldots; C_n\ \texttt{end} <: \texttt{sig}\ C_1'; \ldots; C_m'\texttt{end}} \ (40)$$

$$\frac{E \vdash \tau <: \tau'}{E \vdash \texttt{val}\ v_i : \tau <: \texttt{val}\ v_i : \tau'} \ (41) \qquad \frac{E \vdash M <: M'}{E \vdash \texttt{module}\ x_i : M <: \ \texttt{module}\ x_i : M'} \ (42)$$

$$E \vdash \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i \ (43) \qquad E \vdash \texttt{type}\ t_i <: \texttt{type}\ t_i \ (44)$$

$$\frac{E \vdash \tau \equiv \tau'}{E \vdash \texttt{type}\ t_i = \tau <: \texttt{type}\ t_i = \tau'} \ (45) \qquad \frac{E \vdash t_i \equiv \tau}{E \vdash \texttt{type}\ t_i <: \texttt{type}\ t_i = \tau} \ (46)$$

Pour terminer cette introduction au système de types du langage de modules, nous allons nous pencher sur la définition de l'opération de *renforcement* M/p, dans laquelle M est une signature et p un identificateur de module. Il est important pour nous de décrire cette opération car celle-ci aura un impact important dans l'extension de notre analyse au système de modules.

Considérons le programme suivant :

```
module M = struct type t = A let v = A end
let v' = M.A
```

Le module `M` a donc pour signature `sig type t = A val v : t end` et la valeur `v'` a pour type `M.t`. Si maintenant nous tentons d'unifier les types de `M.v` et v' (par exemple via une expression `if ...then M.v else v'`) alors une erreur surviendra car les constructeurs de type `A.t` et `t` sont différents. Pour remédier à ce problème il suffit, lors de l'accès à une composante de module M de rendre ses définitions de types manifestes, équivalentes à leur propre nom de constructeur préfixé par le chemin complet qui a permit d'atteindre cette définition.

Ainsi, dans l'exemple ci-dessus, la valeur `M.v` aurait pour type `M.t` puisqu'il est nécessaire de suivre le chemin `M` pour accéder à la composante `A` du type `t`.

L'absence de renforcement pose également d'autres problèmes que nous n'étudierons pas dans le cadre de cette thèse, mais dont on trouvera une description dans [29]. L'opération de renforcement se présente donc formellement sous la forme :

$$
\begin{array}{rcl}
(\texttt{sig}\ S\ \texttt{end})/p & = & \texttt{sig}\ S/p\ \texttt{end} \\
M/p & = & M \text{ si } M \text{ n'est pas une signature} \\
(\texttt{val}\ v_i : \tau;\ S)/p & = & \texttt{val}\ v_i : \tau;\ S/p \\
(\texttt{type}\ t_i;\ S)/p & = & \texttt{type}\ t_i = p.t;\ S/p \\
(\texttt{type}\ t_i = \tau;\ S)/p & = & \texttt{type}\ t_i = p.t;\ S/p \\
(\texttt{module}\ x_i : M;\ S)/p & = & \texttt{module}\ x_i : M/p.x;\ S/p
\end{array}
$$

7.4.2 L'analyse des expressions de modules

Plusieurs problèmes surviennent lorsque l'on tente d'étendre notre analyse au système de modules.

Les équivalences de types

Dans un premier temps il devient nécessaire de gérer les équivalences de types induites par les définitions présentes dans les structures. Nous pouvons dissocier les définitions de types en deux classes. La première concerne les *abréviations*, qui nomment des types composés de types de base et n'introduisant pas de nouveaux constructeurs de valeur (par exemple `type t = int * bool`). Le seconde réfère aux définitions de type abstraits (par exemple `type t`) qui se contentent de définir un nouveau constructeur de type dans préciser ses valeurs, ainsi qu'aux définitions de types concrets qui introduisent, elles, de nouveaux constructeurs de valeur (par exemple `type t = A of int`). Nous voyons que les équivalences de types sont ainsi uniquement introduites par des *abréviations*.

Nous remarquons que l'algèbre d'approximation que nous avons décrite jusqu'à présent (cf. section 4.1.8) ne comporte des éléments de rangées que pour les types de base (comme `exn`, `int`), les types concrets, et les les types enregistrements. Considérons la définition (ou plus exactement l'abréviation) `type t = (int * int)` et une expression que nous contraignons à avoir ce type : `((13, 42) : t)`. Nous ne savons alors pas comment annoter le constructeur de type `t`. En effet, le type annoté tel qu'inféré naturellement pour le couple est $\texttt{int}[13;\ \rho_0] \times \texttt{int}[42;\ \rho_1]$. Or la forme des rangées permettant l'annotation des constructeurs de type concrets n'est en aucun cas compatible avec ce résultat.

Le première solution consiste à considérer que toute abréviation est grossièrement approximée par la rangée $\top$. Ceci induit deux inconvénients majeurs. En premier lieu ceci va à l'encontre de l'invariant énoncé en section 4.1.8 voulant que $\top$ ne soit utilisé que dans des rangées étiquetant des types de base. En second lieu, ceci dégrade notablement les performances de notre analyse, non seulement dans le cadre de l'utilisation des modules, mais également dans des programmes moins complexes étant donné l'utilisation fréquente de telles abréviations. En effet, il est courant de définir des abréviations afin de nommer des expressions de types compliquées en vue de les utiliser dans d'autres définitions ou pour contraindre le type d'expressions. Il sera par exemple plus simple d'écrire (et de lire) le programme suivant :

```
let (v : (string * (int * int) list) ref) = ref []
```

sous la forme :

```
type key = string
type position = int
type extent = int * int
type index = (key * extent) list
let (v : index ref) = ref []
```

La seconde solution consiste simplement à expanser systématiquement les abréviations de types. Bien que cette stratégie puisse sembler notablement inefficace, les résultats pratiques que nous avons obtenus (cf. section 9.1.1) démontrent la viabilité de cette technique, même sur des programmes de taille réelle.

L'annotation des structures

L'analyse des structures est simplement effectuée en inférant leur signature, dans laquelle les composantes de valeurs possèdent des types annotés et les composantes de définition de types sont expansées comme expliqué ci-dessus. Nous voyons en particulier qu'il n'est pas nécessaire de développer de nouvel algorithme d'analyse, le mécanisme standard d'inférence du langage de modules conviendra naturellement pour ces structures. Ainsi, la structure suivante :

```
struct type t = int * bool let (v : t) = (42, true) end
```

aura pour signature :

$$\texttt{sig type t } = \texttt{int} \times \texttt{bool val v} : (\texttt{int}[42; \rho_0] \times \texttt{bool}[\texttt{true}; \rho_1]) \texttt{ end}$$

Les règles de typage du système de modules imposent, lors de l'accès à un chemin de structure, le préfixage de toutes les composantes de ce module par le chemin permettant d'y accéder (cf. substitution $\{n_i \leftarrow p.n \mid n_i \in \mathrm{Dom}\,(S_1)\}$ dans la règle 27 du typage des expressions de modules), ce afin de garantir que le type de la valeur `v` d'un module `module A = struct type t = ...val v : t end` soit `A.t` et non simplement `t` quand nous y faisons référence à l'extérieur de ce module. De plus il reste nécessaire que cette valeur ait le type `t` à l'intérieur de cette structure. Nous avons vu que le système de types des modules conventionnel effectue cette opération, ainsi que la propagation des équivalences de type *dynamiquement*, c'est-à-dire à la demande. Or notre mécanisme de typage requiert lui, comme nous l'avons vu précédemment, que ces mécanismes soient réalisés *systématiquement* et *statiquement*. Ainsi, il nous est impossible de conserver cette ubiquité des types (c'est-à-dire la possibilité de les voir sous différents noms). La solution consiste à effectuer un choix fixe en décidant que les constructeurs de types seront toujours connus sous leur nom préfixé du chemin d'accès de leur module hôte.

Ainsi lorsque nous typons une expression de structure, nous préfixerons systématiquement et récursivement tous les noms de constructeurs de types présents dans la signature obtenue, par leur chemin d'accès complet (aussi bien pour les composantes de valeurs que dans les composantes de définition de types).

Ainsi, l'expression :

```
module M = struct
  type t = A
  let v = A
  module N = struct
    type u = B of t
    let w = B A
    end
end
```

se verra assigner la signature :

```
sig
type t = A
val v : M.t[A : Pre ;  ρ_0]
module N : sig
  type u = B of A.t
  val w : M.N.u[B(M.t[A  :  Pre ;  ρ_1]) ;  ρ_2]
  end
end
```

dans laquelle nous distinguons clairement le préfixage effectué dans les expressions de types. De même, les schémas de types associés aux constructeurs de valeurs des types `t` et `u`, doivent-ils subir le même traitement, nous donnant ainsi respectivement les schémas de types : $A : \forall\rho.\,\mathtt{M.t}[\mathtt{A} : \mathtt{Pre};\, \rho]$ et $B : \forall\rho_0, \rho_1, \rho_2.\,\mathtt{M.t}[\rho_0] \xrightarrow{\rho_1} M.N.u[B(M.t[\rho_0]);\, \rho_2]$.

Cette opération de préfixage requiert *a priori* de repasser sur tous les types déjà inférés, tant dans les signatures que dans les définitions de types. Afin d'éviter un tel parcours, une solution plus adaptée consiste à partager une référence sur le nom d'un constructeur, partout où ce constructeur est utilisé, puis à modifier en place le moment venu cette référence en y substituant le nom du constructeur de type préfixé par son nom de module de définition. Ainsi, dans l'exemple suivant :

```
module A = struct
  type t
  type u = t * t
  let v_1 = ((..., ...) : u)
  let v_2 (x : t) = (... : u)
  end
```

nous obtiendrons dans un premier temps la signature suivante, dans laquelle, d'une part l'expansion des abréviations aura joué son rôle, d'autre part les nom de constructeurs soulignés sont *physiquement* partagés :

```
module A : sig
  type t = t
  type u = t * t
  val v_1 : t * t
  val v_2 : t -φ-> t * t
  end
```

Une fois la substitution en place effectuée (c'est-à-dire en *une seule* opération d'affectation de référence), la signature finale devient alors :

```
module A : sig
  type t = A.t
  type u = A.t * A.t
  val v_1 : A.t * A.t
  val v_2 : A.t -φ-> A.t * A.t
  end
```

Notons qu'il n'est pas nécessaire de préfixer les éléments de rangée que représentent les noms de constructeurs de valeurs, ceci étant déjà effectué au niveau du constructeur de type lui-même.

Ainsi, dans le cas suivant :

```
module A = struct exception E end
module B = A

try ... raise A.E ... with
 | B.E → ...
```

Les deux exceptions E des modules A et B seront de type $\texttt{exn}[E : \texttt{Pre};\ \rho]$ (le type `exn` n'étant pas défini dans un de ces modules, il n'a bien entendu pas à subir de préfixage). L'élément de rangée E n'étant pas préfixé, la soustraction va se comporter comme attendu et l'expression finale aura pour effet : $[E : \delta;\ \rho']$ où δ et ρ' peuvent être généralisées et n'apparaissent qu'en position positive, donc dénotent des ensembles vides. Ainsi, cette expression ne laisse pas s'échaper d'exception.

De plus ceci nous permet, à un niveau purement implémentatoire, de réduire la taille des approximations d'où un gain de vitesse et d'espace mémoire requis par notre analyseur, contribuant ainsi à son application à de larges programmes.

Satisfaction d'une signature par une structure

Nous avons vu qu'il est possible de contraindre une expression de structure S (et par extension une expression de modules en règle générale) à avoir une certaine signature Σ sous la condition que cette expression S satisfasse la signature Σ. Cette condition de satisfaction relève d'une relation de sous-typage entre la signature imposée et celle naturellement inferée par le système de types pour la structure. Deux sémantiques sont alors possibles pour prendre en compte cette fonctionnalité du système de modules dans notre analyse.

La première, appelée *sémantique opaque*, consiste à dire que les seules informations filtrant d'une contrainte $(S : \Sigma)$ sont celles que Σ rend publiques. Dans notre, cas, puisque l'utilisateur fournit des signatures sans annotations sur les types, ceci revient à oublier le résultat de notre analyse du corps du module et à considérer que chaque composante de valeur de la structure analysée est annotée par une rangée $\top$.

La seconde sémantique dite *sémantique transparente*, vérifie seulement que la structure S satisfait la signature Σ mais oublie après coup la restriction imposée par cette signature. Ainsi toutes les informations présentes dans la signature originalement inférée par le système de types restent disponibles et visibles. Intuitivement, ceci revient à simplement ignorer les contraintes de signatures.

Notre analyse repose sur la sémantique *transparente*, la sémantique *opaque* résultant en de trop sévères pertes de précision. De plus l'utilisation d'une sémantique *opaque* résulterait en une impossibilité de déterminer la forme des annotations d'une structure de données en fonction de la définition de son type (rappelons-nous que nous avons mentionné précédemment, en section 7.1.6, la possibilité de choisir la stratégie d'approximation d'un type de données en fonction de la « forme » de sa définition).

Analyse des foncteurs

L'analyse des foncteurs pose des problèmes similaires à celle des structures en terme de satisfaction de signature. En effet, nous avons vu que la définition d'un foncteur nécessitait que la signature de son argument soit explicitement fournie par l'utilisateur (cf. règle 30 du typage du langage de modules).

Puisque tout ce que nous savons de l'argument d'un foncteur pour typer son corps provient de cette signature, l'adoption d'une sémantique opaque pour notre analyse aurait à plus forte raison causé une sévère dégradation de sa précision. En effet ceci nous aurait forcé à présumer une annotation $\top$ sur chaque composante de valeur de l'argument et justifie donc *a posteriori* notre préférence pour la sémantique transparente des contraintes de signatures. Toutefois l'adoption d'une telle sémantique ne suffit pas à éliminer les problèmes posés par l'analyse des foncteurs. Considérons un foncteur de la forme suivante :

```
module F (X : sig f : int → int end) = sig ... end
```

L'adoption d'une sémantique transparente ne nous affranchit pas de devoir annoter les types de la signature de F. Deux solutions s'offrent alors à nous.

1. Donnons-lui la signature annotée : $\texttt{functor}\ (X : \texttt{sig}\ f : \texttt{int} \xrightarrow{\rho} \texttt{int end}) \rightarrow \texttt{sig}\ldots\ \texttt{end}$, dans laquelle la variable de rangée ρ n'est pas quantifiée (n'est pas généralisée). Ainsi, celle-ci va accumuler les effets de toutes les fonctions f provenant de modules appliqués à F, ainsi que les effets provenant de leurs utilisations dans le corps de F. Il en résulte manifestement une sérieuse perte de précision pour notre analyse.
2. Nous voulons donc que la variable de rangée ρ, devant annoter le type de la fonction f de l'argument du foncteur F, soit généralisée afin d'éviter ces accumulations. Il se pose alors la question de savoir où mettre la quantification portant sur ρ. À nouveau, nous avons deux possibilités.

   (a) Mettons-la en tête de la signature de F. Nous obtenons alors une signature de la forme : $\forall \rho. \texttt{functor}\ (X : \texttt{sig}\ f : \texttt{int} \xrightarrow{\rho} \texttt{int end}) \rightarrow \texttt{sig}\ldots\ \texttt{end}$. Sous cette forme, ρ n'accumule plus l'effet des fonctions f issues de différentes applications du foncteur. Toutefois ρ n'étant pas généralisée *à l'intérieur* de F, toutes les utilisations de f au sein de F vont accumuler leurs effets, ce qui résulte à nouveau en une perte de précision de l'analyse.

   (b) Le dernière solution consiste à quantifier ρ directement dans le type de f, donnant ainsi une signature de la forme : $\texttt{functor}\ (X : \texttt{sig}\ f : \forall \rho.\ \texttt{int} \xrightarrow{\rho} \texttt{int end}) \rightarrow \texttt{sig}\ldots\ \texttt{end}$. Malheureusement il devient alors impossible d'inférer les signatures pour un tel langage de modules [33, 53, 59]. De plus, de par la sémantique de la satisfaction de signatures pour le langage de modules, ceci imposerait alors de ne pouvoir appliquer F qu'à des modules comportant une fonction f *exactement* de type $\forall \rho.\ \texttt{int} \xrightarrow{\rho} \texttt{int}$, c'est-à-dire *ne levant aucune exception*, ce qui n'est clairement pas la sémantique attendue.

Considérons maintenant le problème de la *générativité* des déclarations d'exceptions dans le corps d'un foncteur sur l'exemple suivant :

```
module F (X : sig ... end) = struct
  exception Exn
  let f x = raise Exn
end
module A = F(...)
module B = F(...)
```

Chaque fois que le foncteur `F` est appliqué à une structure, une nouvelle exception `Exn` est générée, différente de celles engendrées par les précédentes applications de ce foncteur. Ainsi le schéma de types de $A.f$ devrait être $\alpha \xrightarrow{A.Exn} \beta$ et celui de $B.f$ devrait être $\alpha \xrightarrow{B.Exn} \beta$, nécessitant une opération de renommage des noms d'exceptions dans les types. Or nous avons mentionné précédemment qu'il n'était pas nécessaire de préfixer, dans les effets, les noms de constructeurs d'exceptions.

De plus ce préfixage devient problématique en cas d'application du foncteur à une expression de module anonyme comme par exemple dans les cas suivants :

```
module F (X : sig exception Exn end) = struct let f x = raise X.Exn end

module R1 = F(struct exception Exn end)
R1.f ()
```

```
module G = functor (X : sig ... end) → ...
module Y = struct exception Exn end
module R₂ = F (G (Y))
R₂.f ()
```

En effet, dans le premier exemple, la structure passée en argument à F n'étant pas nommée, nous ne pouvons pas la prendre directement comme préfixe. Par exemple, dans l'effet de la fonction `f`, nous obtiendrions un constructeur d'exceptions manifestement mal formé, de la forme (`struct exception Exn end`).`Exn`.

Le second exemple pose les mêmes problèmes avec une autre forme d'expression de modules, à savoir l'application du foncteur F, non pas à une structure anonyme, mais à une autre application de foncteur, elle-même non nommée. Obtenir par exemple, dans l'effet de la fonction `f`, un constructeur de la forme $(G(Y))$.`Exn` serait également mal formé.

En dernier lieu, comment faire la différence entre les noms des constructeurs exceptions générées dans le cas où l'on applique plusieurs fois le même foncteur à un même module argument. En effet, bien que les modules mis en jeu lors des applications soient identiques, les constructeurs d'exceptions générés sont différents et donc un simple schéma de nommage basé sur le nom de l'argument appliqué au foncteur ne serait pas suffisant.

Pour terminer examinons le problème provenant du *polymorphisme d'exception* qu'autorisent les foncteurs. Considérons les définitions de modules suivantes :

```
module A = struct exception Exn end
module B = struct exception Exn end
module F (X : sig exception Exn end) =
  struct
  let test () = try raise A.Exn with X.Exn → ()
  end
```

Il nous est alors impossible d'analyser finement le corps du foncteur, en particulier d'appliquer notre opération de soustraction tant que nous ne savons pas à quelle structure notre foncteur F est appliqué. En effet, soit ce dernier est appliqué à A et dans ce cas la fonction `test` ne lève aucune exception. Soit il est appliqué à B et cette dernière alternative autorise la fonction `test` à lever une exception. Si nous devions inférer une signature pour le module F, celle-ci serait de la forme :

```
module F :
  functor(X : sig exception Exn end) →
    sig val test : unit →? unit end
```

dans laquelle il nous est impossible de spécifier l'effet supporté par la flèche du type de `test` (ici remplacé par un point d'interrogation), puisque nous ne connaissons pas encore la valeur effective de X. Une solution consisterait à y mettre une annotation $\top$ en sacrifiant à la précision de notre analyse. Notons que nous pourrions être un peu moins pessimiste et décider de mettre l'annotation `A.Exn` systématiquement à la place de $\top$, mais ceci réduirait quand même la précision de notre analyse.

Afin de contourner tous ces écueils, nous avons choisi, non pas d'analyser le corps des foncteurs lors de leur définition, mais plutôt d'expanser leur corps lors de chaque application et de ré-analyser ce dernier β-réduit (c'est-à-dire dans lequel le nom de l'argument formel du foncteur a été remplacé par celui de l'argument effectif de l'application, c'est-à-dire celui du module auquel on applique le foncteur). Ceci impose bien entendu la nécessité de nommer explicitement les expressions de modules servant dans une application. Bien que cette transformation du programme augmente la taille du source à analyser, les programmes écrits en Caml que nous avons vu jusqu'à présent n'utilisent pas les foncteurs de manière intensive et cette approche simple pour analyser les foncteurs apparaît tout à fait efficace en pratique. Notons que de telles techniques sont déjà utilisées dans [6, 9].

Notons en dernier lieu qu'Objective Caml autorise l'application de foncteurs à des structures anonyme. Cette construction n'apparaît pas dans le langage de modules que nous avons décrit jusqu'à présent. Ainsi, est-il possible d'écrire directement une expression de la forme `module R = F(struct ...end)` où `F` est un foncteur dont l'argument est une structure directement donnée en extension. Ceci rend impossible la β-réduction que nous effectuons dans le corps des foncteurs puisque dans un tel cas l'argument formel de ce dernier ne peut être remplacé par un *nom* de module. La solution à ce problème réside simplement dans un nommage au vol de toutes les expressions de modules anonymes avant typage et peut être réalisée directement sur l'arbre de syntaxe abstrait du programme. On veillera seulement à choisir comme noms pour ces expressions de modules, des identificateurs ne pouvant pas entrer en conflit avec ceux définis par l'utilisateur.

7.5 Classes et objets

Puisque notre système repose d'ores et déjà sur une algèbre de types cycliques, l'ajout des objets tels qu'Objective Caml les fournit ne complexifie qu'assez peu notre analyse. En effet, nous nécessitons seulement d'étendre notre algèbre de types avec des types objets, c'est-à-dire des enregistrements de méthodes polymorphes [48] . Le type de chaque méthode est alors annoté par son effet latent. Nous obtenons alors une algèbre de type étendue par :

Expressions de types :

τ	$::= \ldots$	
	$\mid \langle\tilde{\tau}\rangle$	type d'objet

Enregistrements extensible de méthodes :

$\tilde{\tau}$	$::= (m : \tau;\ \tilde{\tau})$	méthode m
	$\mid \rho$	variable de rangée (enregistrement ouvert)
	$\mid \emptyset$	rangée vide (enregistrement fermé)

Nous remarquerons en particulier qu'il n'est pas nécessaire d'étendre ni la structure des rangées ni celle des éléments de rangées. De plus, étant donné qu'il n'existe pas de « motif d'objet » dans les filtrages (`match ... with ...`) notre opération de soustraction n'aura pas non plus à être modifiée.

Le langage de classes d'Objective Caml interfère relativement peu avec l'analyse d'exceptions en elle-même. Aucune modification majeure du typeur d'expressions de classes n'est alors nécessaire pour intégrer ces constructions dans notre analyse. Ainsi, ne décrirons-nous pas ici la structure d'un tel système, cet aspect relevant plutôt du domaine du typage des langages à objets que de celui de l'analyse d'exceptions. Le lecteur désireux d'en savoir plus à ce sujet pourra consulter [32, 48].

7.6 Analyse séparée

Afin de rendre possible le développement de larges applications, Objective Caml autorise un mécanisme de « compilation séparée ». Le programmeur segmente le source de son logiciel en plusieurs unités de compilation, c'est-à-dire en plusieurs fichiers, chacun étant considéré comme un module au niveau du langage lui-même. Chacun de ces fichiers peut alors être compilé indépendamment des autres, pourvu que les interfaces de ceux dont il dépend aient été préalablement compilées. Ceci présente plusieurs avantages :

- la taille des fichiers sources manipulés est plus réduite ;

- le découpage suivant la structure logique du programme est facilitée, d'autant plus qu'il est possible de mettre à profit le mécanisme d'abstraction fourni par le système de modules ;
- la ré-utilisabilité de composants logiciels est relativement simple puisqu'il suffit de reprendre les fichiers correspondant aux unités de compilation que l'on souhaite ré-utiliser ;
- et enfin le temps de compilation est réduit puisqu'il n'est pas nécessaire de re-compiler toute l'application à chaque fois, seuls les fichiers modifiés et ceux en dépendant nécessitant une re-compilation.

Du point de vue de notre analyse d'exceptions, nous souhaitons également bénéficier d'une faculté « d'analyse séparée » afin que le temps nécessaire pour traiter des programmes de grande envergure reste raisonnable.

En particulier, à l'instar de ceux écrits dans d'autres langages, les programmes écrits en Objective Caml font généralement usage de nombreuses fonctions fournies par la bibliothèque standard du langage. Cette dernière doit bien sûr être analysée préalablement aux programmes utilisateurs. Or il serait souhaitable que l'utilisateur n'ai pas à ré-analyser toute cette bibliothèque, ainsi que tous les sources de son application à chaque modification de l'un d'entre eux.

Notre analyse étant basée sur un mécanisme de typage, nous allons naturellement tenter de reproduire un mécanisme similaire à celui permettant la compilation séparée dans Objective Caml lui-même. Ce dernier procède schématiquement de la façon suivante : lors de la compilation d'une unité, un fichier d'interface est généré, dans lequel sont inscrits les types des composantes que cette unité exporte. Ainsi toute autre unité de compilation dépendant de la première n'aura qu'à se reporter à ce fichier d'interface afin de connaître le type des composantes dont elle a besoin et se trouvant dans la première unité de compilation.

De notre point de vue, l'idée est similaire. Toutefois une difficulté technique apparaît via nos variables de rangée et de présence. En Objective Caml, il est refusé qu'une unité de compilation exporte des variables de type non généralisées, ceci afin d'éviter que deux modules dépendant de cette unité n'instancie ces variables avec des types incompatibles. Examinons l'exemple suivant :

```
module A = struct let l = ref [] end
module B = A.l := [1] end
module C = A.l := [true] end
```

La signature du module A étant *a priori* `sig` $v : \alpha$ `list ref end`, le module B instancie α en `int`, et le module B l'instancie en `bool`. En premier lieu ces instanciation semblent légales, toutefois elles résulteraient à l'exécution en une erreur, la liste $A.l$ contenant des valeurs de types différents. Autoriser une telle manipulation rend donc le système de types incorrect.

Dans notre système, les variables de rangée et de présence annotant les structures de données mutables (par exemple) ne sont pas généralisées. Par exemple :

```
module A = struct let v = ref 1
```

aura pour signature `sig` v : `int`$[1:$`Pre`$;\, \rho]$ `end` dans laquelle la variable ρ n'est pas généralisée. En ML ceci ne pose pas de problème particulier puisque le type de v en lui-même n'est pas polymorphe. Par contre, en raison des ajouts de variables induits par notre algèbre de type particulière, une certaine forme de polymorphisme apparaît désormais.

Considérons maintenant que deux autres unités de compilation effectuent, tout à fait légalement du point de vue de ML, des affectations dans cette référence :

```
module B = struct let _ = v : = 13 end
module C = struct let _ = v : = 42 end
```

Il est impératif qu'aucune de ces affectations ne soit faite au détriment de l'autre, et qui plus est, il est nécessaire que chacun des modules B et C en ait connaissance.

Bien entendu, il est hors de question d'interdire de telles expressions de modules sous prétexte d'analyse séparée. Notre système de types deviendrait alors plus restrictif que celui de ML par l'intermédiaire du systèmes de modules.

La solution adoptée consiste à découper notre analyse en deux passes. Lors de la première, les types sont synthétisés indépendamment des utilisations qui en sont faites dans les autres unités de compilation. Un fichier d'interface est alors généré, tout comme c'est le cas avec le compilateur d'Objective Caml. Toutefois ce fichier d'interface mémorise désormais la liste des variables (de rangée et de présence) non généralisées exportées par ce module, ainsi que la liste des variables non généralisées utilisées en provenance d'autre modules.

Une fois que toutes unités de compilation ont été analysées, il devient possible d'examiner les résultats de l'analyse pour le programme tout entier grâce à une phase spéciale « d'édition de liens » (*link*, en Anglais) qui se charge de récupérer et de fusionner toutes les instanciations ayant été faites sur les variables non généralisées de chaque module. Ceci est simplement réalisé par unification de tous les types trouvés pour chaque occurrence de variable dans les modules utilisant ces variables.

Puisque les programmes que nous analysons sont sensés avoir été préalablement acceptés par ML, il apparaît que nous ne pouvons pas avoir à faire de telles unifications sur des variables de type ML. Aussi, nous reste-t'il que des unifications sur des variables de rangées et de présence, et par définition ces unifications ne peuvent pas échouer. Ainsi notre phase finale ne doit jamais provoquer d'erreur.

Il est toutefois intéressant de noter que notre mécanisme de soustraction permet l'introduction de variables de type ML dans les rangées. Considérons l'exemple suivant, certes créé de toutes pièces :

```
module A = struct
  type α t = Test of α
  exception Not_found
  let v = Test (match Not_found with
              | _ → raise Not_found
              | whatever → whatever)
  end
```

Notre opération de soustraction nous permettant, dans le dernier cas, de retourner un type quelconque, par principalité, nous retournons une variable de type ML. Ainsi, le type de v est α $\mathtt{t}[\mathtt{Test}(\alpha);\,\rho]$. La variable α apparaît donc dans la rangée alors qu'en ML, les rangées n'existant pas, ce problème ne se pose pas.

Dans la réalité, il nous faut donc quand même nous attendre à devoir gérer des unifications entre variables de type ML, mais celles-ci ne peuvent également pas échouer. En effet, sans entrer dans une preuve formelle de cette propriété, l'idée est que si de telles variables existent au sein d'un module, c'est qu'elles proviennent de la soustraction et qu'aucune contrainte ne pèse dessus dans le module. Autrement dit, il n'existe aucun lien entre cette variable et d'autres points de ce module. Imaginons maintenant que deux autres modules tente d'instancier cette variable :

```
module B = struct let v' = match A.v with A.Test 1 → 1 end
module C = struct let v' = match A.v with A.Test true → true end
```

alors ces deux instanciations vont entrer en conflit. Mais au passage elles rétablissent un lien entre cette variable et le reste du programme, lien que le typage de ML capture également et qui le fera rejeter lui aussi ce programme qui s'avère mal typé !

Ainsi intuitivement, soit toutes les occurrences de cette variable resteront non contraintes (donc resteront vraiment des variables ML) et nous savons toujours unifier deux variables entre elles. Soient des instanciations auront lieu et elles se devront d'être valides comme elles doivent l'être en ML. En ce sens que les programmes que nous analysons sont sensés avoir été préalablement acceptés (typés) par ML, notre phase finale d'analyse n'échouera donc pas.

Dans la pratique le coût de cette phase finale de l'analyse n'a pas un gros impact sur la rapidité de l'analyse. Néanmoins, le stockage sur disque des fichiers d'interface générés peut, lui, s'avérer plus long. En effet, ces fichiers colportent une grande quantité d'information : les arbres des types, l'arbre de syntaxe du programme annoté par ces types, la listes des variables non généralisées exportées ainsi que la listes des variables non généralisées importées d'autres unités de compilation.

Bien que ces structures bénéficient d'un partage important, les fichiers résultant sont relativement gros comparés à des fichiers d'interface produits par le compilateur Caml.

Toutefois ce coût de stockage reste tout à fait acceptable. Nous mentionnons ce point dans le but de bien exposer la différence entre le temps d'analyse proprement dit et le temps nécessité par les opérations annexes issues de l'implémentation de cette analyse en tant qu'outil utilisable dans le monde réel...

7.6.1 Exceptions homonymes

Afin de rendre uniques tous les identificateurs d'un programme une phase de détermination de la portée lexicale (*scoping*, en Anglais) est réalisée préalablement à l'analyse. Elle consiste à ajouter une étiquette unique à chacun d'entre eux afin d'éviter toute homonymie (en prenant soin de faire attention aux définitions récursives). Ce mécanisme est à rapprocher de l'alpha-conversion. Ainsi, le programme :

```
let x = ... in
... x ...
let x = ... x ... in
... x ...
let rec x = ... x ... in
...
```

est transformé en :

```
let x0 = ... in
... x0 ...
let x1 = ... x0 ... in
... x1 ...
let rec x2 = ... x2 ... in
...
```

Ce mécanisme s'applique bien entendu également aux constructeurs d'exceptions. L'attribution de cette étiquette en terme d'implémentation est simplement effectuée par l'ajout d'un entier au nom, entier issu d'un compteur incrémenté au fur et à mesure que l'on en a besoin. Ainsi au sein d'un même fichier, c'est-à-dire d'une même unité de compilation tous les constructeurs d'exceptions sont de nom différent.

Toutefois, avec ce simple mécanisme, si deux unités de compilation comportent une exception de même nom et que les étiquettes leur étant attribuées sont malheureusement les mêmes, alors il va y avoir quand même un conflit de nom. En effet, nous avons vu précédemment que les constructeurs n'ont pas besoin d'être préfixés par le nom de leur module hôte (contrairement aux noms de types, rappelons-le). Donc, dans l'exemple suivant il va y avoir confusion entre les deux exceptions `Empty` des deux modules `Stack` et `Queue` :

```
Fichier stack.ml :
  exception Empty
  let push ... = ...

Fichier queue.ml :
  exception Empty
  let enqueue ... = ...
```

La solution consiste à préfixer les constructeurs d'exceptions par le nom de l'unité de compilation qui contient leur définition. Ce nom étant unique grâce au système de gestion de fichiers, nous ne risquons plus d'homonymie parasite.

Ce mécanisme appelle une précision. En effet, que se passe-t'il dans le cas suivant :

```
Fichier file1.ml :
exception E

Fichier file2.ml :
module B = File1

Fichier file3.ml :
try ... raise File1.E ... with
 | File2.B.E → ...
```

A priori, l'exception E se trouvant dans le fichier `file1.ml` va posséder le type annoté suivant : `exn`$[File1\_\_E : \texttt{Pre};\, \rho]$. La question est maintenant de savoir comme va être nommée l'exception E se trouvant dans le module B du fichier `file2.ml`. Si celle-ci est appelée `File2__E` (c'est-à-dire si la composante `File2.B.E` a le type `exn`$[File2\_\_E : \texttt{Pre};\, \rho']$), alors notre mécanisme de soustraction va échouer lors du typage de l'expression du fichier `File3.ml` et cette dernière va être considérée comme laissant s'échapper une exception (en effet, il n'y a pas égalité de nom entre « File1__E » et « File2__E »).

En fait, lorsque nous typons l'expression de module du fichier `File2.ml`, nous ne faisons que dire que le module B a la même signature que le module $File1$. En particulier, le type de l'exception E du module `File1` se retrouve intégralement transmis en tant que type de l'exception du module B défini dans `File2.ml`. Et dans ce type, l'élément de rangée étiquetant le type de notre exception *est et reste* `File1__E` !

Ainsi, nous voyons que, naturellement, il y a conservation du nom des exceptions au travers des modules, même en présence de notre mécanisme servant à les désambiguïser.

Chapitre 8

La récursion polymorphe

Le système de types que nous avons présenté dans les chapitres précédents présente de nombreuses similitudes avec celui utilisé par ML. En particulier il autorise du polymorphisme sur les ensembles d'exceptions par généralisation de variables de rangée et de variables d'effet. Comme en ML, le polymorphisme est alors introduit par la règle gérant la construction `let`. Ainsi tout ensemble d'exceptions non lié par un `let` verra ses variables non généralisées, induisant une possibilité d'accumulation de ses éléments avec ceux de tout autre ensemble entrant en contact avec lui par unification. L'introduction du polymorphisme en ML par la construction `let` est un mécanisme fonctionnant fort bien en pratique de par la forme des programmes.

Outre la liaison simple d'identificateurs, ML autorise la *liaison récursive* d'identificateurs, c'est-à-dire la possibilité de lier un identificateur et d'utiliser ce dernier à l'intérieur de sa propre définition. Ceci est réalisé par l'intermédiaire d'une construction `let rec`, comme dans l'exemple suivant :

```
let rec fact = function 0 → 1 | n → n * fact (n-1)
```

Ainsi la grammaire des expressions du langage s'étend simplement par :

Expressions :		
a ::= ...		anciennes expressions
	$\mid$ `let rec` $x = a_1$ `in` a_2	liaison `let rec`

Dans une telle définition, l'identificateur lié récursivement ne peut alors être utilisé de façon polymorphe à l'intérieur de sa propre définition. Autrement dit, il est inséré dans l'environnement de typage du corps de la définition sous forme d'un schéma de types trivial dans lequel aucune variable n'est généralisable, comme l'illustre la règle de typage suivante :

$$\frac{E \oplus (x : \forall\emptyset.\,\tau_1) \vdash a_1 : \tau_1 \qquad \sigma = \mathrm{Gen}\,(\tau_1, E) \qquad E \oplus (x : \sigma) \vdash a_2 : \tau_2}{E \vdash \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Ceci implique alors que toutes les instances de l'identificateur présentes dans le corps de sa définition devront être utilisées avec un seul et même type. Ainsi la phrase :

```
let rec f x = f 1; f true
```

sera considérée comme mal typée et rejetée par le compilateur. En effet dans cet exemple, `f` est utilisée avec les types `int` $\to \alpha$ et `bool` $\to \alpha$. On notera toutefois que donner à `f` le type $\forall \alpha, \beta.\, \alpha \to \beta$ resterait correct vis-à-vis de la sémantique dynamique du langage.

Cette restriction est introduite dans le système de types de ML afin de conserver la décidabilité de l'inférence de types. En effet, l'inférence de types en présence de `let rec` polymorphe, problème aussi connu sous le nom de typage avec « règle μ », est connu pour être indécidable. Ceci signifie qu'il n'existe pas d'algorithme (donc de processus dont la terminaison est garantie) de synthèse de types complet pour ce problème. Tout au plus est-il possible d'écrire un semi-algorithme qui, s'il existe une solution au problème de typage posé, trouve cette solution, sinon risque de ne jamais terminer.

Le typage en présence de `let rec` polymorphe a été tout d'abord été introduit par [36] qui décrit alors un semi-algorithme permettant de réaliser l'inférence de types dans un tel cadre. L'indécidabilité de ce problème peut être trouvée dans [26] et dans [19] qui en démontre l'équivalence avec un problème de semi-unification.

Le lecteur se demande peut-être quel est l'intérêt d'exposer ce problème dans le présent travail. Considérons le programme suivant, constitué d'une fonction récursive protégeant son propre appel récursif dans un gestionnaire d'exceptions :

```
let rec f x =
  try f x with
   | C → ()
   | y → raise y
```

Il est clair que cette fonction ne peut en aucun cas laisser s'échapper d'exception. Lors de l'analyse, l'effet latent inféré pour `f` est toutefois C : `Pre`; ρ, signifiant que l'exception C peut en fait s'échapper. En effet, puisque la fonction `f` est récursive, son type n'est pas généralisé lors du typage de son corps, ainsi l'effet de `f(x)` se retrouve unifié avec le type du motif C. Or cet effet contribuant à l'effet latent de `f`, nous retrouvons bien l'exception C dans ce dernier. Bien évidemment, ceci constitue une fausse alerte que nous désirerions éliminer de notre analyse.

Partant de la constatation que c'est l'interdiction de généraliser le type d'une fonction récursive pour typer sa définition qui provoque cette fausse alerte, nous voyons maintenant l'intérêt qu'apporte le typage en présence de récursion polymorphe pour améliorer les résultats de notre analyse. En effet, avec cette récursion polymorphe, nous pourrions assigner à `f` le schéma de types $\forall \alpha, \rho.\, \alpha \xrightarrow{\rho} \texttt{unit}$, à la fois à l'intérieur et à l'extérieur de la récursion. Ce serait alors une instance fraîche de ce schéma de types qui se trouverait unifiée avec le type de C, ainsi ne « polluant » pas le schéma de types de `f`.

D'autre part, il peut être souhaitable d'autoriser un tel mécanisme de typage dans un langage comme ML, indépendamment de toute considération liée à notre analyse, pour principalement deux autres raisons. D'une part ceci permet l'utilisation de structures de données « non régulières », c'est-à-dire dont la définition de type récurse sur un argument différent de celui avec lequel elle est paramétrée. Par exemple :

```
type 'a indice = Zero | Succ of 'a
type 'a lambda =
  | Var of 'a
  | Lam of ('a indice) lambda
  | App of ('a lambda) * ('a lambda)
```

D'autre part, ceci permet d'éviter le rejet de fonctions non mutuellement récursives mais définies dans un seul et même `let rec`. Bien que cet idiome de programmation puisse être considéré comme critiquable, il n'est pas de raison réelle de refuser de tels programmes puisqu'ils restent corrects. Ainsi, le programme suivant :

```
let rec map f = function [] → [] | h : : q → f h : : map f q
and mapsucc f l = map (fun x → x + 1) l
```

serait rejeté en ML alors que nous pouvons associer les types $\forall \alpha, \beta.\, (\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$ à `map` et $\forall \alpha.\, \alpha \to \texttt{int list} \to \texttt{int list}$ à `mapsucc`.

Une façon de contourner le problème consiste à réécrire ce programme en deux fonctions définies séparément, `map`, puis `mapsucc`. Certains compilateurs [20] effectuent une analyse de dépendance afin de transformer automatiquement de tels programmes pour ensuite utiliser un algorithme d'inférence sans `let rec` polymorphe, mais ceci reste au niveau du source et ne saurait en aucun cas être considéré comme une solution au problème de fond.

8.1 Travaux connexes

Plusieurs travaux antérieurs tentent d'apporter des solutions à ce problème. En premier lieu, l'introduction du problème par [36] a donné lieu, comme précisé ci-dessus, à un semi-algorithme d'inférence de types pour un langage comme ML. Le principe consiste à rechercher un point fixe pour chaque schéma de types de la définition récursive, en partant de $\forall\alpha.\,\alpha$. Le semi-algorithme itère alors l'algorithme de typage de ML en prenant comme schémas suivants pour les identificateurs, ceux trouvés à l'itération précédente. Cette méthode présente deux inconvénients majeurs. D'une part elle occasionne un coût de traitement élevé dû aux multiples parcours sur l'arbre de syntaxe des expressions du programme à typer. D'autre part, il est impossible d'utiliser une telle technique dans un compilateur à partir du moment où le risque que ce dernier ne termine pas est présent. En effet, il n'est pas acceptable, du point de vue d'un utilisateur, que la compilation d'un programme, correct ou non, puisse ne pas terminer.

Outre établir l'équivalence entre l'inférence de types avec règle μ et la semi-unification, [19] développe également un semi-algorithme pour ce problème. À partir d'un programme à typer, il génère un ensemble de contraintes d'unification et de semi-unification, qu'il tente de résoudre ensuite. La résolution de ces contraintes est effectuée de façon plus efficace que ce qui résulte du principe itératif présenté dans [36] et les cas de non-terminaison de ce semi-algorithme sont raréfiés par l'utilisation d'un « extended occur-check » permettant l'élimination d'un grand nombre d'entre eux. Toutefois cette technique présente encore un certain nombre d'obstacles en vue d'une intégration dans un compilateur de langage à la ML :

- la forme des contraintes générées à partir des programmes à typer nécessite l'ajout d'un moteur de résolution différent de celui présent dans un typeur ML conventionnel ;
- la quantité de contraintes générées s'avère relativement importante du fait de la nécessité d'ajouter des contraintes supplémentaires afin gérer le fait que certains types, comme ceux des arguments de fonctions, ne doivent pas être généralisables ;
- l'adaptation de cet algorithme à une algèbre de types non conventionnelle comme la notre (récursive, avec effets et modulo théorie équationnelle) reste un problème ouvert ;
- enfin, le risque de non-terminaison de l'inférence de types reste présent, même si très réduit. Notons que ce dernier point pourrait vraisemblablement être levé par l'ajout d'une limite du nombre d'itérations au-delà de laquelle l'algorithme s'arrêterait en reportant un échec de typage. De plus lors qu'un tel cas apparaît, il serait possible de remplacer des contraintes de semi-unification par des contraintes d'égalité (unification avec cycle) ce qui fait perdre la principalité des types mais permet à l'algorithme de continuer à fonctionner.

8.2 Proposition et motivations

Nous présentons maintenant une nouvelle méthode pour approcher l'inférence de types dans ML en présence de `let rec` polymorphe, débouchant sur un algorithme efficace en terme de rejet abusif de programmes sémantiquement corrects, de temps et de travail d'implémentation supplémentaire par rapport à un typeur de ML conventionnel.

En particulier, la présentation du cadre théorique sur lequel repose cet algorithme se veut la plus proche possible de ce qu'il se passe dans une implémentation réelle et efficace. Autrement dit, nous voulons présenter un modèle théorique « collant de près » à une implantation réelle, afin d'établir les propriétés, non pas d'un modèle « qui serait équivalent », mais de l'algorithme lui-même, avec ses caractéristiques et optimisations qui font sa raison d'être.

Bien que la principale motivation de cette recherche provienne de la nécessité d'affiner les résultats de notre analyse d'exceptions, nous allons décrire ce nouvel algorithme dans le cadre plus général de l'inférence de types pour un langage ML. Nous énoncerons et démontrerons alors les propriétés de cet algorithme dans ce cadre, mais ne les étendrons pas aussi formellement à notre algèbre complexe servant à approximer l'ensemble des exceptions pouvant s'échapper de programmes ML. Toutefois, nous exposerons en quoi cet algorithme peut aisément être adapté à une telle algèbre, celui-ci ayant été effectivement inclu dans l'implantation réelle de notre analyseur d'exceptions. Une analyse plus détaillée des résultats pratiques obtenus lors cette implantation sera discutée ultérieurement (voir chapitre 9), en même temps que l'examen général des performance de notre système de types pour le problème de la détection d'exceptions.

Nous allons maintenant examiner les caractéristiques que nous souhaitons qu'un tel algorithme satisfasse afin, d'une part d'être inclu dans un compilateur réel de langage ML, et d'autre part de satisfaire nos exigences quant à l'analyse d'exceptions.

8.2.1 Terminaison

Comme mentionné précédemment, notre but est d'intégrer une inférence de types en présence de `let rec` polymorphe dans un compilateur ML réel. La phase de typage d'un tel compilateur permet de garantir que tout programme accepté par ce compilateur ne résultera pas en une erreur lors de l'évaluation. Il s'agit donc d'une analyse statique préliminaire à la phase de compilation proprement dite. Aussi cette phase de typage se doit de toujours terminer, que le programme à typer soit correct ou non. Cela signifie que seul un *algorithme* d'inférence (par opposition à un *semi-algorithme*) est acceptable au sein du compilateur.

L'inférence de types en présence de `let rec` polymorphe étant par essence indécidable, il va de soit que cette nécessité de garantir la terminaison de l'inférence va nous amener à un algorithme approximant de façon conservative les solutions à notre problème. Aussi, continuerons-nous à rejeter potentiellement des programmes corrects vis-à-vis de la sémantique de ML, mais dont les types n'auront pas pu être inférés par notre algorithme. Notre algorithme comporte donc un critère d'arrêt arbitraire permettant d'éviter la non-terminaison.

Ce critère se doit d'être simple de façon à rester compréhensible par le programmeur désireux de comprendre pourquoi un programme qu'il considère bien typé peut être rejeté par le système de types. Le programmeur n'étant pas forcément un spécialiste des systèmes de types, seul un modèle restant aisément explicable et pouvant s'appuyer sur des intuitions courantes peut satisfaire à cette exigence.

Ce critère doit également être efficace en terme de coût de calcul, afin de ne pas augmenter déraisonnablement le temps de typage.

Et pour finir, nous souhaitons qu'il soit suffisamment flexible pour rejeter le moins possible de programmes sémantiquement corrects. En particulier, il doit permettre impérativement de traiter les cas similaires à `map` et `mapsucc` vus précédemment.

8.2.2 Utilisabilité

Nous considérons maintenant quelques pré-requis de notre algorithme vus du côté d'un utilisateur du compilateur ML autorisant la récursion polymorphe.

Bien que le typage de ML soit connu pour être doublement exponentiel en théorie [24], en pratique il se comporte de façon quasi-linéaire, fournissant ainsi des temps d'analyse très faibles. Ceci confère à la synthèse de types un avantage certain puisqu'elle permet au programmeur de s'affranchir des déclarations de types, de bénéficier d'une vérification forte et automatique de ses programmes, et ceci à un coût très réduit. Afin de conserver l'utilisabilité de ce mécanisme, nous souhaitons que l'ajout du `let rec` polymorphe dans le compilateur ne se fasse pas au détriment de la rapidité de la phase de typage. Aussi notre algorithme se devra-t'il d'être efficace en temps de calcul.

L'explicitation des erreurs de typage à destination du programmeur est un problème délicat, même au sein d'un compilateur utilisant un système de types conventionnel. Nous ne voulons pas que la présence de `let rec` polymorphe nuise à la clarté de ces messages. En particulier, nous souhaitons obtenir des messages d'erreurs dans la même philosophie que ceux auquel le programmeur est habitué lors de l'utilisation d'un compilateur ML.

Cela revient à dire que notre algorithme ne doit pas se manifester extérieurement par d'autres effets que la possibilité de typer plus de programmes qu'avec un système de types conventionnel. Les erreurs de typage survenant au cours du typage d'un `let rec` devront donc s'insérer dans le cadre déjà existant du système de types de ML.

8.2.3 Modifications minimales

Nous avons conçu le présent algorithme dans le but de pouvoir l'adjoindre à un typeur déjà existant. Nous voulions donc un algorithme nécessitant le minimum de modifications du-dit typeur afin d'éviter un surcroît de travail pour le concepteur du compilateur et de pouvoir s'intégrer, non pas dans une implémentation particulière de ML, mais *a priori* dans une large catégorie de compilateurs. Plus précisément, les compilateurs concernés sont ceux utilisant un mécanisme de « niveaux » pour gérer la généralisation des types (en vue d'en faire des schémas), principe relativement répandu dans les implémentations réelles, car il contribue à une bonne efficacité du typeur en terme de temps de réponse [62, 32].

De plus, il est nécessaire qu'en cas d'échec de l'inférence utilisant la récursion polymorphe, il soit possible de retourner aisément dans un mode de typage à la ML conventionnel. Ceci est rendu obligatoire par le fait que notre critère d'arrêt peut rejeter un programme qui serait typable avec le système de types standard. Dans un tel cas, il est impensable d'échouer, ce qui rejetterait des programmes qui auraient été correctement typés avec le système de types de ML. Il nous faut donc, dans ce cas, un moyen efficace de se replier sur l'inférence de types conventionnelle.

La première solution consiste à oublier tout le travail effectué jusqu'à présent et à rappeler le mécanisme d'inférence d'origine. La seconde consiste à exploiter les informations de type accumulées pendant la phase de typage polymorphe pour simuler le typage du `let rec` monomorphe. Dans cette seconde optique, nous voyons qu'il n'est plus nécessaire de conserver un algorithme de typage pour le cas polymorphe et un autre algorithme pour le cas monomorphe. Ceci est alors « un plus » qui permet, d'une part un gain en efficacité, et d'autre part d'éviter une « duplication » de code dans le compilateur, donc minimise les modifications à effectuer dans son implémentation.

Un tel mécanisme permet en outre, en tant que résultat dérivé, d'unifier le traitement du typage des constructions `let` et `let rec`, factorisant ainsi un peu plus l'implémentation du typeur.

8.2.4 Capacité d'adaptation

L'algèbre de types de ML peut-être étendue dans diverses directions, par des types rationnels [47, 45, 49] ou des effets [56, 63] comme c'est le cas pour notre analyse d'exceptions. Il peut être intéressant, voire nécessaire, d'ajouter l'inférence de types en présence de `let` `rec` polymorphe dans de tels systèmes de types. Notre algorithme doit donc pouvoir s'adapter à d'autres algèbres que celle de ML. Nous verrons que le fait de s'appuyer le plus possible sur les mécanismes et propriétés de base fournis par le typeur d'origine (unification, instanciation et généralisation) nous permet de garantir une intégration très aisée dans ces genres de contextes. Nous verrons quelles sont les opérations à fournir sur l'algèbre de types de façon à garantir cette intégration.

8.3 Trois systèmes de types pour ML

8.3.1 Le langage source et son algèbre de types

Le langage source que nous considérons dans cette présentation est un sous-ensemble simple de ML avec les entiers comme seule structure de données de base, les identificateurs, les fonctions, l'application et la possibilité de définir (récursivement ou non) des identificateurs au sein d'une expression. En particulier, afin de simplifier l'exposé, nous abandonnons les constructions ayant trait aux exceptions que nous avons introduites lors de la présentation de l'analyse proprement dite. L'extension à ces constructions, et plus généralement à un langage complet s'effectue de façon triviale, le cœur du problème se trouvant posé uniquement sur les *définitions* et les *utilisations* d'identificateurs.

Le langage sur lequel nous allons travailler se présente donc sous la forme :

Expressions :

$a \ ::= x$	identificateur
$\mid i$	constante entière
$\mid \lambda x.a$	abstraction (fonction)
$\mid a_1(a_2)$	application
$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	liaison d'identificateur
$\mid \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2$	liaison récursive d'identificateur

La majorité des constructions ont déjà été présentées en section 2.1. Seules deux nouvelles constructions apparaissent ici :

- $\texttt{let}\ \mathrm{x} = a_1\ \texttt{in}\ a_2$ permet de lier l'identificateur x à la valeur de l'expression a_1 (dans laquelle x ne peut pas être utilisé) pour évaluer l'expression a_2.
- $\texttt{let rec}\ x = a_1\ \texttt{in}\ a_2$ permet de lier *récursivement* l'identificateur x à la valeur de l'expression a_1 (dans laquelle x peut alors apparaître) pour ensuite évaluer l'expression a_2.

L'algèbre de types associée à ce langage est celle habituellement présentée dans ML, ressemblant à celle de notre système d'analyse d'exceptions sans les effets. Nous rappelons cette algèbre afin d'éviter au lecteur de se reporter aux sections précédentes :

Expressions de type :

$\tau \ ::= \alpha$	variable de type
$\mid \texttt{int}$	type entier
$\mid \tau_1 \rightarrow \tau_2$	type fonctionnel

8.3.2 Le(s) système(s) de types

Afin d'exposer notre algorithme, nous allons successivement envisager trois variantes du système de types de ML. Chacun d'entre eux reposera sur les expressions de types présentées ci-dessus. C'est sciemment que nous ne définissons pas les schémas de types maintenant car ceux-ci vont prendre successivement trois formes différentes au fur et à mesure des systèmes de types que nous étudierons.

Afin de procurer du polymorphisme, le système de types de ML requiert deux notions distinctes : les *types* et les *schémas de types*. On peut considérer ces derniers comme des « moules » permettant, par *instanciation*, d'obtenir différentes *instances* de types ayant un certain squelette.

Ainsi, les environnements de typage contiennent des couples (identificateur $\times$ schéma de types). Seules les constructions `let` et `let rec` introduisent du polymorphisme, c'est-à-dire des schémas de types non triviaux (dont la liste des variables généralisées n'est pas vide) dans l'environnement de typage. Lors de l'utilisation d'un identificateur, il y a *instanciation* du schéma auquel il est lié (c'est-à-dire copie de son corps avec remplacement des variables généralisées par des variables fraîches, c'est-à-dire n'existant dans aucun type du problème).

Usuellement, la gestion de la généralisabilité des types est présentée en termes de variables libres / liées dans l'environnement de typage. Les implémentations réelles de compilateurs utilisant le typage de ML implantent cette gestion de façon plus efficace : par l'intermédiaire de « niveaux de généralisation », *sans faire de distinction entre types et schémas de types*. Autrement dit, dans une telle présentation, les environnements contiennent des couples (identificateur $\times$ type).

Puisque notre algorithme est conçu pour être implanté dans un compilateur réel, nous allons l'exposer dans le cadre d'un système de types « avec niveaux ». Ceci va donc nous obliger à formaliser le système de types de ML avec récursion polymorphe dans ce cadre. Pour y parvenir, nous allons étudier trois versions différentes de notre système, la première s'appuyant sur la présentation habituelle de tels systèmes, et les deux suivantes étant chacune dérivées de la précédente. Nous montrerons qu'il y a équivalence entre chacune de ces dérivations afin d'établir la correction de ce mécanisme vis-à-vis de la présentation théorique usuellement adoptée.

Munis de la dernière présentation de notre système de types, nous pourrons ensuite aborder l'inférence de types.

Le système de types de ML usuel (avec schémas) (S_1)

Le système de types de ML est généralement étudié en utilisant des schémas de types comportant, en tête et quantifiées universellement, les variables généralisées dont on prendra une copie lors d'une instanciation du schéma considéré. La gestion de la généralisabilité des variables est alors effectuée en terme de variables libres dans l'environnement. Nous ne détaillons pas ici la présentation des notions sous-tendant ce système (schémas, types, environnements, etc...) celle-ci ayant déjà été réalisée au début du chapitre 4 et nous nous concentrons directement sur ses règles de typage :

Instanciation et généralisation :

$\tau' \leq \sigma$ si σ est de la forme $\forall \alpha_i. \tau$ et il existe τ_i tels que $\tau' = \tau\{\alpha_i \leftarrow \tau_i\}$.

$\mathrm{Gen}\,(\tau, E)$ est $\forall \alpha_i. \tau$ avec $\{\alpha_i\} = \mathcal{L}(\tau) \setminus \mathcal{L}(E)$.

Typage des expressions :

$$\frac{\tau \leq E(x)}{E \vdash x : \tau} \ (47) \qquad E \vdash i : \texttt{int} \ (48) \qquad \frac{E \vdash a_1 : \tau_1 \to \tau \qquad E \vdash a_2 : \tau_1}{E \vdash (a_1\ a_2) : \tau} \ (49)$$

$$\frac{E \oplus (x : \tau_1) \vdash a_1 : \tau_2}{E \vdash \lambda x.a_1 : \tau_1 \to \tau_2} \ (50) \qquad \frac{E \vdash a_1 : \tau_1 \qquad \sigma = \mathrm{Gen}\,(\tau_1, E) \qquad E \oplus (x : \sigma) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2} \ (51)$$

$$\frac{E \oplus (x : \sigma) \vdash a_1 : \tau_1 \qquad \sigma = \mathrm{Gen}\,(\tau_1, E) \qquad E \oplus (x : \sigma) \vdash a_2 : \tau_2}{E \vdash \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \tau_2} \ (52)$$

A l'exception de la règle (52) traitant le `let rec`, ce système de types est identique à celui de ML. C'est par l'intermédiaire de cette dernière que la récursion polymorphe apparaît, en autorisant le typage de l'expression définissante associée à l'identificateur `x` (c'est-à-dire l'expression a_1) sous l'hypothèse que dans l'environnement de typage ce dernier n'est non pas lié à un type, mais à un *schéma de types* σ.

Pour les besoins de l'exemple qui va suivre, considérons que notre langage est étendu par une expression conditionnelle (`if ...then ...else ...`) et les valeurs de vérité (`true` et `false`, de type `bool`). L'expression :

```
let rec f x = if f 1 then f true else false in f true
```

mal typée avec les règles habituelles de ML est maintenant acceptée avec le système ci-dessus. En effet, la fonction `f` est utilisée à l'intérieur de son corps avec le type $\texttt{int} \to \texttt{bool}$ (à l'intérieur de la condition), et avec le type $\texttt{bool} \to \texttt{bool}$ (pour la première alternative de la conditionnelle). Prenons $\sigma = \forall \alpha.\, \alpha \to \texttt{bool}$ dans la règle (52). Alors la conditionnelle se trouvant dans le corps de l'expression de fonction liée à `f` est bien typée puisque les occurrences de `f` utilisées en condition et en première alternative de l'expression conditionnelle ont bien des types instances de σ. Nous en déduisons que l'expression fonctionnelle liée à `f` est de type $\alpha \to \texttt{bool}$. Nous pouvons alors typer la seconde partie de notre `let rec`, à savoir `f` 1, qui nous amène au type `bool`.

Le système de types de ML avec schémas et niveaux (S_2)

La seconde présentation que nous étudions maintenant se trouve à mi-chemin entre la présentation habituelle et celle généralement implantée dans les typeurs réels. Elle combine à la fois la notion de schémas et celle de niveaux de généralisation.

Chaque variable de type comporte désormais l'indication du niveau *auquel elle a été créée*. Les jugements de typage sont maintenant établis à un *niveau courant*. Ainsi, $E \vdash_n a : \tau$ se lit comme : « sous l'environnement de typage E, au niveau de généralisation courant n, l'expression a a le type τ ». Nous donnons ici informellement quelques propriétés relatives à ces jugements afin d'apporter une vue intuitive de cette notion de niveaux :

- n est le niveau maximum des types pouvant apparaître dans τ ;
- n est le niveau auquel on étiquette les variables que l'on peut créer si nécessaire afin de typer l'expression courante a ;
- l'environnement de typage E ne peut contenir que des variables de niveau $\leq n$.

Nous démontrerons ultérieurement ces propriétés. L'algèbre de types prend donc la forme suivante :

Expressions de type :

$$\begin{array}{lll} \tau & ::= \alpha^n & \text{variable de type de niveau } n \\ & \mid \texttt{int} & \text{type entier} \\ & \mid \tau_1 \rightarrow \tau_2 & \text{type fonctionnel} \end{array}$$

Schémas de types :

$$\sigma \ ::= \forall \alpha_1 \ldots \alpha_k . \tau$$

Définition 8.1 (Type de niveau n au plus) *Le type τ est dit de « niveau n » et noté $\tau \in \tau^{\leq n}$ s'il ne comporte que des variables créées à un niveau au plus égal à n (c'est-à-dire de niveau $\leq n$).*

Définition 8.2 (Variable de niveau n au moins dans un type) *Soit un type τ. On note $\mathcal{V}^{\geq n}(\tau)$ l'ensemble des variables de type présentes dans τ dont le niveau est supérieur ou égal à n.*

Instanciation :

$\tau' \leq_n \sigma$ si σ est de la forme $\forall \alpha_i . \tau$ et il existe $\tau_i \in \tau^{\leq n}$ tels que $\tau' = \tau\{\alpha_i \leftarrow \tau_i\}$.

Typage des expressions :

$$\frac{\tau \leq_n E(x)}{E \vdash_n x : \tau} \ (53) \qquad E \vdash_n i : \texttt{int} \ (54) \qquad \frac{E \vdash_n a_1 : \tau_1 \rightarrow \tau \qquad E \vdash_n a_2 : \tau_1}{E \vdash_n (a_1 \ a_2) : \tau} \ (55)$$

$$\frac{E \oplus (x : \tau_1) \vdash_n a_1 : \tau_2 \qquad \tau_1 \in \tau^n}{E \vdash_n \lambda x . a_1 : \tau_1 \rightarrow \tau_2} \ (56)$$

$$\frac{E \vdash_{n+1} a_1 : \tau_1 \qquad \sigma = \forall \, \mathcal{V}^{\geq n+1}(\tau_1) . \tau_1 \qquad E \oplus (x : \sigma) \vdash_n a_2 : \tau_2}{E \vdash_n \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2} \ (57)$$

$$\frac{E \oplus (x : \sigma) \vdash_{n+1} a_1 : \tau_1 \qquad \sigma = \forall \, \mathcal{V}^{\geq n+1}(\tau_1) . \tau_1 \qquad E \oplus (x : \sigma) \vdash_n a_2 : \tau_2}{E \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \tau_2} \ (58)$$

Ce système permet donc d'introduire les niveaux de généralisation sans les déconnecter des schémas de types. Ceci nous permettra d'établir plus aisément le lien qu'il existe entre ces deux notions lors du passage à la prochaine présentation du système de types.

Notons à nouveau que l'introduction de la récursion polymorphe s'effectue au niveau de la règle 58 qui, de la même façon que dans la présentation précédente, type l'expression définissante associée à l'identificateur `x` sous l'hypothèse que ce dernier est lié à un schéma de types.

Remarquons également que les généralisations effectuées dans les règles 57 et 58 sont faites sans référence à l'environnement de typage E.

Le système de types de ML sans schémas, avec niveaux (S_3)

Nous abordons maintenant la présentation finale du système de types, reflétant la structure couramment utilisée dans les implantations réelles. La notion de schéma de types disparaît maintenant totalement au profit des niveaux.

Les environnements ne contiennent plus de couples (identificateur $\times$ schéma), mais des couples (identificateur $\times$ (type $\times$ niveau)). $E(x) = (\tau, m)$ signifie alors que « toutes les variables de type de niveau *strictement* supérieur à m se trouvant dans τ sont généralisées ».

Ainsi, pour chacune d'entre elles, lors d'une prise d'instance par la règle 59 une copie sera effectuée.

Définition 8.3 (Relation d'instance entre types) *Un type τ' est une instance d'un type τ, et l'on note $\tau \leq \tau'$ s'il existe une substitution φ telle que $\tau' = \varphi(\tau)$. On dit alors que τ* généralise *(en Anglais* subsume*) τ' ou que τ'* est filtré par *(en Anglais* matche*) τ.*

On alors étend la notion d'instance entre types aux types avec niveaux.

Définition 8.4 (Instance de niveau (m, n)) *Un type τ' est une instance de niveau (m, n) d'un type τ, et l'on note $\tau' \leq_n^m \tau$ s'il existe une substitution φ telle que $\tau' = \varphi(\tau)$ et* $\mathrm{Dom}\,(\varphi) \subseteq \mathcal{V}^{\geq m+1}(\tau)$ *et* $\mathrm{Rng}\,(\varphi) \subseteq \tau^{\leq n}$. *Autrement dit, φ est une substitution qui envoie toutes les variables de niveau supérieur ou égal à $m+1$ présentes dans τ vers des types de niveau n au plus.*

Typage des expressions :

$$\frac{E(x) = (\tau, m) \qquad \tau \leq_n^m \tau'}{E \vdash_n x : \tau'} \ (59) \qquad\qquad E \vdash_n i : \mathtt{int} \ (60)$$

$$\frac{E \vdash_n a_1 : \tau_1 \to \tau \qquad E \vdash_n a_2 : \tau_1}{E \vdash_n (a_1\ a_2) : \tau} \ (61) \qquad \frac{E \oplus (x : (\tau_1, n)) \vdash_n a_1 : \tau_2 \qquad \tau_1 \in \tau^n}{E \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2} \ (62)$$

$$\frac{E \vdash_{n+1} a_1 : \tau_1 \qquad E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2}{E \vdash_n \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2} \ (63)$$

$$\frac{E \oplus (x : (\tau_1, n)) \vdash_{n+1} a_1 : \tau_1 \qquad E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2}{E \vdash_n \mathtt{let\ rec}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2} \ (64)$$

Dans cette présentation du système de types, deux règles sont particulièrement intéressantes : 64 (pour les liaisons `let rec`) et 59 (pour les identificateurs).

La première permet, comme précédemment, d'introduire la récursion polymorphe dans le système de types. Maintenant que la notion de schéma a disparu, nous ne pouvons plus dire que le corps de l'expression définissante est typé sous l'hypothèse que l'identificateur est lié à un schéma. Le niveau $(n+1)$ auquel on type le corps de l'expression définissante va jouer ce rôle.

Introduisons par une intuition l'effet de typer l'expression définissante au niveau $n+1$ plutôt qu'au niveau n. Lorsque nous allons typer le corps de l'expression définissante, le niveau courant sera donc $n+1$. C'est-à-dire que toute instance de l'identificateur défini récursivement contiendra des sous-termes de niveau $n+1$. Ainsi, lier le type τ au niveau n permet de le considérer comme polymorphe puisque ces sous-termes de niveau $n+1$ sont considérés généralisés par définition de (τ, m) (cf. la remarque en début de la présentation de ce système).

Remplacer $\vdash_{n+1}$ par $\vdash_n$ dans la prémisse de gauche de la règle `LETREC` nous interdirait tout polymorphisme, y compris dans le corps du `let`, puisque les sous-termes composant τ_1 seraient tous de niveau $\leq n$, et τ_1 étant lié au niveau n, seules les parties de niveau supérieur ou égal à $n+1$ (donc aucune) seraient considérées généralisables dans τ_1, ni dans a_1, ni dans a_2 (puisque dans a_2 nous typons également au niveau n).

Par contre, remplacer $(x : (\tau_1, n))$ par $(x : (\tau_1, n+1))$ dans la prémisse de gauche de la règle `LETREC` nous ramènerait au système de types de ML sans récursion polymorphe. En effet, lors du typage de a_1, les sous-termes composant τ_1 seraient tous de niveau $\leq n+1$, et τ_1 étant lié au niveau $n+1$, seules les parties de niveau supérieur ou égal à $n+1$ (donc aucune) seraient considérées généralisables dans τ_1. Ensuite, lors du typage de a_2, nous

nous retrouverions au niveau n (qui est donc inférieur à $n+1$) et donc les parties de niveau $n+1$ de τ_1 seraient bien considérées comme généralisées, fournissant le polymorphe à la ML attendu.

Le règle 59 spécifie quant à elle, le fait que toute instance d'un type lié à un identificateur dans un environnement de typage est prise au niveau courant (n). Elle exprime également que toutes les variables de types présentes dans τ et de niveau supérieur ou égal à m sont considérées généralisées. Ainsi nous retrouvons bien le comportement présent dans la présentation initiale, autorisant les variables de types d'un schéma à être « remplacées » par des variables fraîches du niveau courant. C'est le mécanisme de copie « par moule » que nous avons décrit précédemment.

Ainsi $\tau \leq_n^m \tau'$ peut être intuitivement entendu comme « au niveau courant n, on considère que les variables, de niveau $m+1$ et plus, présentes dans τ sont généralisées, et donc on en prend une copie avant de les instancier pour obtenir τ' ».

8.3.3 Équivalence entre les trois systèmes

L'algorithme que nous allons présenter repose sur le dernier système que nous avons étudié, lui-même dérivé des deux précédents. Plutôt que de démontrer à nouveaux toutes les propriétés attendues pour un système de types dans le cadre du dernier que nous avons vu, nous préférons montrer l'équivalence entre chaque dérivation successive du système de types, ce qui montrera par transitivité, l'équivalence entre le système originel et celui sur lequel fonctionne notre algorithme. Deux systèmes seront dits « équivalents » s'ils permettent de typer le même ensemble de programmes clos. Ainsi allons-nous montrer l'équivalence entre S_1 et S_2, puis entre S_2 et S_3. Chacune elles sera démontrée par deux implications de sens contraire.

Les preuves qui suivent s'appuient sur le théorème suivant que nous ne démontrerons pas dans le cadre de cette thèse. Le lecteur intéressé pourra consulter [45] afin de trouver la démonstration de théorème.

Théorème 8.1 (Existence d'unificateur préservant les niveaux) *Soit E un ensemble de contraintes d'unification. Si E admet un unificateur, alors il admet également un unificateur principal préservant les niveaux.*

S_2 implique S_1

Définition 8.5 (Effacement des niveaux) *On définit l'opération permettant de passer d'un type avec niveaux τ du système S_2 à un type sans niveaux τ' du système S_1, et le note $\overline{\tau} = \tau'$, par :*

$$\begin{array}{lcl} \overline{\alpha^n} & = & \alpha \\ \overline{\texttt{int}} & = & \texttt{int} \\ \overline{\tau_1 \to \tau_2} & = & \overline{\tau_1} \to \overline{\tau_2} \end{array}$$

Les variables quantifiées dans les schémas de types de S_2 ne portant pas de niveau, on étend l'opération d'effacement des niveaux à un schéma de types de la façon suivante :

$$\overline{\forall \alpha_i . \tau} = \forall \alpha_i . \overline{\tau}$$

On étend simplement cette notion à un environnement de typage E de S_2 par effacement des niveaux dans chaque liaison (c'est-à-dire à chaque couple (identificateur $\times$ schéma de types)) de cet environnement.

Lemme 8.1 (Commutation entre généralisation et traduction) *Soient un environnement de typage E et un type τ dans S_2. Si toutes les variables libres de E sont de niveau $\leq n$ alors* $\overline{\forall \mathcal{V}^{\geq n+1}(\tau).\, \tau} = \mathrm{Gen}\,(\overline{\tau}, \overline{E})$.

Preuve : Par définition, $\mathrm{Gen}\,(\overline{\tau}, \overline{E}) = \forall \{\alpha_1 \ldots \alpha_k\}.\overline{\tau}$ avec $\{\alpha_1 \ldots \alpha_k\} = \mathcal{L}(\overline{\tau}) \setminus FV(\overline{E})$.
Prouvons alors que $\forall \mathcal{V}^{\geq n+1}(\tau).\, \tau \equiv \forall \{\alpha_1 \ldots \alpha_k\}.\overline{\tau}$ où $\{\alpha_1 \ldots \alpha_k\} = \mathcal{L}(\overline{\tau}) \setminus \mathcal{L}(\overline{E})$.
Ceci revient donc à montrer que $\mathcal{V}^{\geq n+1}(\tau) = \mathcal{L}(\overline{\tau}) \setminus \mathcal{L}(\overline{E})$.
Cette propriété revient alors à dire que l'environnement de typage E ne contient que des variables de niveau $\leq n$, ce qui est une hypothèse de notre lemme.

□

Théorème 8.2 (S_2 implique S_1) *Soient un programme a et un environnement de typage E. Si $E \vdash_n a : \tau$ dans S_2 et si toutes les variables libres de E sont de niveau $\leq n$, alors $\overline{E} \vdash a : \overline{\tau}$. En conséquence, un programme clos typable dans S_2 est aussi typable dans S_1.*

Preuve : La preuve est effectuée par cas sur la forme de a.

Cas a est x.
Hypothèse :

$$E \vdash_n x : \tau' \tag{8.1}$$

Nous voulons prouver que dans S_1 :

$$\overline{E} \vdash x : \overline{\tau'} \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une dérivation 47 de S_1 de la forme :

$$\frac{\ldots}{\overline{E} \vdash x : \overline{\tau'}}$$

et donc nous devons en prouver la prémisse, à savoir :

$$\overline{\tau'} \leq \overline{E}(x) \tag{8.3}$$

Par l'hypothèse 8.1, en utilisant une règle (53) de S_2, nous déduisons que nous avons :

$$\tau' \leq E(x) \tag{8.4}$$

Donc, par définition de $\leq$ dans S_2 appliquée à 8.4, nous avons :

$$E(x) \text{ est de la forme } \forall \alpha_i.\, \tau \tag{8.5}$$
$$\tau' = \tau\{\alpha_i \leftarrow \tau_i\} \text{ où } \tau_i' \in \tau^{\leq n} \tag{8.6}$$

Effaçons simplement les niveaux dans 8.5 et 8.6, il nous vient alors que :

$$\overline{E}(x) \text{ est de la forme } \forall \alpha_i.\overline{\tau} \tag{8.7}$$
$$\overline{\tau'} = \tau\{\alpha_i \leftarrow \overline{\tau_i}\} \tag{8.8}$$

ce qui, par définition de $\leq$ dans S_1 est équivalent à $\overline{\tau'} \leq \overline{E}(x)$ comme attendu en 8.2. Nous pouvons ainsi, à partie de cette prémisse, reconstituer une dérivation 47 et déduire que $\overline{E} \vdash x : \overline{\tau'}$ comme attendu en 8.2.

Cas a est i.

Hypothèse :

$$E \vdash_n i : \texttt{int} \tag{8.1}$$

Nous voulons prouver que dans S_1 :

$$\overline{E} \vdash i : \overline{\texttt{int}} \tag{8.2}$$

Ce cas est trivial puisque $\overline{\texttt{int}} = \texttt{int}$. Ainsi pouvons-nous immédiatement déduire l'existence d'une dérivation 48 dans S_1 qui nous donne le résultat attendu en 8.2 et clore ce cas de la preuve.

Cas a est $(a_1\ a_2)$.
Hypothèses :

$$E \vdash_n (a_1\ a_2) : \tau \tag{8.1}$$
$$\forall \alpha^l \in \mathcal{L}(E), l \leq n \tag{8.2}$$

Nous voulons prouver que dans S_1 :

$$\overline{E} \vdash (a_1\ a_2) : \overline{\tau} \tag{8.3}$$

Pour y parvenir, nous voulons utiliser une dérivation 49 de S_1 de la forme :

$$\frac{\ldots \quad \ldots}{\overline{E} \vdash (a_1\ a_2) : \overline{\tau}}$$

et donc nous devons en prouver les prémisses, à savoir :

$$\overline{E} \vdash a_1 : \overline{\tau_1 \to \tau} \tag{8.4}$$
$$\overline{E} \vdash a_2 : \overline{\tau_1} \tag{8.5}$$

De 8.1, par utilisation d'une règle (55) de S_2, nous déduisons que :

$$E \vdash_n a_1 : \tau_1 \to \tau \tag{8.6}$$
$$E \vdash_n a_2 : \tau_1 \tag{8.7}$$

Nous pouvons alors appliquer directement l'hypothèse de récurrence sur 8.6 et 8.7 pour en déduire les prémisses attendues en 8.4 et 8.5. Nous pouvons ainsi reconstituer une dérivation 49 qui nous donne le résultat attendu en 8.1.

Cas a est $\lambda x.a_1$.
Hypothèses :

$$E \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2 \tag{8.1}$$
$$\forall \alpha^l \in \mathcal{L}(E), l \leq n \tag{8.2}$$

Nous voulons prouver que dans S_1 :

$$\overline{E} \vdash \lambda x.a_1 : \overline{\tau_1 \to \tau_2} \tag{8.3}$$

Pour y parvenir, nous voulons utiliser une dérivation 50 de S_1 de la forme :

$$\frac{\ldots}{\overline{E} \vdash \lambda x.a_1 : \overline{\tau_1 \to \tau_2}}$$

et donc nous devons en prouver la prémisse, à savoir :

$$\overline{E} \oplus (x : \overline{\tau_1}) \vdash a_1 : \overline{\tau_2} \tag{8.4}$$

À partir de 8.1, par application d'une règle 56 de S_2, nous déduisons que :

$$E \oplus (x : \tau_1) \vdash_n a_1 : \tau_2 \tag{8.5}$$
$$\tau_1 \in \tau^n \tag{8.6}$$

Puisque, par 8.6 $\tau_1 \in \tau^n$, nous avons que $\mathcal{L}(E \oplus (x : \tau_1))$ ne contient que des variables de niveau $\leq n$. Nous pouvons alors simplement appliquer l'hypothèse de récurrence sur 8.5 et nous déduisons directement la prémisse attendue en 8.4. Ainsi nous pouvons reconstruire la dérivation souhaitée afin de satisfaire 8.1.

Cas a est $\texttt{let } x = a_1 \texttt{ in } a_2$.
Hypothèse :

$$E \vdash_n \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2 \tag{8.1}$$
$$\forall \alpha^l \in \mathcal{L}(E), l \leq n \tag{8.2}$$

Nous voulons prouver que dans S_1 :

$$\overline{E} \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \overline{\tau_2} \tag{8.3}$$

Pour y parvenir, nous voulons utiliser une dérivation 51 de S_1 de la forme :

$$\frac{\ldots \quad \ldots \quad \ldots}{\overline{E} \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \overline{\tau_2}}$$

dont nous devons donc en établir les prémisses, à savoir :

$$\overline{E} \vdash a_1 : \overline{\tau_1} \tag{8.4}$$
$$\sigma = \operatorname{Gen}(\overline{\tau_1}, \overline{E}) \tag{8.5}$$
$$\overline{E} \oplus (x : \sigma) \vdash a_2 : \overline{\tau_2} \tag{8.6}$$

De 8.1, par utilisation d'une règle 57 de S_2, nous déduisons que :

$$E \vdash_{n+1} a_1 : \tau_1 \tag{8.7}$$
$$\sigma = \forall\, \mathcal{V}^{\geq n+1}(\tau_1).\, \tau_1 \tag{8.8}$$
$$E \oplus (x : \sigma) \vdash_n a_2 : \tau_2 \tag{8.9}$$

Par l'hypothèse 8.2, les variables libres de E étant de niveau $\leq n$, elle sont également de niveau $\leq n+1$. Nous pouvons donc appliquer l'hypothèse de récurrence à 8.7 et déduire que :

$$\overline{E} \vdash a_1 : \overline{\tau_1} \tag{8.10}$$

résultat attendu en 8.4.

Puisque par le lemme 8.1 (commutation entre généralisation et traduction), $\overline{E \oplus (x : \sigma)} \equiv \overline{E} \oplus (x : \sigma)$ (en effet, par l'hypothèse 8.2 les variables de E sont de niveau $\leq n$, par 8.8 les variables libres dans σ sont celles de niveau $< n+1$ donc $\leq n$, ainsi les variables libres dans $E \oplus (x : \sigma)$ sont bien toutes de niveau $\leq n$), nous pouvons appliquer l'hypothèse de récurrence sur 8.9 et en déduire directement notre but 8.6. Ainsi pouvons-nous reconstituer une dérivation 51 dans S_1, déduire le résultat attendu en 8.3 et clore ce cas de la preuve.

Cas a est $\texttt{let rec } x = a_1 \texttt{ in } a_2$.
Hypothèse :

$$E \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \tau_2 \tag{8.1}$$
$$\forall \alpha^l \in \mathcal{L}(E), l \leq n \tag{8.2}$$

Nous voulons prouver que dans S_1 :

$$\overline{E} \vdash \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \overline{\tau_2} \tag{8.3}$$

Pour y parvenir, nous voulons utiliser une dérivation 52 de S_1 de la forme :

$$\frac{\ldots \quad \ldots \quad \ldots}{\overline{E} \vdash \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \overline{\tau_2}}$$

dont nous devons donc en établir les prémisses, à savoir :

$$\overline{E} \oplus (x : \sigma) \vdash a_1 : \overline{\tau_1} \tag{8.4}$$
$$\sigma = \mathrm{Gen}\,(\overline{\tau_1}, \overline{E}) \tag{8.5}$$
$$\overline{E} \oplus (x : \sigma) \vdash a_2 : \overline{\tau_2} \tag{8.6}$$

De 8.1, par utilisation d'une règle 58 de S_2, nous déduisons que :

$$E \oplus (x : \sigma) \vdash_{n+1} a_1 : \tau_1 \tag{8.7}$$
$$\sigma = \forall\, \mathcal{V}^{\geq n+1}(\tau_1).\, \tau_1 \tag{8.8}$$
$$E \oplus (x : \sigma) \vdash_n a_2 : \tau_2 \tag{8.9}$$

Puisque par le lemme 8.1 (commutation entre généralisation et traduction), $E \oplus (x : \sigma) \equiv \overline{E} \oplus (x : \sigma)$ (en effet, par l'hypothèse 8.2 les variables de E sont de niveau $\leq n$ donc $\leq n+1$, par 8.8 les variables libres dans σ sont celles de niveau $< n+1$ donc $\leq n$, ainsi les variables libres dans $E \oplus (x : \sigma)$ sont bien toutes de niveau $\leq n$), nous pouvons appliquer l'hypothèse de récurrence sur 8.7 et en déduire que :

$$\overline{E} \oplus (x : \sigma) \vdash a_1 : \overline{\tau_1} \tag{8.10}$$

résultat attendu en 8.4.

Par un raisonnement identique, nous pouvons appliquer l'hypothèse de récurrence sur 8.9 pour en déduire notre but 8.6. Ainsi pouvons-nous reconstituer une dérivation 52 dans S_1, déduire le résultat attendu en 8.3 et clore ce cas de la preuve.

□

S_3 implique S_2

Définition 8.6 (Traduction d'un environnement) *Soit (τ, m) une information de type liée à un identificateur dans un environnement de typage E du système S_3. On appelle « traduction » de (τ, m) et l'on note $[\![(\tau, m)]\!]$ le schéma $\forall\, \vec{\alpha}.\, \tau$ où $\vec{\alpha} = \mathcal{V}^{\geq m+1}(\tau)$.*

Théorème 8.3 (S_3 implique S_2) *Soient un programme a et un environnement de typage E. Si $E \vdash_n a : \tau$ dans S_3 alors $[\![E]\!] \vdash a : \tau$. En conséquence, un programme clos typable dans S_3 est aussi typable dans S_2.*

Preuve : La preuve est effectuée par cas sur la forme des dérivations de typage.

Cas a est x.

Hypothèse :

$$E \vdash_n x : \tau' \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$[\![E]\!] \vdash_n x : \tau' \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une dérivation 53 de S_2 de la forme :

$$\frac{\dots}{[\![E]\!] \vdash_n x : \tau'}$$

et donc nous devons en prouver la prémisse, à savoir :

$$\tau' \leq [\![E]\!](x) \tag{8.3}$$

Par application d'une règle 59 de S_3 sur 8.1, nous déduisons que $\tau' \leq E(x)$, c'est-à-dire que :

$$\tau' = \varphi(\tau) \text{ où } \varphi = \forall \alpha_i \in \mathcal{V}^{\geq m+1}(\tau), \{\alpha_i \leftarrow \tau^{\leq n}\} \tag{8.4}$$

ce qui par définition de $[\![\ ]\!]$, est équivalent à $\tau' \leq [\![E]\!](x)$ comme attendu en 8.3. Nous pouvons donc reconstituer une dérivation 53 comme attendu en 8.2.

Cas a est i.
Hypothèse :

$$E \vdash_n i : \mathtt{int} \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$[\![E]\!] \vdash_n i : \mathtt{int} \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 54 dans S_2 ce qui est trivialement possible à partir de l'hypothèse 8.1 et nous permet de conclure immédiatement ce cas.

Cas a est $(a_1\ a_2)$.
Hypothèse :

$$E \vdash_n (a_1\ a_2) : \tau \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$[\![E]\!] \vdash_n (a_1\ a_2) : \tau \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 55 dans S_2 de la forme :

$$\frac{\dots \quad \dots}{[\![E]\!] \vdash_n (a_1\ a_2) : \tau}$$

Nous devons donc en établir les prémisses, à savoir :

$$[\![E]\!] \vdash_n a_1 : \tau_1 \to \tau \tag{8.3}$$
$$[\![E]\!] \vdash_n a_2 : \tau_1 \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 61 dans S_3 et nous déduisons que :

$$E \vdash_n a_1 : \tau_1 \to \tau \tag{8.5}$$
$$E \vdash_n a_2 : \tau_1 \tag{8.6}$$

Nous appliquons alors l'hypothèse de récurrence sur 8.5 et 8.6 qui nous donne directement les prémisses attendues en 8.3 et 8.4. Nous pouvons donc reconstituer notre dérivation et nous obtenons bien le résultat attendu en 8.2.

Cas a est $\lambda x.a_1$.

Hypothèse :

$$E \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2 \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$[\![E]\!] \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2 \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 56 dans S_2 de la forme :

$$\frac{\ldots \quad \ldots}{[\![E]\!] \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2}$$

Nous devons donc en établir les prémisses, à savoir :

$$[\![E]\!] \oplus (x : \tau_1) \vdash_n a_1 : \tau_2 \tag{8.3}$$

$$\tau_1 \in \tau^n \tag{8.4}$$

À partir de (8.1), nous pouvons utiliser une règle (62) dans S_3 et déduire que :

$$E \oplus (x : (\tau_1, n)) \vdash_n a_1 : \tau_2 \tag{8.5}$$

$$\tau_1 \in \tau^n \tag{8.6}$$

Appliquons l'hypothèse de récurrence sur 8.5, il nous vient que :

$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_n a_1 : \tau_2 \tag{8.7}$$

Le schéma $[\![(\tau_1, n)]\!]$ de S_2 est égal à $\forall \vec{\alpha}. \tau_1$ où $\vec{\alpha} = \mathcal{V}^{\geq n+1}(\tau_1)$. Or puisque $\tau_1 \in \tau^n$, nous avons que $\mathcal{V}^{\geq n+1}(\tau_1) = \emptyset$. Donc $[\![(\tau_1, n)]\!] = \forall \emptyset. \tau_1$. Par conséquent, $[\![(x : (\tau_1, n))]\!] = (x : \tau_1)$. Appliqué à 8.7, nous en déduisons bien 8.3. Quant à 8.4, il provient directement de notre déduction 8.6.

Ainsi, munis de ces deux prémisses, nous pouvons effectivement reconstituer une dérivation 56 et nous obtenons donc le résultat attendu en 8.2.

Cas a est $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$.

Hypothèse :

$$E \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$[\![E]\!] \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 57 dans S_2 de la forme :

$$\frac{\ldots \quad \ldots}{[\![E]\!] \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Nous devons donc en établir les prémisses, à savoir :

$$[\![E]\!] \vdash_{n+1} a_1 : \tau_1 \tag{8.3}$$

$$\sigma = \forall \mathcal{V}^{\geq n+1}(\tau_1). \tau_1 \tag{8.4}$$

$$[\![E]\!] \oplus (x : \sigma) \vdash_n a_2 : \tau_2 \tag{8.5}$$

À partir de 8.1, nous pouvons utiliser une règle 63 dans S_3 et déduire que :

$$E \vdash_{n+1} a_1 : \tau_1 \tag{8.6}$$

$$E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2 \tag{8.7}$$

Appliquons l'hypothèse de récurrence sur 8.6 et 8.7, il nous vient :

$$[\![E]\!] \vdash_{n+1} a_1 : \tau_1 \tag{8.8}$$
$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_n a_2 : \tau_2 \tag{8.9}$$

La déduction 8.8 est exactement la prémisse attendue en 8.3. Par définition de $[\![\ ]\!]$, $[\![(x : (\tau_1, n))]\!] = \forall \vec{\alpha}. \tau_1$ avec $\vec{\alpha} = \mathcal{V}^{\geq n+1}(\tau_1)$, ce qui est bien égal à σ dans S_2. Donc $[\![E \oplus (x : (\tau_1, n))]\!] = [\![E]\!] \oplus (x : \sigma)$ et combiné avec 8.9 nous obtenons bien notre prémisse 8.5. Ainsi nous pouvons reconstituer une dérivation 57 comme attendu et conclure à 8.2.

Cas a est `let rec` $x = a_1$ `in` a_2.
Hypothèse :

$$E \vdash_n \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$[\![E]\!] \vdash_n \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 58 dans S_2 de la forme :

$$\frac{\ldots \quad \ldots}{[\![E]\!] \vdash_n \texttt{let rec}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Nous devons donc en établir les prémisses, à savoir :

$$[\![E]\!] \oplus (x : \sigma) \vdash_{n+1} a_1 : \tau_1 \tag{8.3}$$
$$\sigma = \forall \mathcal{V}^{\geq n+1}(\tau_1). \tau_1 \tag{8.4}$$
$$[\![E]\!] \oplus (x : \sigma) \vdash_n a_2 : \tau_2 \tag{8.5}$$

À partir de 8.1, nous pouvons utiliser une règle 64 dans S_3 et déduire que :

$$E \oplus (x : (\tau_1, n)) \vdash_{n+1} a_1 : \tau_1 \tag{8.6}$$
$$E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2 \tag{8.7}$$

Appliquons l'hypothèse de récurrence sur 8.6 et 8.7, il nous vient :

$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_{n+1} a_1 : \tau_1 \tag{8.8}$$
$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_n a_2 : \tau_2 \tag{8.9}$$

Par définition de $[\![\ ]\!]$, $[\![(x : (\tau_1, n))]\!] = \forall \vec{\alpha}. \tau_1$ avec $\vec{\alpha} = \mathcal{V}^{\geq n+1}(\tau_1)$, ce qui est bien égal à σ dans S_2. Donc $[\![E \oplus (x : (\tau_1, n))]\!] = [\![E]\!] \oplus (x : \sigma)$ et combiné avec 8.8 nous obtenons bien notre prémisse 8.3 et combiné avec 8.9 nous obtenons bien notre prémisse 8.5. Ainsi nous pouvons reconstituer une dérivation 58 comme attendu et conclure à 8.2.

□

S_1 implique S_2

Définition 8.7 (Relèvement) *Soit R, une application des variables sans niveau de S_1 dans les variables avec niveau de S_2 qui a toute variable de S_1 fait correspondre une variable de S_2 de niveau n au choix. On supposera qu'il existe une infinité de variables de S_2 de chaque niveau, ainsi R est toujours définie.*

Soit τ un type dans le système S_1. On définit l'opération de relèvement R qui transforme τ en un type avec niveau du système S_2 par :

$$\begin{array}{lcl} R(\alpha) & = & \alpha^n \\ R(\texttt{int}) & = & \texttt{int} \\ R(\tau_1 \to \tau_2) & = & R(\tau_1) \to R(\tau_2) \end{array}$$

Nous étendons ensuite simplement cette opération à un schéma de types $\sigma = \forall\alpha_i.\,\tau$ *de* S_1 *de la façon suivante :*

$$R(\sigma) = R(\forall\alpha_i.\,\tau) = \forall\alpha_i.\,R(\tau)$$

Pour terminer, soit un environnement de typage E *dans* S_1. *On définit le relèvement de* E *comme étant l'environnement dans* S_2 *résultant du relèvement de chaque liaison de* E.

Théorème 8.4 (S_1 implique S_2) *Soient un programme* a *et un environnement de typage* E *dans* S_1. *Si* $E \vdash a : \tau$ *dans* S_1 *et si toutes les variables libres de* $R(E)$ *sont de niveau inférieur ou égal à* n *(c'est-à-dire* $FV(E) \subseteq \tau^{\leq n}$*), alors il existe un renommage* ρ *des variables* α_i *appartenant à* $FV(\tau) \setminus FV(E)$ *en des variables* β_i *telles que* $R(\beta_i)$ *soient de niveau* n *exactement (ce que l'on appellera un renommage acceptable vis-à-vis de* (n, τ, E)*) et* $R(E) \vdash_n a : R(\rho(\tau))$ *dans* S_2.

Preuve : La preuve est effectuée par cas sur la forme des dérivations de typage.

Cas a est x.

Hypothèse :

$$E \vdash x : \tau \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$R(E) \vdash_n x : R(\rho(\tau)) \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 53 dans S_2 de la forme :

$$\frac{\cdots}{R(E) \vdash_n x : R(\rho(\tau)}$$

Nous devons donc en établir la prémisse, à savoir :

$$R(\rho(\tau)) \leq_n R(E(x)) \tag{8.3}$$

Prenons ρ, un renommage acceptable vis-à-vis de (τ, n, E) quelconque. Ceci est toujours possible car, par hypothèse, il existe une infinité de variables α telles que $R(\alpha)$ soit de niveau égal à n.

À partir de 8.1, nous pouvons appliquer le lemme de stabilité du typage par substitution avec la substitution ρ et nous obtenons :

$$\rho(E) \vdash x : \rho(\tau) \tag{8.4}$$

Puisque, par construction, ρ ne touche pas aux variables libres dans E, de 8.4, nous déduisons que :

$$E \vdash x : \rho(\tau) \tag{8.5}$$

À partir de 8.5, nous pouvons utiliser une règle 47 dans S_1 et déduire que :

$$\rho(\tau) \leq E(x) \tag{8.6}$$

c'est-à-dire, par définition de Gen dans S_1 dire que :

$$E(x) = \forall\alpha_i.\,\rho(\tau') \text{ et } \rho(\tau) = \rho(\tau')\{\alpha_i \leftarrow \tau_i''\}$$

Puisque par hypothèse $R(\rho(\tau)) \in \tau^{\leq n}$, nous en déduisons que $R((\rho(\tau'))\{\alpha_i \leftarrow \tau_i''\}) \in \tau^{\leq n}$, donc que :

$$R(\tau_i'') \in \tau^{\leq n} \tag{8.7}$$

$$R(\rho(\tau')) \in \tau^{\leq n} \tag{8.8}$$

De 8.8, nous avons alors que $\forall \alpha_i . R(\rho(\tau')) \in \tau^{\leq n}$, ce qui signifie bien, combiné avec 8.7, que dans S_2, $R(\rho(\tau)) \leq_n R(E(x))$ par définition de l'instanciation dans S_2. C'est le résultat attendu en 8.3.

Nous pouvons donc reconstituer une dérivation 53 comme attendu et clore ce cas de la notre preuve.

Cas a est i.
Hypothèse :

$$E \vdash i : \texttt{int} \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$R(E) \vdash_n i : R(\rho(\texttt{int})) \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 54 dans S_2 de la forme :

$$R(E) \vdash_n i : R(\rho(\texttt{int}))$$

Prenons le renommage ρ égal à la substitution identité. Celui-ci est trivialement acceptable vis-à-vis de $(\texttt{int}, n, E)$ puisque l'ensemble des variables libres dans `int` est vide.

De plus, par définition $R(\texttt{int}) = \texttt{int}$. Donc trivialement nous pouvons reconstituer une dérivation 54 dans S_2 à partir de 8.1, ce qui clos ce cas de notre preuve.

Cas a est $(a_1\ a_2)$.
Hypothèse :

$$E \vdash (a_1\ a_2) : \tau \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$R(E) \vdash_n (a_1\ a_2) : R(\rho(\tau)) \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 55 dans S_2 de la forme :

$$\frac{\ldots \quad \ldots}{R(E) \vdash_n (a_1\ a_2) : R(\rho(\tau))}$$

Nous devons donc en établir les prémisses, à savoir :

$$R(E) \vdash_n a_1 : R(\rho(\tau_1 \rightarrow \tau)) \tag{8.3}$$
$$R(E) \vdash_n a_2 : R(\rho(\tau_1)) \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 49 dans S_1 et déduire que :

$$E \vdash a_1 : \tau_1 \rightarrow \tau \tag{8.5}$$
$$E \vdash a_2 : \tau_1 \tag{8.6}$$

Appliquons l'hypothèse de récurrence sur 8.5. Nous en déduisons alors que :

$$\text{il existe un renommage } \rho_1 \text{ acceptable vis-à-vis de } (\tau_1 \rightarrow \tau, n, E) \tag{8.7}$$
$$R(E) \vdash_n a_1 : R(\rho_1(\tau_1 \rightarrow \tau)) \tag{8.8}$$

Par application du lemme de stabilité du typage par substitution à 8.6, nous déduisons que :

$$\rho_1(E) \vdash a_2 : \rho_1(\tau_1) \tag{8.9}$$

Puisque par construction, ρ_1 ne touche pas aux variables libres dans E, de 8.9, nous déduisons que :

$$E \vdash a_2 : \rho_1(\tau_1) \tag{8.10}$$

Appliquons maintenant l'hypothèse de récurrence sur 8.10. Il nous vient alors que :

$$\text{il existe un renommage } \rho_2 \text{ acceptable vis-à-vis de } (\rho_1(\tau_1), n, E) \tag{8.11}$$
$$R(E) \vdash_n a_2 : R(\rho_2 \circ \rho_1(\tau_1)) \tag{8.12}$$

Appliquons le lemme de stabilité du typage par substitution à 8.8 avec ρ_2, il nous vient que :

$$\rho_2(R(E)) \vdash_n a_1 : R(\rho_2 \circ \rho_1(\tau_1 \to \tau)) \tag{8.13}$$

Puisque par construction, ρ_2 ne touche pas aux variables libres dans E, donc dans $R(E)$, de 8.13, nous déduisons que :

$$R(E) \vdash_n a_1 : R(\rho_2 \circ \rho_1(\tau_1 \to \tau)) \tag{8.14}$$

Posons alors $\rho = \rho_2 \circ \rho_1$. De 8.14 et 8.12 nous déduisons donc que :

$$R(E) \vdash_n a_1 : R(\rho(\tau_1 \to \tau)) \tag{8.15}$$
$$R(E) \vdash_n a_2 : R(\rho(\tau_1)) \tag{8.16}$$

ce qui est bien nos deux prémisses attendues en 8.3 et 8.4. Nous pouvons donc reconstituer une dérivation 55 dans S_2 comme attendu et clore ce cas de la preuve.

Cas a est $\lambda x.a_1$.
Hypothèse :

$$E \vdash \lambda x.a_1 : \tau_1 \to \tau_2 \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$R(E) \vdash_n \lambda x.a_1 : R(\rho(\tau_1 \to \tau_2)) \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 56 dans S_2 de la forme :

$$\frac{\ldots \quad \ldots}{R(E) \vdash_n \lambda x.a_1 : R(\rho(\tau_1 \to \tau_2))}$$

Nous devons donc en établir les prémisses, à savoir :

$$R(E \oplus (x : \rho(\tau_1))) \vdash_n a_1 : \rho(\tau_2) \tag{8.3}$$
$$\rho(\tau_1) \in \tau^n \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 50 dans S_1 et déduire que :

$$E \oplus (x : \tau_1) \vdash a_1 : \tau_2 \tag{8.5}$$

Soit ρ_1 un renommage acceptable vis-à-vis de (τ_1, n, E) (c'est-à-dire tel qu'il envoie les variables de $FV(\tau_1) \setminus FV(E)$ vers des variables de niveau n).

Par stabilité du typage par substitution appliqué à 8.5 avec la substitution ρ_1, nous déduisons que :

$$\rho_1(E) \oplus (x : \rho_1(\tau_1)) \vdash a_1 : \rho_1(\tau_2) \tag{8.6}$$

Comme par construction, ρ_1 ne touche pas aux variables libres dans E, de 8.6, nous déduisons alors que :

$$E \oplus (x : \rho_1(\tau_1)) \vdash a_1 : \rho_1(\tau_2) \tag{8.7}$$

Appliquons alors l'hypothèse de récurrence sur 8.7, et il nous vient que :

$$\text{il existe un renommage } \rho_2 \text{ acceptable vis-à-vis de } (\rho_1(\tau_2), n, E \oplus (x : \rho_1(\tau_1))) \tag{8.8}$$
$$R(E \oplus (x : \rho_1(\tau_1))) \vdash_n a_1 : \rho_2 \circ \rho_1(\tau_2) \tag{8.9}$$

Or nous pouvons remarquer que $\rho_2 \circ \rho_1(\tau_1) = \rho_1(\tau_1)$ puisque par construction ρ_2 ne touche pas aux variables libres dans $E \oplus (x : \rho_1(\tau_1))$, donc en particulier dans τ_1. Ainsi, 8.9 est équivalent à :

$$R(E \oplus (x : \rho_1(\tau_1))) \vdash_n a_1 : \rho_1(\tau_2) \tag{8.10}$$

Prenons alors $\rho = \rho_2 \circ \rho_1$ et de 8.10, nous obtenons bien le résultat attendu en 8.3. De plus, par construction de ρ, nous avons que $\rho(\tau_1)$ est un type de niveau n comme attendu en 8.4.

Ainsi, munis de ces deux prémisses, nous pouvons reconstituer une dérivation 56 et clore ce cas de la preuve.

Cas a est $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$.
Hypothèse :

$$E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2 \tag{8.1}$$

Nous voulons prouver que dans S_2 :

$$R(E) \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : R(\rho_2(\tau_2)) \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 57 dans S_2 de la forme :

$$\frac{\cdots \quad \cdots}{R(E) \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : R(\rho_2(\tau_2))}$$

Nous devons donc en établir les prémisses, à savoir :

$$R(E) \vdash_{n+1} a_1 : R(\rho_1(\tau_1)) \tag{8.3}$$
$$\sigma' = \forall\, \mathcal{V}^{\geq n+1}(R(\rho_1(\tau_1))).\, R(\rho_1(\tau_1)) \tag{8.4}$$
$$R(E) \oplus (x : \sigma') \vdash_n a_2 : R(\rho_2(\tau_2)) \tag{8.5}$$

À partir de 8.1, nous pouvons utiliser une règle 51 dans S_1 et déduire que :

$$E \vdash a_1 : \tau_1 \tag{8.6}$$
$$\sigma = \mathrm{Gen}\,(\tau_1, E) \tag{8.7}$$
$$E \oplus (x : \sigma) \vdash a_2 : \tau_2 \tag{8.8}$$

Par application de l'hypothèse de récurrence sur 8.6, nous déduisons que :

$$\text{il existe un renommage } \rho_1 \text{ acceptable vis-à-vis de } (\tau_1, n+1, E) \tag{8.9}$$
$$R(E) \vdash_{n+1} a_1 : R(\rho_1(\tau_1)) \tag{8.10}$$

comme attendu en 8.3.

Le prémisse 8.4 n'est pas à proprement parler à établir : il s'agit d'une définition. Par contre, nous allons maintenant, pour pouvoir ensuite prouver 8.5, montrer que $R(\sigma) = \sigma'$, c'est-à-dire que :

$$R(\sigma) = \forall\, \mathcal{V}^{\geq n+1}(R(\rho_1(\tau_1))).\, R(\rho_1(\tau_1)) \tag{8.11}$$

De 8.7, nous déduisons par définition de Gen dans S_1, que $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau_1$ avec $\alpha_i \in FV(\tau_1) \backslash FV(E)$. Puisque ρ_1 est une renommage acceptable vis-à-vis de $(\tau_1, E, , n+1)$, nous avons que :

$$\sigma = \forall \beta_1 \ldots \beta_n . \rho_1(\tau_1) \tag{8.12}$$

$$\text{avec } \beta_i = \rho_1(\alpha_i) \tag{8.13}$$

Ainsi, $R(\sigma) = R(\forall \beta_1 \ldots \beta_n . \rho_1(\tau_1)) = \forall R(\beta_1) \ldots R(\beta_n) . R(\rho_1(\tau_1))$. Par construction, puisque ρ_1 est acceptable vis-à-vis de $(\tau_1, n+1, E)$, l'ensemble de variables quantifiées $\{R(\beta_1) \ldots R(\beta_n)\}$ est bien égal à l'ensemble $\mathcal{V}^{\geq n+1}(R(\rho_1(\tau_1)))$. Ainsi, nous établissons bien le résultat attendu en 8.11.

Appliquons maintenant l'hypothèse de récurrence sur 8.8 (grâce à 8.11) et il nous vient directement que $R(E) \oplus (x : \sigma) \vdash_n a_2 : R(\rho_2(\tau_2))$ comme nous l'attendions en 8.5.

Ainsi, munis de ces trois prémisses, nous pouvons effectivement reconstituer une dérivation 57 dans S_2 et clore ce cas de la preuve.

Cas a est `let rec` $x = a_1$ `in` a_2.

On se convaincra aisément que ce cas est traité identiquement au cas précédent et nous ne le décrirons pas afin de ne pas rallonger abusivement cette démonstration. □

S_2 implique S_3

Théorème 8.5 (S_2 implique S_3) *Soient un programme a et un environnement de typage E. Si $[\![E]\!] \vdash_n a : \tau$ dans S_2, alors $E \vdash_n a : \tau$ dans S_3. Autrement dit, un programme clos typable dans S_2 est aussi typable dans S_3.*

Preuve : La preuve est effectuée par cas sur la forme des dérivations de typage.

Cas a est x.

Hypothèse :

$$[\![E]\!] \vdash_n x : \tau' \tag{8.1}$$

Nous voulons prouver que dans S_3 :

$$E \vdash_n x : \tau' \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 59 dans S_3 de la forme :

$$\frac{\ldots \quad \ldots}{E \vdash_n x : \tau'}$$

Nous devons donc en établir les prémisses, à savoir :

$$E(x) = (\tau, m) \tag{8.3}$$

$$\tau \leq_n^m \tau' \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 53 dans S_2 et déduire que :

$$\tau' \leq [\![E]\!](x) \tag{8.5}$$

Par définition de l'instanciation dans S_2, nous déduisons de 8.5 que :

$$[\![E(x)]\!] \text{ est de la forme } \forall \alpha_i . \tau \tag{8.6}$$

$$\text{il existe } \tau_i \in \tau^{\leq n} \text{ tels que } \tau' = \tau\{\alpha_i \leftarrow \tau_i\} \tag{8.7}$$

Puisque E est un environnement dans S_3, et que par 8.5 $x \in Dom[\![E]\!]$, alors

$$E(x) \text{ est de la forme } (\tau, m) \text{ pour un certain } m \tag{8.8}$$

ce que nous attendions en 8.3.

Par définition de la traduction, puisque nous avons 8.6 et 8.8, c'est que les variables α_i sont des variables de niveau $\geq m+1$. Ceci combiné avec 8.7, nous permet de déduire, par définition de $\overset{?}{\leq}{}^{m}_{n}$, que nous avons bien $\tau \leq^m_n \tau'$ comme attendu en 8.4.

Ainsi munis de ces deux prémisses, nous pouvons alors reconstituer une dérivation 59 dans S_3 et conclure ce cas de la preuve.

Cas a est i.

Hypothèse :

$$[\![E]\!] \vdash_n i : \texttt{int} \tag{8.1}$$

Nous voulons prouver que dans S_3 :

$$E \vdash_n i : \texttt{int} \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle (60) dans S_3 de la forme :

$$E \vdash_n i : \texttt{int}$$

N'ayant aucune prémisse à établir et munis de l'hypothèse 8.1, nous déduisons trivialement notre but et closons ce cas de la preuve.

Cas a est $(a_1\ a_2)$.

Hypothèse :

$$[\![E]\!] \vdash_n (a_1\ a_2) : \tau \tag{8.1}$$

Nous voulons prouver que dans S_3 :

$$E \vdash_n (a_1\ a_2) : \tau \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 61 dans S_3 de la forme :

$$\frac{\dots \quad \dots}{E \vdash_n (a_1\ a_2) : \tau}$$

Nous devons donc en établir les prémisses, à savoir :

$$E \vdash_n a_1 : \tau_1 \to \tau \tag{8.3}$$

$$E \vdash_n a_2 : \tau_1 \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 55 dans S_2 et déduire que :

$$[\![E]\!] \vdash_n a_1 : \tau_1 \to \tau \tag{8.5}$$

$$[\![E]\!] \vdash_n a_2 : \tau_1 \tag{8.6}$$

Appliquons l'hypothèse de récurrence une fois sur 8.5 et une fois sur 8.6, et nous déduisons directement nos buts 8.3 et 8.4.

Nous pouvons ainsi reconstituer une dérivation 61 comme attendu pour conclure à 8.1 et clore ce cas de la preuve.

Cas a est $\lambda x.a_1$.

Hypothèse :

$$[\![E]\!] \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2 \tag{8.1}$$

Nous voulons prouver que dans S_3 :

$$E \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2 \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 62 dans S_3 de la forme :

$$\frac{\ldots \quad \ldots}{E \vdash_n \lambda x.a_1 : \tau_1 \to \tau_2}$$

Nous devons donc en établir les prémisses, à savoir :

$$E \oplus (x : (\tau_1, n)) \vdash_n a_1 : \tau_2 \tag{8.3}$$
$$\tau_1 \in \tau^n \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 56 dans S_2 et déduire que :

$$[\![E]\!] \oplus (x : \tau_1) \vdash_n a_1 : \tau_2 \tag{8.5}$$
$$\tau_1 \in \tau^n \tag{8.6}$$

La déduction 8.6 nous permet de satisfaire directement le but 8.4.

Par 8.6, nous savons que τ_1 est de niveau n au plus. Donc, par définition de la traduction, $[\![(\tau_1, n)]\!] = \forall \emptyset. \tau_1$. Donc, $(x : \tau_1) = [\![(x : (\tau_1, n))]\!]$. Combiné avec 8.5, nous déduisons donc que nous avons :

$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_n a_1 : \tau_2 \tag{8.7}$$

Appliquons l'hypothèse de récurrence sur 8.7, il vient que :

$$E \oplus (x : (\tau_1, n)) \vdash_n a_1 : \tau_2 \tag{8.8}$$

comme attendu en 8.3. Munis de ces deux prémisses, nous pouvons alors reconstituer une dérivation 62 dans S_3 et conclure ce cas de la preuve.

Cas a est `let` $x = a_1$ `in` a_2.

Hypothèse :

$$[\![E]\!] \vdash_n \texttt{let } x = a_1 \texttt{ in } a_2 : \tau \tag{8.1}$$

Nous voulons prouver que dans S_3 :

$$E \vdash_n \texttt{let } x = a_1 \texttt{ in } a_2 : \tau \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 63 dans S_3 de la forme :

$$\frac{\ldots \quad \ldots}{E \vdash_n \texttt{let } x = a_1 \texttt{ in } a_2 : \tau}$$

Nous devons donc en établir les prémisses, à savoir :

$$E \vdash_{n+1} a_1 : \tau_1 \tag{8.3}$$
$$E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2 \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 57 dans S_2 et déduire que :

$$[\![E]\!] \vdash_{n+1} a_1 : \tau_1 \tag{8.5}$$
$$\sigma = \forall \mathcal{V}^{\geq n+1}(\tau_1). \tau_1 \tag{8.6}$$
$$[\![E]\!] \oplus (x : \sigma) \vdash_n a_2 : \tau_2 \tag{8.7}$$

Appliquons l'hypothèse de récurrence sur 8.5, il vient que :

$$E \vdash_{n+1} a_1 : \tau_1 \tag{8.8}$$

ce qui était attendu en but 8.3. Par définition de la traduction, nous savons que $\mathcal{V}^{\geq n+1}(\tau_1).\tau_1 = [\![(\tau_1, n)]\!]$, donc $\sigma = [\![(\tau_1, n)]\!]$. Combinons ceci avec 8.7, nous obtenons que :

$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_n a_2 : \tau_2 \tag{8.9}$$

Nous appliquons alors l'hypothèse de récurrence sur 8.9 et nous déduisons que :

$$E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2 \tag{8.10}$$

comme attendu en 8.4. Ainsi, munis de ces deux prémisses, nous pouvons alors reconstituer une dérivation 63 dans S_3 et conclure ce cas de la preuve.

Cas a est `let rec` $x = a_1$ `in` a_2.
Hypothèse :

$$[\![E]\!] \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \tau \tag{8.1}$$

Nous voulons prouver que dans S_3 :

$$E \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \tau \tag{8.2}$$

Pour y parvenir, nous voulons utiliser une règle 64 dans S_3 de la forme :

$$\frac{\dots \qquad \dots}{E \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \tau}$$

Nous devons donc en établir les prémisses, à savoir :

$$E \oplus (x : (\tau_1, n)) \vdash_{n+1} a_1 : \tau_1 \tag{8.3}$$
$$E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2 \tag{8.4}$$

À partir de 8.1, nous pouvons utiliser une règle 58 dans S_2 et déduire que :

$$[\![E]\!] \oplus (x : \sigma) \vdash_{n+1} a_1 : \tau_1 \tag{8.5}$$
$$\sigma = \forall\, \mathcal{V}^{\geq n+1}(\tau_1).\, \tau_1 \tag{8.6}$$
$$[\![E]\!] \oplus (x : \sigma) \vdash_n a_2 : \tau_2 \tag{8.7}$$

Par définition de la traduction, nous savons que $\mathcal{V}^{\geq n+1}(\tau_1).\tau_1 = [\![(\tau_1, n)]\!]$, donc $\sigma = [\![(\tau_1, n)]\!]$. Combinons ceci avec 8.5 puis avec 8.7, nous obtenons que :

$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_{n+1} a_1 : \tau_1 \tag{8.8}$$
$$[\![E \oplus (x : (\tau_1, n))]\!] \vdash_n a_2 : \tau_2 \tag{8.9}$$

Nous appliquons alors l'hypothèse de récurrence sur 8.8 puis sur 8.9 et nous déduisons que :

$$E \oplus (x : (\tau_1, n)) \vdash_{n+1} a_1 : \tau_1 \tag{8.10}$$
$$E \oplus (x : (\tau_1, n)) \vdash_n a_2 : \tau_2 \tag{8.11}$$

comme attendu en 8.3 et en 8.4. Ainsi, munis de ces deux prémisses, nous pouvons alors reconstituer une dérivation 64 dans S_3 et conclure ce cas de la preuve.

□

Correction de S_2 et de S_3

La correction de S_3 est un corollaire de celle de S_2, cette dernière étant elle-même un corollaire de celle de S_1. En effet, considérons comme acquise la correction du système S_1. La propriété de correction énonce que si un programme a est typable alors sa réduction n'est pas bloquée.

Nous avons montré que tout programme typable dans S_2 est typable dans S_1. De plus tout programme typable dans S_1 n'est pas bloqué. Les règles de réduction étant les mêmes dans S_1 et dans S_2, il en découle que le programme ne sera pas bloqué dans S_2.

Pour établir la correction de S_3, il suffit d'appliquer le même raisonnement en se basant sur la correction de S_2 et l'équivalence entre S_3 et S_2.

8.4 Un algorithme d'inférence avec récursion polymorphe

Nous avons étudié jusqu'à présent un *système de types* étendant celui de ML par la récursion polymorphe. Comme ce fut le cas pour notre analyse d'exceptions, ce système décrit la procédure de *vérification* d'un programme étiqueté par des types. Un compilateur ML se doit, non pas de vérifier des annotations de types sur des programmes, mais de synthétiser ces types, sans que le programmeur n'ait à les expliciter dans son programme : c'est ce que nous avons nommé *l'inférence de types*.

Nous allons donc maintenant étudier l'inférence de types associée au système que nous venons de décrire. Celle-ci est donc décrite par un algorithme prenant en entrée un programme ML et générant en sortie un programme dont chaque expression et sous-expression est annotée par son type.

Dans un premier temps, nous allons étudier l'inférence hors de tout critère d'arrêt. Nous décrirons ainsi un *semi-algorithme*. Puis nous ajouterons à celui-ci un test d'arrêt nous garantissant la terminaison et verrons de quelle façon il est possible de retrouver l'inférence de types à la ML sans `let rec` polymorphe dans le cas où ce test provoque l'échec de la recherche de la solution.

8.4.1 Présentation globale de l'algorithme

Notre algorithme d'inférence est présenté en deux parties indépendantes : la génération d'un ensemble de contraintes d'unification et de semi-unification à partir du programme à typer, puis la résolution de cet ensemble de contraintes.

Dans une implémentation réelle, il est possible de fusionner partiellement ces deux phases. En effet, les implantations réelles et efficaces de typeurs pour ML n'utilisent pas des techniques de compositions de substitutions pour effectuer l'unification entre termes de types, mais un mécanisme d'unification en place (dite *destructrice*) dans lequel deux termes unifiés sont physiquement modifiés pour devenir égaux. Ceci permet d'obtenir des performances compatibles en terme de temps de calcul avec l'utilisation de l'inférence de types sur des programmes de plusieurs milliers de lignes.

Nous montrerons que l'ordre de résolution des contraintes d'unification et de semi-unification n'a aucune importance du point de vue de la correction de l'algorithme. Ceci garantira donc que la résolution « au vol » de contraintes d'unification (comme cela se produit lorsque l'on effectue de l'unification destructrice) dans notre ensemble de contraintes ne pose pas de problème particulier. Notre algorithme d'inférence peut donc être modélisé de la façon suivante :

1. Créer un ensemble de contraintes d'unification et de semi-unification à partir du programme (algorithme $\mathcal{C}$) ;
2. Résoudre cet ensemble (algorithme SU) ;
3. En cas d'échec de la résolution dû au test d'arrêt, se ramener à une forme de typage sans récursion polymorphe.

Segmenter l'algorithme suivant ces deux phases nous permet d'ores et déjà d'établir les propriétés devant relier chacune d'entre elles, propriétés que démontrerons ultérieurement afin d'assurer la cohérence de notre algorithme :

$$\text{Programme } a \quad \underset{\xrightarrow{2}}{\overset{\xleftarrow{1}}{\mathcal{C}}} \quad \text{Système de contraintes } C \quad \underset{\xrightarrow{4}}{\overset{\xleftarrow{3}}{SU}} \quad \text{Solution du système}$$

1. « Si le système C a une solution, alors le programme a est bien typé » **(correction)** : Si $\mathcal{C}(E, a, n, V) = (\tau, C, V')$ et $\exists\, \varphi$ solution de C, alors $\varphi(E) \vdash_n a : \varphi(\tau)$.
2. « Si le programme a est bien typé, alors le système C a une solution » **(complétude)** : Si $\mathcal{C}(E, a, n, V) = (\tau, C, V')$ et $FV(E) \subseteq V$ et $\exists\, \varphi_0, \tau_0$ tels que $Dom(\varphi_0) \subseteq V$ et $\varphi_0(E) \vdash_n a : \tau_0$, alors $\exists\, \varphi$ avec $\varphi_{|V} = \varphi_0$ telle que φ soit solution de C et $\tau_0 = \varphi(\tau)$ et $Dom(\varphi) \subseteq V'$.
3. « Si l'algorithme SU réussit, alors il renvoie une solution principale de C ».
4. *« ~~S'il existe une solution de C, alors l'algorithme SU réussit~~ ».*

Il est clair que nous n'avons aucune chance de prouver le dernier point, dû à la présence du test d'arrêt devant garantir la terminaison de notre algorithme. Rappelons que ce critère arbitraire a pour but de remédier à l'indécidabilité de l'inférence en restreignant la recherche de solutions dans les cas où celle-ci risquerait de ne jamais aboutir.

8.4.2 La génération de l'ensemble de contraintes

Définition 8.8 (Ensemble de contraintes) *On appellera « ensemble de contraintes », un ensemble de la forme :*

$$C = \{\tau_1 \overset{m_1}{\underset{n_1}{\stackrel{?}{\leq}}} \tau'_1; \ldots; \tau_k \overset{m_k}{\underset{n_k}{\stackrel{?}{\leq}}} \tau'_k; \tau_{k+1} \underset{n_{k+1}}{\stackrel{?}{=}} \tau'_{k+1}; \ldots; \tau_l \underset{n_l}{\stackrel{?}{=}} \tau'_l\}$$

où $\stackrel{?}{\leq}{}^m_n$ *est une contrainte de* semi-unification *et* $\stackrel{?}{=}_n$ *une contrainte* d'unification.

Notons que l'on rajoute explicitement, sur les contraintes d'unification, un niveau n représentant le niveau auquel doivent être créées les variables intermédiaires dont pourrait avoir besoin l'algorithme d'unification. Dans le cadre de l'unification utilisée par ML, il n'est jamais besoin de générer de telles variables intermédiaires et cette information peut être oubliée. Néanmoins dans le cas d'autres algèbres de types (avec rangées par exemple, comme dans notre analyse d'exceptions ou dans [44, 45, 61, 43]), ceci peut se produire.

Nous pourrons alors écrire C sous la forme $C = E \cup I$, où E est le sous-ensemble ne contenant que les contraintes d'unification et I le sous-ensemble ne contenant que les contraintes de semi-unification.

Définition 8.9 (Substitution respectant les niveaux) *Une substitution φ de la forme $\{\alpha_i^{n_i} \leftarrow \tau_i\}$ préserve les niveaux si pour tout i, on a : $\tau_i \in \tau^{\leq n_i}$.*

Autrement dit, une substitution respecte les niveaux si elle envoie des variables de types de niveau n sur des types de niveau n au plus.

Définition 8.10 (Solution d'un ensemble de contraintes) *On dit qu'une substitution φ est solution d'un ensemble de contraintes C si :*

- *φ respecte les niveaux,*
- *et pour toute contrainte $\tau_1 \stackrel{?}{=}_n \tau_2$ de C l'on a $\varphi(\tau_1) =_n \varphi(\tau_2)$. La substitution φ est alors un* unificateur *de τ_1 et τ_2.*
- *et pour toute contrainte $\tau_1 \stackrel{?}{\leq}{}^m_n \tau_2$ de C l'on a $\varphi(\tau_1) \leq^m_n \varphi(\tau_2)$. La substitution φ est alors un* semi-unificateur *de τ_1 et τ_2.*

Nous décrivons maintenant l'algorithme $\mathcal{C}$ qui produit l'ensemble des contraintes associé à une expression. Comme dans les algorithmes formalisés précédemment, nous rendons explicite l'ensemble des variables qui ne peuvent pas être utilisées comme variables fraîches durant l'exécution de l'algorithme $\mathcal{C}$.

Le résultat de l'algorithme est un triplet $(\tau, C, V') = \mathcal{C}(E, a, n, V)$ où E est un environnement de typage, a une expression, n le niveau courant de typage, V l'ensemble des variables non utilisables comme variables fraîches, τ le type trouvé pour l'expression, C l'ensemble de contraintes généré et V' le nouvel ensemble de variables non utilisables comme variables fraîches. L'algorithme est défini par induction de la façon suivante :

• **Si** a **est** x **:**
soit $(\tau, m) = E(x)$
soit $\alpha \notin V$ **une variable fraîche de niveau** n
retourner $(\alpha, \{\tau \stackrel{?}{\leq}{}^m_n \alpha\}, V \cup \{\alpha\})$.

• **Si** a **est** i **:**
retourner $(\texttt{int}, \{\}, V)$.

• **Si** a **est** $(a_1\ a_2)$ **:**
soit $(\tau_1, C_1, V_1) = \mathcal{C}(E, a_1, n, V)$
soit $(\tau_2, C_2, V_2) = \mathcal{C}(E, a_2, n, V_1)$
soit $\alpha \notin V_2$ **une variable fraîche de niveau** n
retourner $(\alpha, C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\}, V_2 \cup \{\alpha\})$.

• **Si** a **est** $\lambda x.a_1$ **:**
soit $\alpha \notin V$ **une variable fraîche de niveau** n
soit $(\tau, C, V_1) = \mathcal{C}(E \oplus (x : (\alpha, n)), a_1, n, V \cup \{\alpha\})$
retourner $(\alpha \to \tau, C, V_1)$.

• **Si** a **est** $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$ **:**
soit $(\tau_1, C_1, V_1) = \mathcal{C}(E, a_1, n+1, V)$
soit $(\tau_2, C_2, V_2) = \mathcal{C}(E \oplus (x : (\tau_1, n), a_2, n, V_1)$
retourner $(\tau_2, C_1 \cup C_2, V_2)$.

• **Si** a **est** $\texttt{let rec}\ x = a_1\ \texttt{in}\ a_2$ **:**
soit $\alpha \notin V$ **une variable fraîche de niveau** $n+1$
soit $(\tau_1, C_1, V_1) = \mathcal{C}(E \oplus (x : (\alpha, n), a_1, n+1, V \cup \{\alpha\})$
soit $(\tau_2, C_2, V_2) = \mathcal{C}(E \oplus (x : (\tau_1, n), a_2, n, V_1)$
retourner $(\tau_2, C_1 \cup C_2 \cup \{\alpha \stackrel{?}{=}_{n+1} \tau_1\}, V_2)$.

8.4.3 Preuves de l'algorithme de génération de contraintes

Il est maintenant nécessaire de s'assurer que cet algorithme de génération d'un ensemble de contraintes est correct et complet vis-à-vis des règles de typage du système S_3.

Établir sa *correction* revient alors à montrer que si un ensemble de contraintes généré à partir d'un programme comporte une solution, alors ce programme est bien typé dans notre système de types.

Dualement, la *complétude* de cet algorithme montrera que si un programme est bien typé dans notre système de types, alors l'ensemble de contraintes généré par notre algorithme comporte effectivement une solution.

Correction

Théorème 8.6 (L'algorithme $\mathcal{C}$ est correct) *Soient un environnement de typage E, une expression a, un niveau courant de typage n et un ensemble de variables V. S'il existe une substitution φ préservant les niveaux, solution de C et si $(\tau, C, V') = \mathcal{C}(E, a, n, V)$ est défini, alors $\varphi(E) \vdash_n a : \varphi(\tau)$ dans le système S_3.*

De plus l'algorithme garantit les invariants suivants : $V \subseteq V'$ et si $FV(E) \subseteq V$ alors $FV(\tau) \subseteq V'$ et $FV(C) \subseteq V'$.

Preuve : La démonstration de ce théorème est effectuée par cas sur la forme des expressions.

Cas a est x.
Hypothèses :

$$(\tau, m) = E(x) \tag{8.1}$$
$$\alpha \notin V \text{ est une variable fraîche de niveau } n \tag{8.2}$$
$$\varphi \text{ est solution de } \{\tau \overset{?}{\leq}{}^m_n \alpha\} \tag{8.3}$$

Nous voulons prouver que :

$$\varphi(E) \vdash_n x : \varphi(\alpha) \tag{8.4}$$

Pour y parvenir nous voulons utiliser une règle 59 de la forme :

$$\frac{\dots \quad \dots}{\varphi(E) \vdash_n x : \varphi(\alpha)}$$

dont nous devons établir les prémisses, à savoir :

$$\varphi(E(x)) = (\varphi(\tau), m) \tag{8.5}$$
$$\varphi(\tau) \leq^m_n \varphi(\alpha) \tag{8.6}$$

Nous obtenons 8.5 trivialement. Quant à 8.6, elle découle directement de la définition de $\leq^m_n$ et du fait que φ soit solution de $\{\tau \overset{?}{\leq}{}^m_n \alpha\}$ (hypothèse 8.3). Ainsi munis de nos deux prémisses nous pouvons reconstituer une dérivation 59 et conclure 8.4.

Prouvons maintenant les invariants de l'algorithme :

- $V \subseteq V \cup \{\alpha\}$: trivial
- $FV(\alpha) \subseteq V \cup \{\alpha\}$: trivial
- $FV(\{\tau \overset{?}{\leq}{}^m_n \alpha\}) \subseteq V \cup \{\alpha\}$: trivial car $\tau \in \mathrm{Dom}\,(E)$ et par hypothèse des invariants, $FV(E) \subseteq V$.

Cas a est i.
Nous voulons prouver que :

$$\varphi(E) \vdash_n i : \varphi(\mathtt{int}) \tag{8.7}$$

Ce cas est immédiatement prouvé, la substitution φ n'ayant aucun effet sur le type du résultat, et l'environnement E n'étant pas utile pour le typage de l'expression. De même, les invariants de l'algorithme sont trivialement vérifiés.

Cas a est $(a_1\ a_2)$.
Hypothèses :

$$(C_1, \tau_1, V_1) = \mathcal{C}(E, a_1, n, V) \tag{8.1}$$
$$(C_2, \tau_2, V_2) = \mathcal{C}(E, a_2, n, V_1) \tag{8.2}$$
$$\varphi \text{ est solution de } C_1 \cup C_2 \cup \{\tau_1 \overset{?}{=}_n \tau_2 \to \alpha\} \tag{8.3}$$

Nous voulons prouver que :

$$\varphi(E) \vdash_n (a_1\ a_2) : \varphi(\alpha) \tag{8.4}$$

Pour y parvenir nous voulons utiliser une règle 61 de la forme :

$$\frac{\dots \quad \dots}{\varphi(E) \vdash_n (a_1\ a_2) : \varphi(\alpha)}$$

dont nous devons donc montrer ses deux prémisses, à savoir :

$$\varphi(E) \vdash_n a_1 : \varphi(\tau_2 \to \alpha) \tag{8.5}$$
$$\varphi(E) \vdash_n a_2 : \varphi(\tau_2) \tag{8.6}$$

Puisque par 8.3, φ est solution de $C_1 \cup C_2 \cup \{\tau_1 \stackrel{?}{=}_n \tau_2 \to \alpha\}$ nous déduisons que :

$$\varphi \text{ est solution de } C_1 \tag{8.7}$$
$$\varphi \text{ est solution de } C_2 \tag{8.8}$$
$$\varphi \text{ est solution de } \{\tau_1 \stackrel{?}{=}_n \tau_2 \to \alpha\} \tag{8.9}$$

Nous pouvons donc appliquer l'hypothèse de récurrence sur 8.1 grâce à 8.7 et au fait que $FV(E) \subseteq V$, puis sur 8.2 grâce à 8.8 et au fait que $FV(E) \subseteq V_1$. Nous déduisons alors que :

$$\varphi(E) \vdash_n : a_1 : \varphi(\tau_1) \tag{8.10}$$
$$\varphi(E) \vdash_n : a_2 : \varphi(\tau_2) \tag{8.11}$$
$$V \subseteq V_1 \quad FV(\tau_1) \subseteq V_1 \quad FV(C_1) \subseteq V_1 \tag{8.12}$$
$$V_1 \subseteq V_2 \quad FV(\tau_2) \subseteq V_2 \quad FV(C_2) \subseteq V_2 \tag{8.13}$$

Par 8.9, nous savons que :

$$\varphi(\tau_1) = \varphi(\tau_2 \to \alpha) \tag{8.14}$$

Ainsi, en combinant 8.10 et 8.14, nous obtenons bien la prémisse attendue en 8.5. De plus, la déduction 8.11 est directement ce que nous attendions en 8.6. Ainsi, munis de ces deux prémisses, nous pouvons reconstituer la dérivation 61 attendue et conclure 8.4.

Prouvons maintenant les invariants de l'algorithme :
- $V \subseteq V_2 \cup \{\alpha\}$: par 8.12 et 8.13.
- $FV(\alpha) \subseteq V_2 \cup \{\alpha\}$: trivial
- $FV(C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\}) \subseteq V_2 \cup \{\alpha\}$: conséquence de 8.12 et 8.13.

Cas a est $\lambda x.a_1$.
Hypothèses :

$$(\tau, C, V_1) = \mathcal{C}(E \oplus (x : (\alpha, n)), a_1, n, V \cup \{\alpha\}) \tag{8.1}$$
$$\alpha \notin V \text{ est de niveau } n \tag{8.2}$$
$$\varphi \text{ est solution de } C \tag{8.3}$$

Nous voulons prouver que :

$$\varphi(E) \vdash_n \lambda x.a_1 : \varphi(\alpha \to \tau) \tag{8.4}$$

Pour y parvenir nous voulons utiliser une règle 62 de la forme :

$$\frac{\dots \quad \dots}{\varphi(E) \vdash_n \lambda x.a_1 : \varphi(\alpha \to \tau)}$$

dont nous devons établir les prémisses, à savoir :

$$\varphi(E) \oplus (x : (\varphi(\alpha), n)) \vdash_n a_1 : \varphi(\tau) \tag{8.5}$$
$$\varphi(\alpha) \in \tau^n \tag{8.6}$$

Puisque $FV(E \oplus (x : (\varphi(\alpha), n))) = FV(E) \oplus \{\alpha\} \subseteq V \cup \{\alpha\}$, nous pouvons appliquer l'hypothèse de récurrence sur 8.1 et déduire que :

$$\varphi(E \oplus (x : (\alpha, n))) \vdash_n a_1 : \varphi(\tau) \tag{8.7}$$
$$V \cup \{\alpha\} \subseteq V_1 \quad FV(\tau) \subseteq V_1 \quad FV(C) \subseteq V_1 \tag{8.8}$$

où 8.7 correspond bien (par mise en facteur de φ) à la prémisse attendue en 8.5.

La substitution φ préserve les niveaux (cf. définition 8.10). En particulier, par 8.2, α étant une variable de niveau n, φ va l'envoyer vers un type de niveau n au plus. Donc on aura bien $\varphi(\alpha) \in \tau^n$ comme attendu en 8.6.

Ainsi munis de ces deux prémisses, nous pouvons reconstituer une dérivation 62 comme attendu et effectivement en déduire le but 8.4.

Prouvons maintenant les invariants de l'algorithme :
- $V \subseteq V_1$: trivial par 8.8.
- $FV(\alpha \to \tau) \subseteq V_1$: car par 8.8, $\alpha \in V_1$ et $FV(\tau) \subseteq V_1$.
- $FV(C) \subseteq V_1$: trivial, directement obtenu par la déduction 8.8.

Cas a est $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$.

Hypothèses :

$$(\tau_1, C_1, V_1) = \mathcal{C}(E, a_1, n+1, V) \tag{8.1}$$
$$(\tau_2, C_2, V_2) = \mathcal{C}(E \oplus (x : (\tau_1, n), a_2, n, V_1) \tag{8.2}$$
$$\varphi \text{ est solution de } C_1 \cup C_2 \tag{8.3}$$

Nous voulons prouver que :

$$\varphi(E) \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2) \tag{8.4}$$

Pour y parvenir nous voulons utiliser une règle 63 de la forme :

$$\frac{\ldots \qquad \ldots}{\varphi(E) \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2)}$$

dont nous devons établir les prémisses, à savoir :

$$\varphi(E) \vdash_{n+1} a_1 : \varphi(\tau_1) \tag{8.5}$$
$$\varphi(E) \oplus (x : (\varphi(\tau_1), n)) \vdash_n a_2 : \varphi(\tau_2) \tag{8.6}$$

De l'hypothèse 8.3, nous déduisons que :

$$\varphi \text{ est solution de } C_1 \tag{8.7}$$
$$\varphi \text{ est solution de } C_2 \tag{8.8}$$

Grâce à 8.7, nous pouvons appliquer l'hypothèse de récurrence sur 8.1. Nous déduisons alors que :

$$\varphi(E) \vdash_{n+1} a_1 : \varphi(\tau_1) \tag{8.9}$$

comme attendu en 8.5. De plus nous en déduisons, par invariants de l'algorithme, que :

$$V \subseteq V_1 \quad FV(\tau_1) \subseteq V_1 \quad FV(C_1) \subseteq V_1 \tag{8.10}$$

Nous appliquons maintenant l'hypothèse de récurrence sur 8.2 ce qui nous mène directement au résultat attendu en 8.6. De plus nous en déduisons, par invariants de l'algorithme, que :

$$V_1 \subseteq V_2 \quad FV(\tau_2) \subseteq V_2 \quad FV(C_2) \subseteq V_2 \tag{8.11}$$

Ainsi, munis de nos deux prémisses, nous reconstituons une dérivation 63 et déduire 8.4.

Prouvons maintenant les invariants de l'algorithme :

- $V \subseteq V_2$: trivial par 8.10 puis 8.11.
- $FV(\tau_2) \subseteq V_2$: trivial, directement déduit de l'hypothèse de récurrence (8.11).
- $FV(C_1 \cup C_2) \subseteq V_2$: trivial puisque par 8.11 $FV(C_2) \subseteq V_2$, par 8.10 $FV(C_1) \subseteq V_1$. Or, à nouveau par 8.11 $V_1 \subseteq V_2$.

Cas a est `let rec` $x = a_1$ `in` a_2.

Hypothèses :

$$(\tau_1, C_1, V_1) = \mathcal{C}(E \oplus (x : (\alpha, n), a_1, n+1, V \cup \{\alpha\}) \tag{8.1}$$
$$(\tau_2, C_2, V_2) = \mathcal{C}(E \oplus (x : (\tau_1, n), a_2, n, V_1) \tag{8.2}$$
$$\alpha \notin V \text{ est une variable fraîche de niveau } n+1 \tag{8.3}$$
$$\varphi \text{ est solution de } C_1 \cup C_2 \cup \{\alpha \stackrel{?}{=}_{n+1} \tau_1\} \tag{8.4}$$

Nous voulons prouver que :

$$\varphi(E) \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \varphi(\tau_2) \tag{8.5}$$

Pour y parvenir nous voulons utiliser une règle 64 de la forme :

$$\frac{\ldots \quad \ldots}{\varphi(E) \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : \varphi(\tau_2)}$$

dont nous devons établir les prémisses, à savoir :

$$\varphi(E) \oplus (x : (\varphi(\tau_1), n)) \vdash_{n+1} a_1 : \varphi(\tau_1) \tag{8.6}$$
$$\varphi(E) \oplus (x : (\varphi(\tau_1), n)) \vdash_n a_2 : \varphi(\tau_2) \tag{8.7}$$

De l'hypothèse 8.4, nous déduisons que :

$$\varphi \text{ est solution de } C_1 \tag{8.8}$$
$$\varphi \text{ est solution de } C_2 \tag{8.9}$$
$$\varphi \text{ est solution de } \{\alpha \stackrel{?}{=}_{n+1} \tau_1\} \tag{8.10}$$

Appliquons l'hypothèse de récurrence sur 8.1, il vient alors que $\varphi(E \oplus (x : (\varphi(\alpha), n))) \vdash_{n+1} a_1 : \varphi(\tau_1)$ c'est-à-dire :

$$\varphi(E) \oplus (x : (\varphi(\alpha), n)) \vdash_{n+1} a_1 : \varphi(\tau_1) \tag{8.11}$$

et par invariants de l'algorithme, que :

$$V \cup \{\alpha\} \subseteq V_1 \quad FV(\tau_1) \subseteq V_1 \quad FV(C_1) \subseteq V_1 \tag{8.12}$$

De 8.10 nous déduisons que $\varphi(\alpha) = \varphi(\tau_1)$ et combiné avec 8.11 nous déduisons bien 8.6. L'application de l'hypothèse de récurrence à 8.2 nous permet de déduire que :

$$\varphi(E \oplus (x : (\tau_1, n))) \vdash_n a_2 : \varphi(\tau_2) \tag{8.13}$$

et par invariants de l'algorithme, que :

$$V_1 \subseteq V_2 \quad FV(\tau_2) \subseteq V_2 \quad FV(C_2) \subseteq V_2 \tag{8.14}$$

La déduction 8.13 est exactement ce que nous attendions en 8.7. Nous pouvons ainsi reconstituer une dérivation 64 et conclure au résultat attendu en 8.5.

Prouvons maintenant les invariants de l'algorithme :

- $V \subseteq V_2$: trivial par 8.12 et 8.14.
- $FV(\tau_2) \subseteq V_2$: directement déduit de 8.14.
- $FV(C_1 \cup C_2 \cup \{\alpha \overset{?}{\doteq}_{n+1} \tau_1\}) \subseteq V_2$: obtenu simplement en combinant les déductions 8.12 et 8.14 obtenues suite aux appels à la récurrence.

$\square$

Complétude

Théorème 8.7 (L'algorithme $\mathcal{C}$ est complet) *Soient un environnement de typage E, une expression a, un niveau courant de typage n et un ensemble de variables V.*

Si $(\tau, C, V') = \mathcal{C}(E, a, n, V)$ est défini et $FV(E) \subseteq V$ et s'il existe une substitution φ_0 préservant les niveaux et un type t_0 tels que $\mathrm{Dom}\,(\varphi_0) \subseteq V$ et $\varphi_0(E) \vdash_n a : t_0$, alors il existe une substitution φ avec $\varphi_{|V} = \varphi_0$ telle que φ est solution de C et $t_0 = \varphi(\tau)$ et $\mathrm{Dom}\,(\varphi) \subseteq V'$.

Preuve : La preuve est effectuée par cas sur la forme de l'expression a. Notons que montrer que l'algorithme est défini est trivial puisque par construction ce dernier ne peut échouer. C'est la raison pour laquelle l'énoncé du présent théorème n'a pas la forme couramment vue pour les propriétés de complétude : « si ..., alors l'algorithme n'échoue pas et ... ».

Cas a est x.

Hypothèses :

$$(\alpha, \{\tau \overset{?}{\dot{\leq}}{}^m_n \alpha\}, V \cup \{\alpha\}) = \mathcal{C}(E, x, n, V) \tag{8.1}$$
$$FV(E) \subseteq V \tag{8.2}$$
$$\mathrm{Dom}\,(\varphi_0) \subseteq V \tag{8.3}$$
$$\varphi_0(E) \vdash_n x : t_0 \tag{8.4}$$

Nous voulons prouver que :

$$\exists\, \varphi \text{ telle que } \varphi_{|V} = \varphi_0 \tag{8.5}$$
$$\varphi \text{ solution de } \{\tau \overset{?}{\dot{\leq}}{}^m_n \alpha\} \tag{8.6}$$
$$t_0 = \varphi(\alpha) \tag{8.7}$$
$$\mathrm{Dom}\,(\varphi) \subseteq V \cup \{\alpha\} \tag{8.8}$$

De 8.1, nous déduisons par la définition de l'algorithme $\mathcal{C}$ que :

$$(\tau, m) = E(x) \tag{8.9}$$
$$\alpha \notin V \text{ est une variable fraîche de niveau } n \tag{8.10}$$

Par l'hypothèse 8.4 et l'utilisation d'une règle de typage 59 nous déduisons que :

$$\varphi_0(E(x)) = (t_1, m) \tag{8.11}$$
$$t_1 \leq^m_n t_0 \tag{8.12}$$

Prenons $\varphi = \varphi_0 \oplus \{\alpha \leftarrow t_0\}$ et montrons qu'elle satisfait les propriétés que nous cherchons à démonter, à savoir 8.5–8.8.

- $$\begin{array}{rcll} \varphi_{|V} & = & (\varphi_0 \cup \{\alpha \leftarrow t_0\})_{|V} & \text{par choix de } \varphi \\ & = & \varphi_0 & \text{car par 8.10 } \alpha \notin V \end{array}$$

  comme attendu en 8.5.

- Pour satisfaire 8.6, il faut que φ soit solution de $\{\tau \overset{?}{\leq}{}^m_n \alpha\}$ et qu'elle préserve les niveaux. Nous devons donc prouver que :

$$\begin{array}{lll} & \varphi(\tau) \leq^m_n \varphi(\alpha) & \text{par définition de } \overset{?}{\leq}{}^m_n \\ \text{càd que} & \varphi(\tau) \leq^m_n t_0 & \text{par 8.7} \\ \text{càd que} & (\varphi_0 \cup \{\alpha \leftarrow t_0\})(\tau) \leq^m_n t_0 & \text{par choix de } \varphi \end{array}$$

  Par 8.9, nous savons que $\tau \in \mathrm{Dom}\,(E)$. Or par 8.2, $FV(E) \subseteq V$ et par 8.10 $\alpha \notin V$. Donc α n'apparaît pas dans E. Ainsi nous supprimons la partie de la substitution qui affecte α et il nous reste à vérifier que $\varphi_0(\tau) \leq^m_n t_0$. Par l'hypothèse 8.11, on sait que, $\varphi_0(E(x)) = (t_1, m)$ et que par 8.9 $E(x) = (\tau, m)$. Donc $\varphi_0(\tau) = t_1$. Introduisons ceci dans ce qu'il nous restait à vérifier et nous devons donc prouver maintenant que $t_1 \leq^m_n t_0$ ce qui nous est directement fourni par la déduction 8.12.
  De plus, par hypothèse φ_0 préserve les niveaux et, α et τ_0 étant tout deux de niveau n, la substitution $\{\alpha \leftarrow t_0\}$ les préserve également. Ainsi, comme attendu, φ étant égale par construction à $\varphi_0 \oplus \{\alpha \leftarrow t_0\}$, elle préserve les niveaux.
- Par choix de φ, nous avons $\varphi(\alpha) = (\varphi_0 \oplus \{\alpha \leftarrow t_0\})(\alpha) = t_0$ comme attendu en 8.7.
- Par choix de φ, nous avons $\mathrm{Dom}\,(\varphi) = \mathrm{Dom}\,(\varphi_0) \cup \mathrm{Dom}\,(\{\alpha \leftarrow t_0\}) = \mathrm{Dom}\,(\varphi_0) \cup \{\alpha\}$. Puisque par 8.3 $\mathrm{Dom}\,(\varphi_0) \subseteq V$, nous avons bien $\mathrm{Dom}\,(\varphi) = V \cup \{\alpha\}$ comme attendu en 8.8.

Cas a est i.
Hypothèses :

$$(\mathtt{int}, \{\}, V) = \mathcal{C}(E, i, n, V) \tag{8.1}$$
$$FV(E) \subseteq V \tag{8.2}$$
$$\mathrm{Dom}\,(\varphi_0) \subseteq V \tag{8.3}$$
$$\varphi_0(E) \vdash_n i : \mathtt{int} \tag{8.4}$$

Nous voulons prouver que :

$$\exists\, \varphi \text{ telle que } \varphi_{|V} = \varphi_0 \tag{8.5}$$
$$\varphi \text{ solution de } \{\} \tag{8.6}$$
$$\mathtt{int} = \varphi(\mathtt{int}) \tag{8.7}$$
$$\mathrm{Dom}\,(\varphi) \subseteq V \tag{8.8}$$

La substitution φ_0 est *id*. Nous choisissons alors trivialement $\varphi = id$ et chacun des buts 8.5–8.8 est immédiat.

Cas a est $(a_1\ a_2)$.
Hypothèses :

$$(\alpha, C_1 \cup C_2 \cup \{\tau_1 \overset{?}{=}{}_n \tau_2 \to \alpha\}, V_2 \cup \{\alpha\}) = \mathcal{C}(E, (a_1\ a_2), n, V) \tag{8.1}$$
$$FV(E) \subseteq V \tag{8.2}$$
$$\mathrm{Dom}\,(\varphi_0) \subseteq V \tag{8.3}$$
$$\varphi_0(E) \vdash_n (a_1\ a_2) : t_0 \tag{8.4}$$

Nous voulons prouver que :

$$\exists\, \varphi \text{ telle que } \varphi_{|V} = \varphi_0 \tag{8.5}$$
$$\varphi \text{ solution de } C_1 \cup C_2 \cup \{\tau_1 \overset{?}{=}{}_n \tau_2 \to \alpha\} \tag{8.6}$$
$$t_0 = \varphi(\alpha) \tag{8.7}$$
$$\mathrm{Dom}\,(\varphi) \subseteq V_2 \cup \{\alpha\} \tag{8.8}$$

De 8.1, nous déduisons par la définition de l'algorithme $\mathcal{C}$ que :

$$(\tau_1, C_1, V_1) = \mathcal{C}(E, a_1, n, V) \tag{8.9}$$
$$(\tau_2, C_2, V_2) = \mathcal{C}(E, a_2, n, V_1) \tag{8.10}$$
$$\alpha \notin V_2 \text{ est une variable fraîche de niveau } n \tag{8.11}$$

Par l'hypothèse 8.4 et l'utilisation d'une règle de typage 61 nous déduisons que :

$$\varphi_0(E) \vdash_n a_1 : t_1 \to t_0 \tag{8.12}$$
$$\varphi_0(E) \vdash_n a_2 : t_1 \tag{8.13}$$

Nous pouvons alors appliquer l'hypothèse de récurrence sur 8.2, 8.3, 8.9 et 8.12 pour déduire que :

$$\exists \varphi_1 \text{ avec } \varphi_{1|V} = \varphi_0 \tag{8.14}$$
$$\varphi_1 \text{ solution de } C_1 \tag{8.15}$$
$$t_1 \to t_0 = \varphi_1(\tau_1) \tag{8.16}$$
$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.17}$$

et que (par invariant dûs à la construction du système de contraintes par l'algorithme $\mathcal{C}$) :

$$V \subseteq V_1 \quad FV(\tau_1) \subseteq V_1 \quad FV(C_1) \subseteq V_1 \tag{8.18}$$

Nous voudrions maintenant appliquer l'hypothèse de récurrence sur 8.10 et 8.13, mais avec φ_1 et non φ_0. Il faut donc d'abord prouver que :

$$\varphi_0(E) \vdash_n a_2 : t_1 \Rightarrow \varphi_1(E) \vdash_n a_2 : t_1 \tag{8.19}$$
$$FV(E) \subseteq V_1 \tag{8.20}$$
$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.21}$$
$$\varphi_1 \text{ préserve les niveaux} \tag{8.22}$$

Puisque $FV(E) \subseteq V$ (hypothèse 8.2) et que par 8.18, $V \subseteq V_1$, nous satisfaisons 8.20. Le but 8.21 correspond directement à la déduction 8.17. Quant à 8.19, il suffit de prouver que $\varphi_0(E) = \varphi_1(E)$, ce qui est vérifié, car par 8.2 $FV(E) \subseteq V$ et par 8.14 $\varphi_{1|V} = \varphi_0$. Pour terminer, la condition 8.22 nous est immédiatement donnée par la déduction 8.15.

Nous pouvons donc appliquer l'hypothèse de récurrence comme voulu ci-dessus et déduire que :

$$\exists \varphi_2 \text{ avec } \varphi_{2|V_1} = \varphi_1 \tag{8.23}$$
$$\varphi_2 \text{ solution de } C_2 \tag{8.24}$$
$$t_1 = \varphi_2(\tau_2) \tag{8.25}$$
$$\mathrm{Dom}\,(\varphi_2) \subseteq V_2 \tag{8.26}$$

et que (par invariant dûs à la construction du système de contraintes par l'algorithme $\mathcal{C}$) :

$$V_1 \subseteq V_2 \quad FV(\tau_2) \subseteq V_2 \quad FV(C_2) \subseteq V_2 \tag{8.27}$$

Prenons $\varphi = \varphi_2 \oplus \{\alpha \leftarrow t_0\}$ et vérifions qu'elle satisfait les propriétés que nous cherchons à démonter, à savoir 8.5–8.8.

•

$$\begin{array}{rcll}
\varphi_{|V} & = & (\varphi_2 \oplus \{\alpha \leftarrow t_0\})_{|V} & \text{par choix de } \varphi \\
 & = & \varphi_{2|V} & \text{car } \alpha \notin V_2 \text{ et } V \subseteq V_1 \subseteq V_2 \\
 & = & \varphi_{1|V} & \text{car par } 8.23, \varphi_{2|V_1} = \varphi_1 \text{ et } V \subseteq V_1 \\
 & = & \varphi_0 & \text{par } 8.14, \text{ comme attendu en } 8.5.
\end{array}$$

- Pour satisfaire 8.6, φ doit être solution de $C_1 \cup C_2 \cup \{\tau_1 \stackrel{?}{=}_n \tau_2 \to \alpha\}$ et préserver les niveaux.
  - φ est solution de C_1 car $\varphi_{|V_1} = \varphi_1$ et $FV(C_1) \subseteq V_1$, or φ_1 solution de C_1
  - φ est solution de C_2 car $\varphi_{|V_2} = \varphi_2$ et $FV(C_2) \subseteq V_2$, or φ_2 solution de C_2
  - φ est solution de $\{\tau_1 \stackrel{?}{=}_n \tau_2 \to \alpha\}$ car :
    - $\varphi(\tau_1) = \varphi_1(\tau_1) = t_1 \to t_0$
    - $\varphi(\tau_2 \to \alpha) = \varphi(\tau_2) \to \varphi(\alpha) = \varphi_2(\tau_2) \to t_0 = t_1 \to t_0$

  De plus, par 8.24 φ_2 respecte les niveaux. Par l'hypothèse 8.11, α est une variable de niveau n et t_0 est également un type de niveau n. Ainsi $\{\alpha \leftarrow t_0\}$ respecte aussi les niveaux et donc $\varphi = \varphi_2 \oplus \{\alpha \leftarrow t_0\}$ également.
- $t_0 = \varphi(\alpha)$ par construction de φ, comme attendu en 8.7.
- Pour satisfaire 8.8, il faut que Dom $(\varphi) \subseteq V_2 \cup \{\alpha\}$. Par choix de φ, nous avons Dom (φ) $=$ Dom $(\varphi_2 \oplus \{\alpha \leftarrow t_0\})$ $=$ Dom $(\varphi_2) \oplus \{\alpha\}$. Or par 8.26, Dom $(\varphi_2) \subseteq V_2$. Donc nous avons bien Dom $(\varphi) \subseteq V_2 \cup \{\alpha\}$

Cas a est $\lambda x.a_1$.

Hypothèses :

$$(\alpha \to \tau, C, V_1) = \mathcal{C}(E, \lambda x.a_1, n, V) \tag{8.1}$$
$$FV(E) \subseteq V \tag{8.2}$$
$$\text{Dom}\,(\varphi_0) \subseteq V \tag{8.3}$$
$$\varphi_0(E) \vdash_n \lambda x.a_1 : t_1 \to t_2 \tag{8.4}$$

Nous voulons prouver que :

$$\exists\, \varphi \text{ telle que } \varphi_{|V} = \varphi_0 \tag{8.5}$$
$$\varphi \text{ solution de } C \tag{8.6}$$
$$t_1 \to t_2 = \varphi(\alpha \to \tau) \tag{8.7}$$
$$\text{Dom}\,(\varphi) \subseteq V_1 \tag{8.8}$$

De 8.1, nous déduisons par la définition de l'algorithme $\mathcal{C}$ que :

$$(\tau, C, V_1) = \mathcal{C}(E \oplus (x : (\alpha, n)), a_1, n, V \cup \{\alpha\}) \tag{8.9}$$
$$\alpha \notin V \text{ est une variable fraîche de niveau } n \tag{8.10}$$

Par l'hypothèse 8.4 et l'utilisation d'une règle de typage 62 nous déduisons que :

$$\varphi_0(E) \oplus (x : (t_1, n)) \vdash_n b : t_2 \tag{8.11}$$
$$t_1 \in \tau^n \tag{8.12}$$

Nous voudrions appliquer l'hypothèse de récurrence sur 8.11, non pas avec φ_0, mais avec $\varphi_0' = \varphi_0 \oplus \{\alpha \leftarrow t_1\}$. Vérifions qu'on a bien le droit, autrement dit que φ_0' présente bien les hypothèses nécessaires à l'application de l'hypothèse de récurrence. Il faut donc prouver que :

$$\text{Dom}\,(\varphi_0') \subseteq V \cup \{\alpha\} \tag{8.13}$$
$$\varphi_0'(E) \oplus (x : (t_1, n)) \vdash_n b : t_2 \tag{8.14}$$
$$\varphi_0' \text{ préserve les niveaux} \tag{8.15}$$

Puisque Dom $(\varphi_0') =$ Dom $(\varphi_0) \cup \{\alpha\}$ et que par 8.3, $Dom(\varphi_0) \subseteq V$, nous vérifions bien le but 8.13.

Quant à 8.14, cela revient à prouver que $\varphi_0'(E) = \varphi_0(E)$, ce qui est vérifié car $FV(E) \subseteq V$, Dom $(\varphi_0') \subseteq V \cup \{\alpha\}$ or $\alpha \notin V$.

Pour terminer, φ'_0 respecte effectivement les niveaux puisque par construction elle est égale à $\varphi_0 \oplus \{\alpha \leftarrow t_1\}$, que par hypothèses φ_0 les respecte et que $\{\alpha \leftarrow t_1\}$ également (α et t_1 étant tous deux de niveau n).

Nous pouvons donc appliquer l'hypothèse de récurrence avec φ'_0 comme désiré et nous déduisons que :

$$\exists\, \varphi_1 \text{ avec } \varphi_{1|V \cup \{\alpha\}} = \varphi'_0 \tag{8.16}$$
$$\varphi_1 \text{ solution de } C \tag{8.17}$$
$$t_2 = \varphi_1(\tau) \tag{8.18}$$
$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.19}$$

et que (par invariants dûs à la construction du système de contraintes) :

$$V \cup \{\alpha\} \subseteq V_1 \quad FV(\tau) \subseteq V_1 \quad FV(C) \subseteq V_1 \tag{8.20}$$

Prenons $\varphi = \varphi_1$ et vérifions qu'elle satisfait les propriétés que nous cherchons à démonter, à savoir : 8.5–8.8.

- Par choix de φ, nous avons $\varphi_{|V} = \varphi_{1|V}$. Par 8.16, on sait que $\varphi_{1|V \cup \{\alpha\}} = \varphi'_0$. Donc, par construction de φ'_0 nous avons $\varphi_{1|V \cup \{\alpha\}} = \varphi_0 \oplus \{\alpha \leftarrow t_1\}$. Par hypothèse 8.3, $\mathrm{Dom}\,(\varphi_0) \subseteq V$ et par 8.10 $\alpha \notin V$. Donc φ_0 ne touche pas à α. Donc on a bien que $\varphi_{1|V} = \varphi_0$ comme attendu en 8.5.
- Pour satisfaire 8.6, φ doit être solution de C, ce qui est trivialement obtenu par la déduction 8.17.
  De plus, φ préserve les niveaux puisque par construction elle est égale à φ_1 qui elle-même les préserve par 8.17.
- Pour satisfaire 8.7, nous devons vérifier que $t_1 \to t_2 = \varphi(\alpha \to \tau)$.

$$\begin{array}{rcll} \varphi(\alpha \to \tau) & = & \varphi_1(\alpha \to \tau) & \\ & = & \varphi_1(\alpha) \to \varphi_1(\tau) & \\ & = & \varphi_1(\alpha) \to t_2 & \text{par 8.18} \\ & = & \varphi'_0(\alpha) \to t_2 & \text{car } \varphi_{1|V \cup \{\alpha\}} = \varphi'_0 \\ & = & t_1 \to t_2 & \end{array}$$

- Nous avons bien $\mathrm{Dom}\,(\varphi) \subseteq V_1$ car par choix de φ $\mathrm{Dom}\,(\varphi) = \mathrm{Dom}\,(\varphi_1)$ et par 8.19 $\mathrm{Dom}\,(\varphi_1) \subseteq V_1$, comme attendu en 8.8.

Cas a est `let` $x = a_1$ `in` a_2.

Hypothèses :

$$(\tau_2, C_1 \cup C_2, V_2) = \mathcal{C}(E, \texttt{let}\ x = a_1\ \texttt{in}\ a_2, n, V) \tag{8.1}$$
$$FV(E) \subseteq V \tag{8.2}$$
$$\mathrm{Dom}\,(\varphi_0) \subseteq V \tag{8.3}$$
$$\varphi_0(E) \vdash_n \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : t_2 \tag{8.4}$$

Nous voulons prouver que :

$$\exists\, \varphi \text{ telle que } \varphi_{|V} = \varphi_0 \tag{8.5}$$
$$\varphi \text{ solution de } C_1 \cup C_2 \tag{8.6}$$
$$t_2 = \varphi(\tau_2) \tag{8.7}$$
$$\mathrm{Dom}\,(\varphi) \subseteq V_2 \tag{8.8}$$

De 8.1, nous déduisons par la définition de l'algorithme $\mathcal{C}$ que :

$$(\tau_1, C_1, V_1) = \mathcal{C}(E, a_1, n+1, V) \tag{8.9}$$
$$(\tau_2, C_2, V_2) = \mathcal{C}(E \oplus (x : (\tau_1, n), a_2, n, V_1) \tag{8.10}$$

Par l'hypothèse 8.4 et l'utilisation d'une règle de typage 63 nous déduisons que :

$$\varphi_0(E) \vdash_{n+1} a_1 : t_1 \tag{8.11}$$
$$\varphi_0(E) \oplus (x : (t_1, n)) \vdash_n a_2 : t_2 \tag{8.12}$$

Nous pouvons donc appliquer l'hypothèse de récurrence sur 8.11 et déduire que :

$$\exists \varphi_1 \text{ telle que } \varphi_{1|V} = \varphi_0 \tag{8.13}$$
$$\varphi_1 \text{ solution de } C_1 \tag{8.14}$$
$$t_1 = \varphi_1(\tau_1) \tag{8.15}$$
$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.16}$$

et que (par invariants dûs à la construction du système de contraintes) :

$$V \subseteq V_1 \quad FV(\tau_1) \subseteq V_1 \quad FV(C_1) \subseteq V_1 \tag{8.17}$$

Nous allons maintenant appliquer l'hypothèse de récurrence sur 8.12 mais avec φ_1 et non φ_0. Il faut d'abord vérifier que φ_1 présente bien les hypothèses nécessaires à l'application de cette récurrence, à savoir :

$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.18}$$
$$\varphi_1(E) \oplus (x : (t_1, n)) \vdash_n c : t_2 \tag{8.19}$$
$$\varphi_1 \text{ préserve les niveaux} \tag{8.20}$$

Les buts 8.18 et 8.20 nous sont directement fournis par les déductions 8.16 et 8.14. Établir 8.19 revient à montrer que $\varphi_1(E) \oplus (x : (t_1, n)) = \varphi_0(E) \oplus (x : (t_1, n))$, c'est-à-dire que $\varphi_1(E) = \varphi_0(E)$. Par 8.2 on sait que $FV(E) \subseteq V$, donc $\varphi_1(E) = \varphi_{1|V}(E)$ qui est égal, par 8.13 à $\varphi_0(E)$.

Nous pouvons donc appliquer l'hypothèse de récurrence comme attendue et déduire que :

$$\exists \varphi_2 \text{ telle que } \varphi_{2|V_1} = \varphi_1 \tag{8.21}$$
$$\varphi_2 \text{ solution de } C_2 \tag{8.22}$$
$$t_2 = \varphi_2(\tau_2) \tag{8.23}$$
$$\mathrm{Dom}\,(\varphi_2) \subseteq V_2 \tag{8.24}$$
$$\tag{8.25}$$

et que (par invariants dûs à la construction du système de contraintes) :

$$V_1 \subseteq V_2 \quad FV(\tau_2) \subseteq V_2 \quad FV(C_2) \subseteq V_2 \tag{8.26}$$

Prenons $\varphi = \varphi_2$ et vérifions qu'elle satisfait les propriétés que nous cherchons à démonter, à savoir : 8.5–8.8.

- φ doit être telle que $\varphi_{|V} = \varphi_0$. Nous avons choisi $\varphi = \varphi_2$ donc il faut prouver que $\varphi_{2|V} = \varphi_0$. On sait par 8.21 que $\varphi_{2|V_1} = \varphi_1$ et que par 8.13 $\varphi_{1|V} = \varphi_0$. Or par 8.17 et 8.26 nous avons $V \subseteq V_1 \subseteq V_2$, donc $\varphi_{2|V} = \varphi_0$ comme attendu en 8.5.
- Afin de satisfaire 8.6, il faut que φ, soit solution de $C_1 \cup C_2$. Par choix de φ, il faut donc que φ_2, soit solution de $C_1 \cup C_2$. Par 8.22 φ_2 est solution de C_2. De plus on sait que $FV(C_1) \subseteq V_1$. Or $\varphi_{2|V_1} = \varphi_1$ et par 8.6 φ_1 est solution de C_1. Donc φ_2 est aussi solution de C_2, d'où solution de $C_1 \cup C_2$ comme attendu en 8.6.
  De plus, par l'hypothèse 8.22, nous savons que φ_2 préserve les niveaux. Par choix de φ, celle-ci étant égale à φ_2, nous avons bien que φ préserve les niveaux.
- Par 8.23, $t_2 = \varphi(\tau_2)$, comme attendu en 8.7.
- Par choix de φ_2, nous avons $\mathrm{Dom}\,(\varphi) = \mathrm{Dom}\,(\varphi_2)$. Or par 8.24, $\mathrm{Dom}\,(\varphi_2) \subseteq V_2$ donc $\mathrm{Dom}\,(\varphi) \subseteq V_2$, comme attendu en 8.8.

Cas a est `let rec` $x = a_1$ `in` a_2.

Hypothèses :

$$(\tau_2, C_1 \cup C_2 \cup \{\alpha \overset{?}{=}_{n+1} \tau_1\}, V_2) = \mathcal{C}(E, \texttt{let rec } x = a_1 \texttt{ in } a_2, n, V) \tag{8.1}$$
$$FV(E) \subseteq V \tag{8.2}$$
$$\mathrm{Dom}\,(\varphi_0) \subseteq V \tag{8.3}$$
$$\varphi_0(E) \vdash_n \texttt{let rec } x = a_1 \texttt{ in } a_2 : t_2 \tag{8.4}$$

Nous voulons prouver que :

$$\exists\,\varphi \text{ telle que } \varphi_{|V} = \varphi_0 \tag{8.5}$$
$$\varphi \text{ solution de } C_1 \cup C_2 \cup \{\alpha \overset{?}{=}_{n+1} \tau_1\} \tag{8.6}$$
$$t_2 = \varphi(\tau_2) \tag{8.7}$$
$$\mathrm{Dom}\,(\varphi) \subseteq V_2 \tag{8.8}$$

De 8.1, nous déduisons par la définition de l'algorithme $\mathcal{C}$ que :

$$(\tau_1, C_1, V_1) = \mathcal{C}(E \oplus (x : (\alpha, n), a_1, n+1, V \cup \{\alpha\}) \tag{8.9}$$
$$(\tau_2, C_2, V_2) = \mathcal{C}(E \oplus (x : (\tau_1, n), a_2, n, V_1) \tag{8.10}$$
$$\alpha \notin V \text{ est une variable fraîche de niveau } n+1 \tag{8.11}$$

Par l'hypothèse 8.4 et l'utilisation d'une règle de typage 64 nous déduisons que :

$$\varphi_0(E) \oplus (x : (t_1, n)) \vdash_{n+1} a_1 : t_1 \tag{8.12}$$
$$\varphi_0(E) \oplus (x : (t_1, n)) \vdash_n a_2 : t_2 \tag{8.13}$$

Nous voulons appliquer l'hypothèse de récurrence sur 8.12, non pas avec φ_0, mais avec $\varphi'_0 = \varphi_0 \oplus \{\alpha \leftarrow t_1\}$. Il faut donc d'abord vérifier que φ'_0 présente bien les hypothèses nécessaires à l'application de cette récurrence, à savoir :

$$\mathrm{Dom}\,(\varphi'_0) \subseteq V \cup \{\alpha\} \tag{8.14}$$
$$\varphi'_0(E) \oplus (x : (t_1, n)) \vdash_{n+1} b : t_1 \tag{8.15}$$
$$\varphi'_0 \text{ respecte les niveaux} \tag{8.16}$$

Par choix de φ'_0, nous avons $\mathrm{Dom}\,(\varphi'_0) = \mathrm{Dom}\,(\varphi_0) \cup \mathrm{Dom}\,(\{\alpha \leftarrow t_1\}) = \mathrm{Dom}\,(\varphi_0) \cup \{\alpha\}$. Or par 8.3 $\mathrm{Dom}\,(\varphi_0) \subseteq V$, donc on a bien $\mathrm{Dom}\,(\varphi'_0) \subseteq V \cup \{\alpha\}$ comme attendu en 8.14.

Établir 8.15 revient à prouver que $\varphi'_0(E) = \varphi_0(E)$. On sait que $\varphi'_0(E) = (\varphi_0 \oplus \{\alpha \leftarrow t_1\})(E)$. Or par 8.2 $FV(E) \subseteq V$ et par 8.11 $\alpha \notin V$, donc $\varphi'_0(E) = \varphi_0(E)$.

Comme attendu en 8.16, φ'_0 préserve les niveau puisque par hypothèses φ_0 le fait et que α et t_1 sont tous deux de niveaux $n+1$.

Nous pouvons donc appliquer notre hypothèse de récurrence comme désiré et déduire que :

$$\exists\,\varphi_1 \text{ telle que } \varphi_{1|V \cup \{\alpha\}} = \varphi'_0 \tag{8.17}$$
$$\varphi_1 \text{ solution de } C_1 \tag{8.18}$$
$$t_1 = \varphi_1(\tau_1) \tag{8.19}$$
$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.20}$$

et que (par invariants dûs à la construction du système de contraintes) :

$$V \cup \{\alpha\} \subseteq V_1 \quad FV(\tau_1) \subseteq V_1 \quad FV(C_1) \subseteq V_1 \tag{8.21}$$

Nous voulons maintenant appliquer l'hypothèse de récurrence sur 8.13, non pas avec φ_0, mais avec φ_1. Il faut donc d'abord vérifier que φ_1 satisfait les hypothèses nécessaires à l'application de la récurrence, c'est-à-dire :

$$\mathrm{Dom}\,(\varphi_1) \subseteq V_1 \tag{8.22}$$

$$\varphi_1(E) \oplus (x : (t_1, n)) \vdash_n c : t_2 \tag{8.23}$$

$$\varphi_1 \text{ préserve les niveaux} \tag{8.24}$$

Les buts 8.22 et 8.24 sont respectivement directement issus des déductions 8.20 et 8.18.

Montrer 8.23 revient à prouver que $\varphi_1(E) = \varphi_0(E)$. On sait que $\varphi_{1|V\cup\{\alpha\}} = \varphi_0' = \varphi_0 \oplus \{\alpha \leftarrow t_1\}$. Donc $\varphi_{1|V} = \varphi_0$ car $\alpha \notin \mathrm{Dom}\,(\varphi_0)$. Comme $FV(E) \subseteq V$, on a bien que $\varphi_1(E) = \varphi_{1|V}(E) = \varphi_0(E)$.

Nous pouvons donc appliquer l'hypothèse de récurrence comme désiré et déduire que :

$$\exists\, \varphi_2 \text{ telle que } \varphi_{2|V_1} = \varphi_1 \tag{8.25}$$

$$\varphi_2 \text{ solution de } C_2 \tag{8.26}$$

$$t_2 = \varphi_2(\tau_2) \tag{8.27}$$

$$\mathrm{Dom}\,(\varphi_2) \subseteq V_2 \tag{8.28}$$

et que (par invariants dûs à la construction du système de contraintes) :

$$V_1 \subseteq V_2 \quad FV(\tau_2) \subseteq V_2 \quad FV(C_2) \subseteq V_2 \tag{8.29}$$

Prenons $\varphi = \varphi_2$ et vérifions qu'elle satisfait les propriétés que nous cherchons à démonter, à savoir : 8.5–8.8.

- Il faut que $\varphi_{|V} = \varphi_0$, donc par choix de φ, que $\varphi_{2|V} = \varphi_0$. Par 8.25 nous savons que $\varphi_{2|V_1} = \varphi_1$. Or, par 8.21 $V \cup \{\alpha\} \subseteq V_1$. Donc

  $$\varphi_{2|V\cup\{\alpha\}} = \varphi_{1|V\cup\{\alpha\}} = \varphi_0' = \varphi_0 \oplus \{\alpha \leftarrow t_1\}$$

  De plus $V \subseteq V \cup \{\alpha\}$, donc

  $$\varphi_{2|V} = \varphi_{1|V} = \varphi'_{0|V} = (\varphi_0 \oplus \{\alpha \leftarrow t_1\})_{|V}$$

  Puisque $\alpha \notin V$ et que $\mathrm{Dom}\,(\varphi_0) \subseteq V$, nous nous déduisons que $\varphi_{0|V} = \varphi_0$ et que $\{\alpha \leftarrow t_1\}_{|V} = \{\}$. Ainsi nous avons bien $\varphi_{|V} = \varphi_{2|V} = \varphi_0$ comme attendu en 8.5.
- Par choix de φ, nous devons montrer que φ_2 est solution de $C_1 \cup C_2 \cup \{\alpha \mapsto \tau_1\}$. Par 8.26, φ_2 est solution de C_2. Nous savons $FV(C_1) \subseteq V_1$, or $\varphi_{2|V} = \varphi_1$ et φ_1 est solution de C_1. Donc φ_2 est solution de C_1. Nous devons finalement vérifier que φ_2 est solution de $\{\alpha \mapsto \tau_1\}$.

  $$\begin{array}{lcll}
  \varphi_2(\alpha) & = & \varphi_{2|V_1}(\alpha) & \text{car } \alpha \in V_1 \text{ puisque } V_1 \subseteq V \cup \{\alpha\} \\
  & = & \varphi_1(\alpha) & \text{par } 8.25 \\
  & = & \varphi_{1|V\cup\{\alpha\}}(\alpha) & \\
  & = & \varphi_0'(\alpha) & \text{par } 8.17 \\
  & = & (\varphi_0 \oplus \{\alpha \leftarrow t_1\})(\alpha) & \\
  & = & t_1 &
  \end{array}$$

  De plus φ préserve les niveaux, puisque égale par choix à φ_2 qui respecte les niveaux par l'hypothèse 8.26.

  $$\begin{array}{lcll}
  \varphi_2(\tau_1) & = & \varphi_{2|V_1}(\tau_1) & \text{car par } 8.21, FV(\tau_1) \subseteq V_1 \\
  & = & \varphi_1(\tau_1) & \text{par } 8.25 \\
  & = & t_1 & \text{par } 8.19
  \end{array}$$

  Donc φ_2 est bien solution de $\{\alpha \mapsto \tau_1\}$ et donc de $C_1 \cup C_2 \cup \{\alpha \mapsto \tau_1\}$ et φ satisfait bien 8.6.
- $t_2 = \varphi_2(\tau_2)$ par la déduction 8.27, comme attendu en 8.7.
- $\mathrm{Dom}\,(\varphi_2) \subseteq V_2$ par la déduction 8.28, comme attendu en 8.8.

□

8.4.4 La résolution des ensembles de contraintes : préambule

Maintenant que nous savons comment générer un ensemble de contraintes d'unification et de semi-unification à partir d'un programme, et que nous avons montré que ce processus est correct et complet, nous allons nous intéresser à la résolution de cet ensemble.

Nous allons donc montrer maintenant la correction de notre algorithme de résolution de contraintes. Comme nous l'avons dit précédemment, notre algorithme d'inférence résout (au vol) des contraintes d'unification préalablement aux contraintes de semi-unification. Nous allons donc commencer par démontrer qu'il est correct et complet de procéder ainsi vis-à-vis de la résolution d'un ensemble de contraintes. En second lieu, nous effectuerons un rappel sur l'unification. Puis nous terminerons en prouvant que notre algorithme, s'il réussit calcule bien la solution principale d'un tel ensemble. Les preuves de cette section reviennent donc à établir la propriété étiquetée 3 (entre le système de contraintes et la solution du système) sur le schéma présenté en section 8.4.1.

La résolution d'un ensemble de contraintes généré à partir d'un programme étant ainsi segmenté en trois phases, nous jugeons bon de les présenter séparément ainsi que leur propriétés respectives.

Définition 8.11 (Variable libre dans un ensemble de contraintes) *Une variable de type α est* libre *dans un ensemble de contraintes C si elle apparaît dans un des types contenus dans les contraintes de C.*

8.4.5 La séparation entre unification et semi-unification

Lemme 8.2 (Composition de solutions) *Soit deux substitutions μ et ψ préservant les niveaux et un ensemble I de contraintes de semi-unification. Alors $\psi \circ \mu$ est solution de I si et seulement si ψ est solution de $\mu(I)$.*

Preuve : La preuve est simplement effectuée en appliquant la définition, pour une substitution, d'être solution d'un ensemble de contraintes de semi-unification.

- Si $\psi \circ \mu$ est solution de I, alors pour toute contrainte $\{\tau_1 \overset{?}{\leq}{}^m_n \tau_2\}$ de I, il existe σ préservant les niveaux et telle que $\sigma \circ \psi \circ \mu(\tau_1) = \psi \circ \mu(\tau_2)$
- Si ψ est solution de $\mu(I)$, alors pour toute contrainte $\{\tau_1 \overset{?}{\leq}{}^m_n \tau_2\}$ de I, c'est-à-dire pour toute contrainte $\{\mu(\tau_1) \overset{?}{\leq}{}^m_n \mu(\tau_2)\}$ de $\mu(I)$, il existe σ préservant les niveaux et telle que $\sigma \circ \psi \circ \mu(\tau_1) = \psi \circ \mu(\tau_2)$

Ces deux propriétés se réduisant à la même chose, nous avons bien équivalence entre elles.

□

Correction de la décomposition des ensembles de contraintes

Théorème 8.8 (Décomposer les ensembles est correct) *Soit C un ensemble de contraintes d'unification et de semi-unification. Notons E le sous-ensemble de C composé des contraintes d'unification et I celui composé des contraintes de semi-unification. On peut alors écrire C sous la forme $C = E \cup I$.*

Si μ est solution de E (c'est-à-dire μ est un unificateur de E), et si φ est solution de $\mu(I)$, alors $\varphi \circ \mu$ est solution de C.

Preuve : La substitution μ étant solution de E, pour toute contrainte $\{\tau_1 \overset{?}{=}{}_n \tau_2\}$ de E, nous avons $\mu(\tau_1) = \mu(\tau_2)$. Donc pour toute substitution, φ en particulier, nous avons $\varphi \circ \mu(\tau_1) = \varphi \circ \mu(\tau_2)$. Ceci signifie que pour toute contrainte $\{\tau_1 \overset{?}{=}{}_n \tau_2\}$ de E, la substitution $\varphi \circ \mu$ est solution.

La substitution φ étant solution de $\mu(I)$, pour toute contrainte $\{\mu(\tau_1) \overset{?}{\leq}{}^m_n \mu(\tau_2)\}$ de $\mu(I)$, nous avons $\varphi \circ \mu(\tau_1) \leq^m_n \varphi \circ \mu(\tau_2)$. Par le lemme de composition de solutions (lemme

8.2), ceci signifie que pour toute contrainte $\{\tau_1 \overset{?}{\leq}{}_n^m \tau_2\}$ de I, la substitution $\varphi \circ \mu$ est solution.

Ainsi, la substitution $\varphi \circ \mu$ étant à la fois solution de E et solution de I, elle est effectivement solution de C. □

Complétude de la décomposition des ensembles de contraintes

Théorème 8.9 (Décomposer les ensembles est complet) *Soit C un ensemble de contraintes d'unification et de semi-unification. Notons E le sous-ensemble de C composé des contraintes d'unification et I celui composé des contraintes de semi-unification. On peut alors écrire C sous la forme $C = E \cup I$.*

Si la substitution φ est solution de C, alors il existe une substitution μ, unificateur principal de E, et il existe ψ telle que $\varphi = \psi \circ \mu$ et ψ est solution de $\mu(I)$.

Preuve : Si φ est solution de $E \cup I$, alors φ est solution de E et φ est solution de I. Donc φ est un unificateur de toutes les équations de E.

Puisque E a une solution (φ en l'occurrence), c'est-à-dire admet un unificateur, il admet également un *unificateur principal* μ. Ainsi φ peut s'écrire sous la forme $\psi \circ \mu$.

Il nous reste alors à vérifier que ψ est solution de $\mu(I)$. Nous savons que $\psi \circ \mu$ est solution de I, donc par le lemme de composition de solutions (8.2) démontré précédemment, nous déduisons que ψ est effectivement solution de $\mu(I)$.

□

8.4.6 La résolution des contraintes d'unification

Théorème 8.10 (Existence d'un algorithme d'unification préservant les niveaux) *Soit E un ensemble de contraintes d'unification entre types bien formés. Il existe un algorithme* `mgu` *qui retourne une substitution μ préservant les niveaux et qui est une solution principale de E, ou échoue, affirmant que E n'a pas de solution.*

Cet algorithme est correct en ce sens que si $\mu = \mathtt{mgu}(E)$ est définie, alors μ est un unificateur de E (autrement dit, $\mu(\tau_1) =_n \mu(\tau_2)$ pour toute équation $\{\tau_1 \overset{?}{=}_n \tau_2\}$ de E).

Cet algorithme est complet et principal en ce sens que s'il existe une substitution ψ solution de l'ensemble E et préservant les niveaux, alors $\mu = \mathtt{mgu}(E)$ est définie et il existe une substitution ξ préservant également les niveaux, telle que $\psi = \xi \circ \mu$.

Nous ne donnons pas ici la description de cet algorithme pour deux raisons. D'une part ceci rallongerait de manière excessive ce chapitre. D'autre part, sa description et sa formalisation rappelleraient fortement la description de l'algorithme $\mathtt{mgu}_V$ que nous avons déjà étudié lors de la présentation de notre analyse d'exceptions.

En effet, l'algorithme d'unification dont il est question ici s'apparente à celui bien connu de Robinson [51], muni de quelques extensions mineures permettant la préservation des niveaux. Le lecteur intéressé pourra trouver plus de détails dans des ouvrages tels que [62, 45].

8.4.7 La résolution des contraintes de semi-unification

Afin de simplifier la présentation de l'algorithme de résolution de contraintes de semi-unification, nous considérerons donc que les ensembles qu'il traite ne comportent plus de contraintes d'unification, c'est-à-dire qu'elles ont toutes été résolues préalablement. On se convaincra aisément que, l'ordre de résolution n'ayant pas d'importance, cette restriction n'entache en rien la présentation de l'algorithme de résolution en lui-même.

Définition 8.12 (Instance triviale aux niveaux m et n) *Soient τ un type, n un niveau courant et m un niveau de généralisation. On appelle « instance triviale » de τ, l'instance la plus générale de τ, c'est-à-dire dans laquelle toutes les variables de τ de niveau strictement supérieur à m (donc considérées comme généralisées) ont été remplacées par des variables fraîches de niveau courant n.*

Ainsi τ' est une instance triviale de τ aux niveaux m et n si $\tau' = \tau\{\alpha^{>m} \leftarrow \alpha'^{n}\}$ avec α'^{n} fraîche c'est-à-dire n'apparaissant pas dans le problème.

Définition 8.13 (Contrainte trivialement satisfaite) *Une contrainte de semi-unification $\tau_1 \overset{?}{\leq}{}^m_n \tau_2$ est dite* trivialement satisfaite *si $\tau_1 \leq^m_n \tau_2$, c'est-à-dire qu'il existe une substitution ψ telle que $\psi(\tau_1) = \tau_2$.*

En d'autres termes, la substitution identité est solution d'une contrainte trivialement satisfaite.

Définition 8.14 (Ensemble trivialement satisfait) *On étend simplement la définition précédente à un ensemble C de contraintes de semi-unification en disant qu'un tel ensemble est* trivialement satisfait *si chacune de ses contraintes l'est.*

En d'autres termes, un ensemble de contraintes est trivialement satisfait si la substitution identité est solution de chacune des contraintes de cet ensemble.

L'algorithme

Soit un ensemble C de contraintes de semi-unification. L'algorithme SU résolvant cet ensemble retourne, ou bien le semi-unificateur principal de cet ensemble, ou bien ERREUR, et est défini de la façon suivante :

$SU(C) =$
    `si` C est trivialement satisfait `alors retourner` id
    `sinon`
        `soit` $\tau_1 \overset{?}{\leq}{}^m_n \tau_2$ une contrainte non trivialement satisfaite de l'ensemble C
        `prendre` une instance triviale $\rho(\tau_1)$ de niveau de généralisation m
                au niveau courant n
        `soit` $\mu = \mathtt{mgu}(\rho(\tau_1), \tau_2)$
        `si` μ est indéfini, `alors retourner` ERREUR
        `sinon`
            `soit` $\mu' = SU(\mu(C))$
            `si` μ' est ERREUR, `alors retourner` ERREUR
            `sinon retourner` $\mu' \circ \mu$

Correction de l'algorithme de résolution

Comme mentionné précédemment, nous allons ici nous concentrer sur la résolution d'un ensemble comportant uniquement des contraintes de semi-unification, les contraintes d'unification ayant été préalablement résolues.

Avant d'examiner la preuve de la correction de SU, nous préférons présenter un schéma informel, montrant de manière intuitive sur un exemple, en quoi l'algorithme défini ci-dessus calcule bien une solution de l'ensemble de contraintes initial.

Soit C un ensemble de contraintes de semi-unification. Nous considérons pour l'exemple que l'algorithme va devoir « tourner » trois fois avant de trouver la solution du système C. Le schéma suivant récapitule le raisonnement permettant de se persuader que la substitution retournée par SU est finalement solution de C :

$$\begin{array}{ccccccc}
\text{non trivialement} & & \text{non trivialement} & & \text{non trivialement} & & \text{trivialement}\\
\text{résolu} & & \text{résolu} & & \text{résolu} & & \text{résolu}\\
\overbrace{C} & \xrightarrow{\mu_0} & \overbrace{\mu_0(C)} & \xrightarrow{\mu_1} & \overbrace{\mu_1 \circ \mu_0(C)} & \xrightarrow{\mu_2} & \overbrace{\mu_2 \circ \mu_1 \circ \mu_0(C)}\\
 & & & & & & \Downarrow\\
\mu_2 \circ \mu_1 \circ \mu_0 & \Longleftarrow & \mu_2 \circ \mu_1 & \Longleftarrow & \mu_2 & \Longleftarrow & \text{donc } id\\
\text{est solution de} & & \text{est solution de} & & \text{est solution de} & & \text{est solution de}\\
C & & \mu_0(C) & & \mu_1 \circ \mu_0(C) & & \mu_2 \circ \mu_1 \circ \mu_0(C)
\end{array}$$

Théorème 8.11 (L'algorithme SU **est correct)** *Soit I un ensemble de contraintes de semi-unification. Si $\varphi = SU(I)$ est définie, alors φ est solution de I.*

Preuve : La preuve est effectuée par induction sur le nombre d'itérations effectuées par l'algorithme.

Puisque par hypothèse SU est défini, il termine sans erreur au bout d'un nombre d'itérations fini. Si l'ensemble I est trivialement satisfait, alors SU retourne id et cette dernière est bien solution de I. Si l'ensemble I n'est pas trivialement satisfait, alors le nombre d'étapes global de l'algorithme étant borné, le nombre qui lui en reste à effectuer est strictement inférieur. Nous pouvons donc appliquer l'hypothèse de récurrence sur l'ensemble $\mu(I)$ et en déduire que SU est défini et nous retourne μ' solution de $\mu(I)$. Le résultat de l'itération courante est alors $\mu' \circ \mu$ qui est bien solution de I par le théorème de composition de solutions (8.2), comme nous l'attendions. ☐

Principalité et « complétude faible » de l'algorithme de résolution

Théorème 8.12 (L'algorithme SU **est principal et faiblement complet)** *Soit I un ensemble de contraintes de semi-unification. Soit φ une solution de I. Si $SU(I)$ est défini alors :*

- *$SU(I)$ ne termine pas sur* ERREUR
- *De plus, nommons $\mu = SU(I)$, alors il existe une substitution ψ telle que $\varphi(\alpha) = \psi(\mu(\alpha))$ pour toute variable de type α libre dans I (autrement dit, μ est solution principale de I).*

Preuve : On commence par effectuer la preuve sur une itération de l'algorithme. Nous en déduisons ainsi un lemme établissant la principalité de la solution pour une étape. Ensuite, nous allons inductivement l'appliquer « autant de fois » que l'algorithme SU fait d'étapes. Nous voulons montrer que si φ est solution de $\tau_1 \lneqq^m_n \tau_2$ et $\rho(\tau_1)$ est une instance triviale de τ_1, alors :

1. $\mu = \mathtt{mgu}((\rho(\tau_1), \tau_2) \neq$ ERREUR
2. Il existe ψ telle que $\varphi(\alpha) = \psi \circ \mu(\alpha)$ pour toute variable α libre (apparaissant) dans I.

• **Montrons que** $\rho(\tau_1)$ **et** τ_2 **sont unifiables.**

Puisque par hypothèse, φ est une solution de $\tau_1 \lneqq^m_n \tau_2$, nous savons qu'il existe une substitution θ telle que :

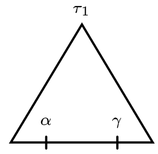
$$\theta \circ \varphi(\tau_1) = \varphi(\tau_2) \tag{8.1}$$
$$\mathrm{Dom}\,(\theta) \subseteq \mathcal{V}^{\geq m+1}(\varphi(\tau_1)) \text{ et } \mathrm{Rng}\,(\theta) \subseteq \tau^{\leq n} \tag{8.2}$$

Construisons une substitution ξ telle que :

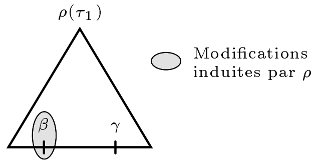
$$\xi \circ \rho(\tau_1) = \theta \circ \varphi(\tau_1) \tag{8.3}$$
$$\xi(\tau_2) = \varphi(\tau_2) \tag{8.4}$$

Nous aurons alors $\xi \circ \rho(\tau_1) = \xi(\tau_2)$ grâce à 8.1 ce qui signifiera que ξ est bien un unificateur de $\rho(\tau_1)$ et τ_2. Illustrons l'idée de la construction de notre substitution ξ grâce aux schémas suivants.

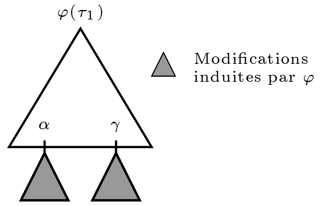
Considérons un type τ_1 qui comporte des variables α généralisées ainsi que des variables γ ne l'étant pas :



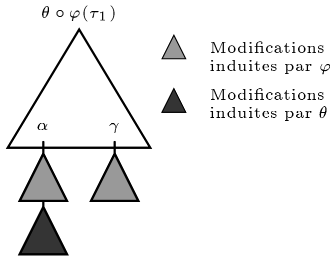
Par définition de la prise d'instance triviale, le type $\rho(\tau_1)$ possède donc la forme suivante, dans laquelle les variables α de τ_1 ont été remplacées par des variables fraîches β :



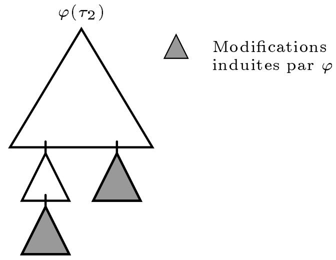
Puisque nous supposons l'existence d'une substitution φ, celle-ci peut modifier τ_1, par exemple de la façon suivante :



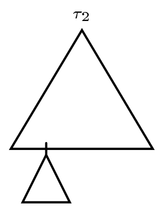
De plus, puisque nous supposons que cette substitution φ est solution de $\tau_1 \lesssim^m_n \tau_2$, cela signifie, par définition, qu'il existe une substitution θ telle que $\theta \circ \varphi(\tau_1) = \varphi(\tau_2)$. Donc cette substitution θ peut modifier $\varphi(\tau_1)$ de la façon suivante :



Considérons maintenant un type τ_2 dont nous aimerions intuiter la forme. Puisque $\theta \circ \varphi(\tau_1) = \varphi(\tau_2)$ et que, par ci-dessus, nous connaissons la forme de $\theta \circ \varphi(\tau_1)$, nous déduisons que $\varphi(\tau_2)$ doit avoir la forme suivante, dans laquelle nous avons choisi un exemple de modifications que φ apporte par rapport à la forme de τ_2 :



Ainsi, du choix effectué ci-dessus pour l'effet de φ sur τ_2, nous déduisons que ce dernier doit avoir la forme :



Reprenons la construction de notre substitution ξ. Pour satisfaire la partie 8.3, il faut que les β de $\rho(\tau_1)$ soient envoyés sur $\theta\varphi(\alpha)$. Pour que la partie 8.4 le soit également, il nous faut prendre ξ égale à φ. Ainsi, nous choisissons notre substitution par :

$$\xi = \varphi \oplus \{\beta \leftarrow \theta \circ \varphi(\alpha)\}$$

Vérifions maintenant que ξ vérifie bien les propriétés que l'on attendait d'elle (à savoir 8.3 et 8.4) :

- Dans τ_1 il existe :
  - des variables α de niveau $> m$ (généralisées) qui sont transformées comme suit : $\alpha \overset{\rho}{\mapsto} \beta \overset{\xi}{\mapsto} \theta(\varphi(\alpha))$. Ainsi nous avons bien $\xi \circ \rho(\tau_1) = \theta \circ \varphi(\tau_1)$ comme attendu en 8.3.
  - des variables γ de niveau $\leq m$ (non généralisées) qui sont transformées comme suit : $\gamma \overset{\rho}{\mapsto} \gamma \overset{\xi}{\mapsto} \varphi(\gamma)$. Or par 8.2, nous savons que θ n'affecte que les variables de niveau $\geq m+1$. Les variables γ étant de niveau $\leq m$ et φ préservant ces niveaux, $\varphi(\gamma) = \theta(\varphi(\gamma)$ et nous avons bien aussi $\xi \circ \rho(\tau_1) = \theta \circ \varphi(\tau_1)$ comme attendu en 8.3.
- Puisque τ_2 ne contient pas de β (variables fraîches issues de la prise d'instance triviale), nous avons que $\xi(\tau_2) = \varphi(\tau_2)$ comme attendu en 8.4.

Ainsi $\xi = \varphi \oplus \{\beta \leftarrow \theta \circ \varphi(\alpha)\}$ est bien un unificateur de $\rho(\tau_1)$ et τ_2, donc ces deux types sont unifiables, et il existe donc un unificateur principal μ que l'algorithme `mgu` trouve (par complétude).

• Montrons qu'il existe ψ telle que $\varphi(\alpha) = \psi \circ \mu(\alpha)$ pour toute variable α libre (apparaissant) dans I.

Puisque est l'unificateur principal de $\rho(\tau_1)$ et τ_2, on peut écrire $\xi = \eta \circ \mu$. Si l'on prend une variable α libre dans I (donc différente des β introduites par la prise d'instance triviale), nous avons $\xi(\alpha) = \varphi(\alpha)$ et $\xi(\alpha) = \eta \circ \mu(\alpha)$, donc $\varphi(\alpha) = \psi \circ \mu(\alpha)$ comme nous attendions. Ainsi, choisir $\psi = \eta$ convient.

Maintenant que nous avons montré qu'à chaque étape définie, l'algorithme SU était principal et faiblement complet, il suffit d'appliquer ce lemme par récurrence sur le nombre d'itérations effectuées par l'algorithme lors d'une exécution complète. L'utilisation de la récurrence est applicable car $SU(I)$ étant considéré comme défini, il en découle que ce nombre d'itérations est fini. Le cas de base est traité trivialement car si I est trivialement satisfait, alors la substitution renvoyée est id qui est clairement principale.

$\square$

8.4.8 Un exemple pas à pas

Nous allons ici résoudre « à la main » un problème de typage simple afin de « voir tourner » l'algorithme. Soit la phrase : `let rec f x = f true ; f 1 in f` pour laquelle nous attendons le type $\alpha \to \beta$. Générons le système de contraintes, nous obtenons l'ensemble suivant :

$$\left\{ \begin{array}{lll} {}'b_{(1)} \stackrel{?}{=}_1 {}'c_{(1)} \to {}'d_{(1)}\,; & {}'b_{(1)} \stackrel{?}{\leq}{}^0_1 {}'e_{(1)}\,; & {}'e_{(1)} \stackrel{?}{=}_1 \texttt{bool} \to {}'f_{(1)}\,; \\ {}'b_{(1)} \stackrel{?}{\leq}{}^0_1 {}'g_{(1)}\,; & {}'g_{(1)} \stackrel{?}{=}_1 \texttt{int} \to {}'d_{(1)}\,; & {}'c_{(1)} \to {}'d_{(1)} \stackrel{?}{\leq}{}^0_0 {}'a_{(0)} \end{array} \right\}$$

et le type résultat : ${}'a_{(0)}$

Maintenant, résolvons ce système. Nous partons donc du système initial ci-dessus dans lequel nous commençons par résoudre les contraintes d'unification. En particulier, nous choisissons de nous intéresser à celle étiquetée par un ▶ :

$$\begin{array}{llll} \blacktriangleright & {}'b_{(1)} \stackrel{?}{=}_1 {}'c_{(1)} \to {}'d_{(1)} & {}'b_{(1)} \stackrel{?}{\leq}{}^0_1 {}'e_{(1)} & {}'e_{(1)} \stackrel{?}{=}_1 \texttt{bool} \to {}'f_{(1)} \\ & {}'b_{(1)} \stackrel{?}{\leq}{}^0_1 {}'g_{(1)} & {}'g_{(1)} \stackrel{?}{=}_1 \texttt{int} \to {}'d_{(1)} & {}'c_{(1)} \to {}'d_{(1)} \stackrel{?}{\leq}{}^0_0 {}'a_{(0)} \end{array}$$

Les variables de types étant toutes de niveau 1 (donc les types en eux-mêmes), nous pouvons choisir une substitution (qui respectera bien les niveaux) unifiant ${}'b_{(1)}$ et ${}'c_{(1)} \to {}'d_{(1)}$, donc remplacer dans le système toutes les occurrences de ${}'b_{(1)}$ par ${}'c_{(1)} \to {}'d_{(1)}$, puis supprimer de l'ensemble la contrainte que nous venons de traiter. On obtient alors le système suivant :

$$\begin{array}{llll} {}'c_{(1)} \to {}'d_{(1)} \stackrel{?}{\leq}{}^0_1 {}'e_{(1)} & \blacktriangleright & {}'e_{(1)} \stackrel{?}{=}_1 \texttt{bool} \to {}'f_{(1)} & {}'c_{(1)} \to {}'d_{(1)} \stackrel{?}{\leq}{}^0_1 {}'g_{(1)} \\ {}'g_{(1)} \stackrel{?}{=}_1 \texttt{int} \to {}'d_{(1)} & & {}'c_{(1)} \to {}'d_{(1)} \stackrel{?}{\leq}{}^0_0 {}'a_{(0)} & \end{array}$$

Continuons à traiter les contraintes d'unification restant. Nous choisissons la suivante dans l'ensemble. De la même façon que ci-dessus, nous pouvons trouver une substitution respectant les niveaux pour cette contrainte et remplacer les occurrences de $'e_{(1)}$ par $\texttt{bool} \to {'f_{(1)}}$. Le nouvel ensemble de contraintes prend donc la forme suivante :

$$'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{1} \texttt{bool} \to {'f_{(1)}} \qquad 'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{1} {'g_{(1)}} \quad \blacktriangleright \quad 'g_{(1)} \stackrel{?}{=}_{1} \texttt{int} \to {'d_{(1)}}$$
$$'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{0} {'a_{(0)}}$$

Nous résolvons la dernière contrainte d'unification pour obtenir :

$$'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{1} \texttt{bool} \to {'f_{(1)}} \quad 'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{1} \texttt{int} \to {'d_{(1)}} \quad 'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{0} {'a_{(0)}}$$

L'ensemble de contraintes que nous obtenons est ainsi uniquement composé de contraintes de semi-unification. Nous allons donc lui appliquer l'algorithme de résolution vu précédemment. Nous recherchons donc une contrainte non satisfaite. Clairement les deux premières le sont, mais pas la troisième. Nous prenons donc une instance de $'c_{(1)} \to {'d_{(1)}}$, mais au niveau 0 (indice de $\stackrel{?}{\leq}{}^{0}_{0}$) et en considérant que les variables de niveau supérieur strictement à 0 (exposant de $\stackrel{?}{\leq}{}^{0}_{0}$) sont généralisées et donc doivent être remplacées par des variables fraîches. Nous obtenons donc un type $'c_{(0)} \to {'d_{(0)}}$ que nous devons unifier avec $'a_{(0)}$. Il s'ensuit que ce dernier devient égal à $'c_{(0)} \to {'d_{(0)}}$ dans tout le problème, et en particulier dans le type du résultat de notre expression qui, rappelons-le, était justement $'a_{(0)}$. Nous pouvons donc maintenant supprimer cette contrainte dans notre ensemble et il nous reste :

$$'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{1} \texttt{bool} \to {'f_{(1)}} \quad 'c_{(1)} \to {'d_{(1)}} \stackrel{?}{\leq}{}^{0}_{1} \texttt{int} \to {'d_{(1)}}$$

dont toutes les contraintes sont satisfaites. L'algorithme s'arrête donc et le type de notre expression est $'c_{(0)} \to {'d_{(0)}}$, comme attendu (à un renommage près des variables).

8.4.9 Le test d'arrêt

Nous avons vu qu'en raison de l'indécidabilité de l'inférence de types en présence de récursion polymorphe, nous ne pouvions espérer un algorithme que sous réserve de restreindre l'ensemble des programmes que nous acceptons de typer. Ainsi certains programmes, pourtant corrects vis-à-vis des règles d'évaluation, seront rejetés par notre algorithme d'inférence de types.

Nous avons également vu que la garantie de terminaison de notre algorithme est due à l'adjonction d'un test d'arrêt décidant arbitrairement quand il faut considérer que la recherche échoue, c'est-à-dire quand il faut considérer qu'il y a risque de poursuivre indéfiniment la recherche d'un type pour l'expression.

Nous allons maintenant décrire le fonctionnement de ce test d'arrêt. Tout d'abord, examinons les différents cas pouvant mener à une recherche infinie, en raisonnant sur des schémas de types (plutôt que sur des types avec niveaux, juste pour une question de simplicité). Soit un identificateur `x` lié récursivement par une construction `let rec`. Soit σ son schéma de types. À chaque étape de l'itération :

- des nouvelles variables généralisées peuvent apparaître. Par exemple, si $\sigma = \forall\alpha.\,\alpha$, ce dernier peut évoluer sous la forme $\sigma_1 = \forall\alpha,\beta.\,\alpha \to \beta$, puis $\sigma_2 = \forall\alpha,\beta,\gamma.\,\alpha \to \beta \to \gamma$ etc…
- le nombre de variables généralisées peut rester stable, mais la structure du schéma peut s'accroître indéfiniment. Par exemple, si $\sigma = \forall\alpha.\,\alpha$, ce dernier peut évoluer sous la forme $\sigma_1 = \forall\alpha.\,\alpha \to \alpha$, puis $\sigma_2 = \forall\alpha.\,\alpha \to \alpha \to \alpha$ etc…

Définition 8.15 (Test d'arrêt primaire) *Soit un ensemble de contraintes C. L'exécution de $SU(C)$ s'arrête :*
- *si l'ensemble C est trivialement satisfait*
- *ou si le nombre de variables généralisées présentes dans C n'a pas* strictement *décru.*

Le nombre de variables généralisées présentes dans l'ensemble C initial étant fini, la décroissance demandée ne pourra certes pas se prolonger *ad vitam aeternam*. Néanmoins ce test pose un nouveau problème.

L'inférence étant initiée avec des schémas *a priori* trop généraux pour chacun des identificateurs définis récursivement (schémas se trouvant encodés dans l'ensemble de contraintes C initial), rien ne nous garantit que le nombre de variables généralisables présentes initialement dans C fournira assez de polymorphisme pour trouver une solution au problème. Intuitivement, supposons que tous les identificateurs du problème soient liés à des schémas de types de la forme $\forall\alpha.\, \alpha$, il semble probable qu'afin de leur trouver un type correct il faille rajouter des variables généralisées (en effet seule les expressions ne terminant pas ont, en ML, un tel schéma de types). Or notre test d'arrêt interdira de continuer la recherche si de telles variables sont rajoutées. Nous contournons ce problème en ajoutant une condition supplémentaire à notre test d'arrêt : nous autorisons chaque identificateur du problème à voir son schéma de types associé grossir *1 seule* fois, séparément des schémas des autres identificateurs. Cette augmentation du nombre de variables peut de plus s'effectuer à tout moment au cours des itérations de l'algorithme SU (c'est-à-dire par forcément à la première ou à la seconde itération).

Définition 8.16 (Schéma de types saturé) *Un schéma de types σ est dit saturé si son nombre de variables généralisées a augmenté 1 fois par rapport aux itérations précédentes de l'algorithme SU.*

Intuitivement un schéma est saturé s'il a « grossi » une fois, c'est-à-dire si sa structure comporte plus de variables généralisées qu'elle n'en avait à l'étape précédente de l'algorithme. Cela revient à dire que ce schéma a atteint la limite au-delà de laquelle SU considérera qu'il n'y a pas de solution au problème.

Munis de ce nouveau raffinement, nous pouvons énoncer un test d'arrêt plus précis et plus fonctionnel que le précédent.

Définition 8.17 (Test d'arrêt relaxé) *Soit un ensemble de contraintes C. L'exécution de $SU(C)$ s'arrête :*
- *si l'ensemble C est trivialement satisfait*
- *ou si le nombre de variables généralisées présentes dans C n'a pas* strictement *décru et le nombre de schémas de types saturés n'a pas strictement augmenté.*

Le nombre de nombre d'identificateurs liés dans le problème étant fini, l'augmentation du nombre de schémas saturés associés ne peut également se prolonger indéfiniment. Le choix d'autoriser les schémas à « grossir » une seule fois est arbitraire et justifié par la constatation que *zéro fois* est impossible (puisque ça ne change rien au problème soulevé) et que *une fois* permet d'obtenir des résultats très satisfaisants dans la pratique.

8.4.10 Implémenter l'inférence

Maintenant que nous avons présenté de façon théorique le fonctionnement de notre algorithme d'inférence de types en présence de récursion polymorphe, nous allons succinctement examiner comment implémenter un tel algorithme dans un typeur réel ainsi que les problèmes qui se posent pour ce genre d'implantation.

Dans ce qui suit, lorsque nous parlerons d'une *définition* nous référerons à une construction de la forme `let (rec) id =` exp_1 `in` exp_2, dans laquelle nous appellerons *corps*, l'expression exp_1 et *partie in*, l'expression exp_2.

L'algorithme que nous avons décrit est modélisé en terme de synthèse d'un ensemble explicite de contraintes qui sont ensuite résolues. Dans un typeur ML les contraintes ne sont pas explicitement mémorisées, mais résolues au fur et à mesure par un mécanisme d'unification en place, en une seule passe sur l'arbre de syntaxe de l'expression à typer. La mémorisation des « schémas de types » (nous devrions plutôt dire des « types représentant des schémas » puisque nous sommes dans le cadre d'un typeur avec niveaux et sans schémas ; néanmoins nous avons toujours une distinction de principe entre un *type* – lié à une expression – et un *type ayant été généralisé* – c'est-à-dire auquel nous avons appliqué la règle Gen avant le mettre dans l'environnement en le liant à un identificateur–) associés aux identificateurs est gérée par un mécanisme d'environnement que l'on peut voir comme une liste d'association (identificateur $\times$ schéma de types). À chaque fois que l'on désire utiliser un identificateur, une recherche est effectuée dans cet environnement et une instance de son « schéma de types » est prise par remplacement des variables marquées généralisées du corps du schéma par des nouvelles variables. De même, afin de procurer le polymorphisme nécessaire à ML, le type inféré pour un identificateur lié par `let (rec)` est transformé en schéma de types par marquage de ses variables généralisées, avant d'être inséré dans l'environnement.

L'ajout de la récursion polymorphe dans un tel cadre peut être décrite par l'algorithme d'inférence schématique qui suit pour le cas des constructions `let` et `let rec` (le typage des autres constructions, hormis delui des identificateurs qui est décrit ci-dessous, ne changent pas) :

1 • Typer une fois pour toutes les corps des définitions,
2 • Généraliser les types obtenus pour obtenir des schémas et les
    mémoriser dans l'environnement de typage,
3 • Construire un « problème-μ » qui reflète les contraintes
    de semi-unification portant sur la phrase,
4 • Typer la suite de la phrase,
5 • Vérifier que le type de chaque occurrence d'un identificateur
    présente le corps des définitions récursives de la phrase est
    une instance du schéma associé obtenu jusqu'à présent,
6 • Déduire de nouveaux schémas à partir des nouveaux types des
    expressions définissantes par généralisation,
7 • Vérifier que toutes les occurrences d'identificateurs présentes
    dans la partie `in` de la définition sont des instances des
    nouveaux schémas respectifs ainsi générés,
8 • Vérifier une condition d'arrêt et l'atteinte du point fixe,
9 • S'il faut continuer alors reprendre au point 5,
  s'il y a échec alors traiter cet échec,
  sinon les schémas obtenus au point 7 sont solution de notre
    problème de typage.

Cette description appelle plusieurs remarques. En premier lieu, lors de la première phase de l'algorithme, c'est à dire celle qui s'occupe de typer le corps des définitions, un certain nombre d'informations vont être emmagasinées afin de pouvoir constituer ce que nous appelons après un « problème-μ ». Lorsque nous disons « typer le corps des définitions », nous entendons simplement appliquer l'algorithme d'inférence sur celles-ci. Nous nous rendre compte que cet algorithme est le même qu'en ML habituel, à l'accès près à l'environnement de typage (cf. 8.4.10 ci-dessous) et au typage des `let / let rec` que nous sommes en train de décrire.

Les vérifications effectuées aux points 5 et 7 de l'algorithme peuvent être simplement réalisées à l'aide des deux primitives d'unification entre types et d'instanciation de schémas de types. Ces deux opérations faisant partie du noyau de tout typeur ML, nous voyons ainsi que ceci ne requiert aucun travail d'implémentation supplémentaire au niveau du compilateur. De même, le point 6 est le résultat d'une généralisation de types via le mécanisme existant dans le typeur d'origine.

La huitième étape de notre algorithme fait apparaître une notion de test d'arrêt comme celle que nous avons étudiée précédemment. Toutefois, une légère modification sera apportée au mécanisme d'origine comme nous le verrons ultérieurement.

Enfin, nous avons expliqué précédemment que notre algorithme pouvait effectuer des unifications aux vol, éventuellement préalablement à la résolution des contraintes de semi-unification, et nous avons montré que ceci était correct. Ce phénomène se manifeste ici le fait que nous commençons par typer (ce qui met en œuvre des unifications) le corps des définitions avant même de rechercher des schémas de types corrects à lier aux identificateurs définis récursivement. Nous noterons également que d'autre unifications peuvent encore se produire lors des phases 4, 5 et 7.

La mémorisation des contraintes de semi-unification

Comme nous venons de le dire, lors de la première phase du typage, les contraintes de semi-unification portant sur le programme doivent être mémorisées. Ces contraintes spécifient plus précisément que le type de chaque occurrence d'un identificateur défini récursivement doit être une instance du schéma trouvé pour l'expression définissant cet identificateur. Il convient donc, lors de chaque utilisation d'un identificateur, de mémoriser le type qu'on lui a attribué, ainsi que le niveau de généralisation auquel on a créé ce type (c'est-à-dire le niveau courant auquel on a typé l'expression où apparaissait cet identificateur). Intuitivement, ce niveau correspond au m que l'on trouve dans notre formalisme $\stackrel{?}{\leq}{}^{m}_{n}$.

Nous devons donc légèrement modifier la structure des environnements de typage. Dans le système de ML sans récursion polymorphe les environnements sont des fonctions des identificateurs vers les schémas (plus intuitivement des listes d'associations composées de couples (identificateur $\times$ schéma de types)). Dorénavant, les environnements devront également mémoriser la liste des types de chaque occurrence des identificateurs définis récursivement.

Notons qu'une fois que la phrase a été typée, il n'est plus nécessaire de conserver cette dernière information et l'on peut repasser dans une représentation habituelle des environnements à la ML.

Nous pouvons donc décrire cette structure par la structure de données ML suivante :

```
type binding =
  | Regular of nom * types_scheme
  | Mu of nom * types_scheme * occurrence list ref
type environment = binding list
```

où une liaison de type `Regular` sert à enregistrer un schéma définitif associé à un identificateur, c'est-à-dire celui trouvé une fois que l'algorithme a réussi à inférer un schéma de types correct. Une liaison de type `Mu` sert à enregistrer un schéma de type provisoire jusqu'à ce que la résolution du problème de typage nous ait donné le bon schéma à lier à l'identificateur ; il conviendra alors de remplacer dans l'environnement le lien de cet identificateur avec un `Mu` par un `Regular`. Autrement dit, un environnement construit après qu'un problème de typage ait été complètement résolu ne contient plus aucune liaison de type `Mu`.

Munis de cette structure d'environnement, nous pouvons esquisser la nouvelle forme de l'opération d'accès à l'environnement :

```
let rec get_ident_type id_name = function
 | [] → raise Not_found
 | Mu (n, scheme, occs) :: rem →
     if n = id_name then
       begin
       let t' = specialize scheme in
       (* Mémoriser l'occurrence *)
       occs := { occ_level = !current_binding_level ; occ_type = t' } :: !occs ;
```

```
       t'
       end
     else get_ident_type id_name rem
 | Regular (n, t) :: rem →
     (* Comme en ML conventionnel *)
     if n = id_name then specialize t else get_ident_type id_name rem
```

Mémoriser ces informations dans l'environnement ne suffit pas à représenter les contraintes de semi-unification pesant sur notre programme. Une fois que le corps des expressions définissantes des identificateurs a été typé, nous devons construire ce que nous avons appelé un « problème-μ », qui va se charger de cette mémorisation.

Qu'est-ce qu'un « problème-μ » ?

Un « problème-μ » est une liste de « sous-problèmes-μ ». Il existe un « sous-problème-μ » par définition. Par exemple, `let rec` x_0 `= ...and` x_1 `= ...and` x_3 `= ... in expr` constituera potentiellement un « problème-μ » composé de trois « sous-problèmes-μ ».

Un « sous-problème-μ » est chargé de mémoriser les informations relatives à un identificateur défini récursivement. Il contient le nom de cet identificateur, le schéma trouvé à chaque étape de l'algorithme pour cet identificateur, la liste des occurrences de cet identificateur apparues dans le corps de sa définition, et celles apparues dans la partie `in` du `let rec ...and ...`, une information de saturation et enfin le type trouvé pour le corps de l'expression définissant cet identificateur. Ceci peut être représenté par la structure de donnée ML suivante :

```
type mu_definition = {
  (* Nom de l'identificateur lié par let ou let rec *)
  mu_name : string ;
  (* Schéma d'accumulation *)
  mu_scheme : types_scheme ;
  (* Type trouvé pour le corps de l'expression définissante *)
  mu_type : simple_type ;
  (* Occurrences de cet identificateur dans l'expression définissante seulement *)
  mu_body_occs : occurrence list ;
  (* Occurrences de cet identificateur dans la partie ``in'' *)
  mu_in_occs : occurrence list ref ;
  (* Compteur d'expansions *)
  mu_saturation : int
 }
type mu_sub_problem = {
  (* Niveau auquel se trouve le "let" qui donne lieu à ce sous-problème *)
  mu_sub_level : int ;
  (* Ensemble des sous-problèmes, du let (rec) and *)
  mu_sub_defs : mu_definition list
  }
type mu_problem = mu_sub_problem list
```

Comment construire un « problème-μ » ?

Trivialement, si les définitions traitées ne contiennent aucune récursivité, il est inutile de chercher à résoudre, ni même de créer de « problème-μ ». Plus précisément, puisque les identificateurs définis n'apparaissent pas dans le corps des définitions, le fait qu'on leur ait assigné un type trop général n'a aucune importance puisque ce schéma n'a pas été utilisé pour typer le corps.

Dans le cas où les définitions sont récursives, il se peut, pour chaque schéma obtenu après le typage des expressions définissantes, qu'il ne contienne pas de variables généralisées. Il est alors inutile de l'ajouter au « problème-μ » (ce qui signifie qu'il ne contribuera ainsi plus à augmenter le temps de calcul de la solution du problème de typage). Dans le cas contraire, il est nécessaire de l'ajouter au « problème-*mu* » global sous forme d'un nouveau « sous-problème-μ », toutes les informations nécessaires à la construction des structures décrites précédemment nous étant désormais disponibles (en particulier le schéma de type initial, obtenu par généralisation du type initial trouvé pour l'expression définissante de l'identificateur).

Remarquons qu'initialement le compteur de saturation est nul et la liste des occurrences de l'identificateur en *partie in* est vide puisque nous n'avons encore typé que le corps de la définition.

Une fois le problème-μ encapsulant les contraintes de semi-unification construit, nous étendons simplement l'environnement de typage afin de typer la *partie in* de la définition, tout comme cela est réalisé en ML standard. Remarquons que les accumulations d'informations sur les occurrences d'identificateurs définis récursivement sont alors faites de façon « transparente » lors des accès à l'environnement.

Une fois le typage effectué, l'algorithme va pouvoir entamer sa recherche des schémas de types corrects comme esquissé dans la description précédente.

Une modification du test d'arrêt

Nous avons vu précédemment que nous demandions une décroissance stricte du nombre de variables généralisées du problème ou une augmentation stricte du nombre de schémas saturés pour continuer la recherche d'une solution, l'arrêt étant obtenu si l'ensemble de contraintes se trouve trivialement satisfait.

Dans l'implémentation, le système de contraintes n'étant pas explicitement généré, nous ne testons pas la satisfaction triviale directement sur ce dernier. À la place, partant du principe que ce dernier est trivialement résolu si la substitution *id* est solution du système et que l'application des substitutions est fait en place dans notre implémentation, nous vérifions plus simplement que les schémas obtenus ont atteint un *point fixe*, c'est-à-dire qu'ils sont restés identiques par rapport à l'itération précédente.

Quelques remarques sur l'algorithme schématique

Nous énonçons ici quelques remarques à propos des définitions au fur et à mesure de leurs évolutions lors que l'algorithme tourne.

Définition 8.18 (Schéma de corps) *σ est un schéma de corps si toutes les occurrences de l'identificateur auquel il est lié présentes dans le corps de la définition sont des instances de σ.*

Définition 8.19 (Schéma de `in`) *σ est un schéma de* `in` *si toutes les occurrences présentes en partie* `in` *de la définition correspondante sont des instances de σ.*

Définition 8.20 (Schéma d'expression définissante) *σ est un schéma d'expression définissante si et seulement si le type de l'expression définissante est une instance de σ.*

Définition 8.21 (Schéma compatible) *σ est un schéma compatible lorsqu'il est à la fois schéma de corps, schéma de* `in` *et schéma d'expression définissante.*

On se convaincra aisément que notre algorithme possède les propriétés suivantes :

- **à l'entrée de l'algorithme :** après l'étape 5, tous les schémas du « problème-μ » d'entrée sont *de corps*. Si n est le nombre d'itérations effectuées par l'algorithme et $n \geq 1$, alors ces schémas sont *compatibles* (car *de corps* et *de* `in` puisqu'ils étaient schémas de sortie (donc *schéma d'expression définissante*) de l'itération $n-1$).
- **à la sortie de l'algorithme :** tous les schémas sont *schémas de* `in`. Tous les schémas sont des *schémas d'expression définissante*.

- **si on obtient le point fixe** (c'est-à-dire les mêmes schémas en sortie qu'en entrée), alors ces schémas sont *compatibles* et chacun est la généralisation du type de l'expression définissante correspondante.

8.4.11 Le retour en mode monomorphe

Dans le cas où l'algorithme $\mathcal{C}$ échoue nous avons vu qu'il était nécessaire de retrouver un mode de typage à la ML (cf. 8.2.3). Deux solutions étaient alors possibles : soit recommencer l'inférence avec un algorithme d'inférence standard comme il en existe déjà dans tout compilateur ML, soit essayer de tirer profit du travail déjà effectué par l'inférence en mode polymorphe. Nous allons montrer ici comment il est possible de retourner dans un mode d'inférence avec `let rec` monomorphe de cette manière.

Nous devons tout d'abord unifier toutes les occurrences de corps avec le type trouvé pour l'expression définissante. En effet, cela simule le fait qu'en ML normal, le schéma introduit dans l'environnement de typage pour typer un identificateur défini récursivement dans le corps de sa définition n'est pas polymorphe.

Notons qu'il ne faut pas unifier les occurrences se trouvant dans la *partie in* car cela serait plus restrictif que ML et reviendrait à interdire le polymorphisme introduit par la construction `let`.

Il reste désormais à gérer ces dernières puisque leurs types sont encore potentiellement incorrects. Nous généralisons donc les types trouvés ci-dessus pour les expressions définissantes (ces mêmes types qui ont accumulé *par effet de bord* toutes les contraintes portant sur les occurrences récursives des identificateurs) et nous trouvons ainsi les schémas de types attendus pour chacun des identificateurs définis récursivement.

Nous devons donc vérifier que toutes les occurrences d'identificateurs présentes en *partie in* de nos définitions sont des instances de leur schéma respectif. Il est important d'effectuer ces généralisations et cette vérification en suivant la structure des environnements de typages et en prenant soin de le faire au niveau correspondant à chaque définition. Ceci permet de simuler le fait qu'en ML les instances du schéma sont prises au niveau de généralisation courant (rappelons-nous que ce niveau a été mémorisé dans le « problème-μ ») et que les contraintes de généralisation des définitions les plus internes (c'est-à-dire des `let` les plus imbriqués) doivent être prises en compte avant de continuer à typer les expressions plus externes (autrement dit, lorsque l'on type une partie `in` de définition, tout ce qui devait être généralisé dans l'expression définissante l'a été et est donc devenu polymorphe).

La possibilité de procéder en suivant la structure des environnements de typage doit nous être garantie par la construction des « problèmes-μ ». Il suffit pour cela d'emmagasiner les sous-problèmes dans l'ordre où ils apparaissent, sous forme d'une liste par exemple, et la structure d'environnement est alors donnée par parcours miroir de cette liste. En effet, considérons le programme suivant :

```
let x =
  let y =
    let z = ... in expr_a
  in expr_b
 in expr_c
```

Nous commençons par typer le corps de `let x =`, ce qui nous amène à typer `let y = ...`, donc son corps, donc `let z = ...`, puis le corps de ce dernier qui est $expr_a$. Nous déduisons un type pour cette expression et construisons un sous-problème-μ pour `let z`. Ensuite nous « remontons d'un cran » et construisons un nouveau sous-problème-μ pour `let y` puis nous typons $expr_b$. Et pour finir nous générons le dernier « sous-problème-μ », celui lié à `let x` puis nous typons $expr_c$. Nous voyons ainsi que les « sous-problème-μ » sont emmagasinés dans un ordre fixe qui peut aisément rendre compte de leur niveau d'imbrication. Notons que la structure qui en découle n'est pas exactement celle des environnements, car dans le cas où à un même niveau se trouvent plusieurs définitions (mutuellement récursives ou non, c'est-à-dire `let rec ...and...` ou `let ...and ...`), l'ordre de mémorisation n'est pas spécifié. Toutefois ceci n'a aucune importance puisque dans ce cas, soit les définitions sont récursives et inter-dépendent donc l'ordre n'a pas d'importance, soit elles ne sont pas récursives et l'ordre n'a pas non plus d'importance car cela revient à des définitions séquentielles (`let ...in let ...in`).

Lorsque ce processus est terminé, soit il y a eu un échec dû à une unification qui a échoué, ce qui dénote une erreur de type, soit les schémas obtenus sont ceux que nous aurait donné un typage ML habituel direct. Ainsi notre algorithme se comporte comme une extension stricte de l'algorithme d'inférence de ML et son ajout dans un compilateur déjà existant ne risque pas de remettre en cause les programmes déjà écrits.

8.4.12 Remarques sur l'algorithme d'inférence

Le résultat de principalité de l'inférence énoncé précédemment nous garantit que, si notre algorithme termine, alors quelque soit l'ordre dans lequel les contraintes de semi-unification sont résolues.

Par contre, dans le cas où notre algorithme échoue (et passe donc en mode monomorphe, rien ne nous garantit que résoudre ces contraintes dans un ordre différent ne nous aurait pas mené à une solution au problème.

Ceci est certes une faiblesse de notre algorithme, mais inhérent à son indécidabilité. Notons toutefois que l'algorithme proposé par [19] possède le même travers. Néanmoins, dans ce dernier, tout comme nous-mêmes, n'avons actuellement pas encore trouvé de tels cas d'échec dans la pratique.

Chapitre 9

Aspects pratiques

DANS CETTE SECTION, nous allons tout d'abord présenter quelques résultats expérimentaux obtenus avec notre implémentation. Ensuite nous verrons comment rendre les résultats de notre analyse plus aisément compréhensibles par l'utilisateur, les types bruts s'avérant relativement difficiles à lire.

9.1 Performances et exemples pratiques

La version actuelle de l'analyseur, utilisée pour illustrer cette section comporte toutes les extensions décrites en section 7 à l'exception des objets qui ont été prototypés à part et demeurent à fusionner définitivement. Ces extensions incluent donc l'analyse du langage de base d'Objective Caml complet (toutes les constructions syntaxiques, séquence, conditionnelles, boucles, enregistrements, références, etc...), l'analyse du langage de modules (structures et foncteurs), la prise en compte des définitions de types utilisateur et celle des structures de données mutables.

D'autre part, l'analyseur comporte également une inférence de types en présence de récursion polymorphe comme décrite en section 8. Son implémentation est une adaptation de l'algorithme que nous avons présenté, adaptation relativement aisée puisque, comme nous le souhaitions lors de la conception de l'algorithme de récursion polymorphe, celui-ci devait être aisément adaptable à d'autres algèbres de types que celle de ML. En ce sens, nous pensons qu'il a correctement rempli son objectif sur ce point.

Dans ce qui suit, notre analyseur est compilé avec Objective Caml 2.01 en mode de génération de code natif et fonctionne sur une machine à base d'un Pentium II à 333 Mhz comportant 128 Mo de mémoire vive et tournant sous Linux.

9.1.1 Vitesse de l'analyse

Le tableau ci-dessous montre les performances de l'analyseur obtenues sur des programmes écrits en Objective Caml dont les tailles s'échelonnent du petit programme au programme de taille moyenne. Nous donnons ici les résultats avec et sans utilisation de l'inférence de types avec récursion polymorphe. De même, afin de faciliter la comparaison entre le typage conventionnel d'un programme et celui nécessité par notre analyse, nous montrons les temps nécessaires au compilateur Objective Caml pour typer ces mêmes programmes. Tous les temps mentionnés incluent la phase de lecture, d'analyse syntaxique et de typage des programmes.

Nous pouvons remarquer que les performances globales sont relativement bonnes, dans l'ordre de grandeur de 1000 à 2000 lignes de source analysé par seconde. Les programmes

Programme	Taille (lignes)	Temps d'analyse	Vitesse d'analyse (lignes par sec.)	Temps de typage (Ocaml)
1. Compression Huffman	233	0.07/0.08 s	3300/2900 l/s	0.08 s
2. Knuth-Bendix	441	0.14/0.16 s	3200/2800 l/s	0.14 s
3. Docteur (Eliza)	556	0.81/0.83 s	680/670 l/s	0.10 s
4. Générateur de lexer	1169	0.27/0.32 s	4300/3700 l/s	0.20 s
5. Nucléic	2919	1.90/1.88 s	1530/1550 l/s	0.62 s
6. Bibliothèque std. OCaml	3082	2.52/2.52 s	1200/1200 l/s	1.89 s
7. Analyseur de fichiers `.h`	3088	0.54/0.58 s	5700/5300 l/s	0.27 s
8. Notre analyseur d'exceptions	12235	10.3/16.1 s	1200/760 l/s	3.86 s
9. Compilateur Ocaml (bytecode)	17439	12.6/22.9 s	1400/760 l/s	4.00 s

FIG. 9.1 – Résultats expérimentaux (avec /sans récursion polymorphe)

contenant de grandes structures de données fournies en extension (Nucléic, Docteur) nécessitent un peu plus de temps pour être analysés du fait de la grande taille des rangées annotant les types de ces structures de données.

En moyenne, l'analyse d'un programme nécessite deux fois plus de temps que son typage par le compilateur Objective Caml bien que les rapports puissent varier entre 1 (pour les programmes simples) et 8 (pour des programmes comportant de grandes structures de données en extension, comme par exemple « Docteur »).

L'ajout de récursion polymorphe augmente le temps d'analyse d'environ un facteur 1.5 sur le test 8 et d'un facteur 1.8 sur le test 9, mais n'a pas d'impact significatif sur les autres tests. Il est intéressant de noter que le ralentissement provoqué reste très acceptable en comparaison de l'amélioration de précision obtenue.

9.1.2 Précision de l'analyse

Globalement, les exceptions non rattrapées mentionnées par notre analyseur se classent en quatre catégories :

- Les *positives vraies* : ce sont les exceptions qui peuvent effectivement s'échapper et mettre terme prématurément à l'exécution du programme. Elles indiquent des erreurs potentielles dans le programme et requièrent une intervention du programmeur afin d'être corrigées ou correctement gérées.
- *Les négatives vraies* : une conséquence de l'utilisation des annotations de présence dans les rangées est que l'analyse peut également rendre compte des exceptions qui sont levées dans le programme et toujours rattrapées. Le programmeur peut alors être sûr que ces exceptions sont correctement traitées dans son programme.
- *Les positives fausses inévitables* : ces exceptions ne peuvent en réalité par s'échapper lors de l'exécution du programme, mais établir cette propriété est au-delà des possibilités de la présente analyse. Les exemples typiques sont les exceptions levées dans le cas d'une division par zéro ou d'un accès hors des limites d'un tableau. En effet, le comportement de telles opérations ne peut être, en général, statiquement déterminé puisqu'il dépend de propriétés dynamiques du programme. Dans de tels cas, notre analyse suppose que ces exceptions peuvent toujours être levées. Supprimer ces reports abusifs d'exceptions requiert des analyses supplémentaires et spécialisées, des annotations par des invariants fournis par le programmeur, ou plus généralement des outils de preuve de programmes ([5, 27]).
- *Les positives fausses qui pourraient être évitées* : ces exceptions proviennent d'une perte de précision de notre analyse et pourraient conceptuellement être évitées grâce à un suivi plus fin du flot des valeurs et des exceptions. Certaines de ces *positives fausses* sont dues au flot d'information bidirectionnel inhérent à notre cadre d'unification. D'autres

peuvent correspondent à une insuffisance de récursion polymorphe durant l'inférence des types, ce dernier algorithme étant, comme nous l'avons vu, une approximation raisonnable et non une solution exacte au problème.

Nous avons manuellement inspecté les résultats de notre analyseur sur les programmes nous servant de jeu de test. Les programmes 1, 3, 4, 5 et 7 possèdent un comportement relativement simple en terme d'exceptions, et notre analyse ne reporte aucune exception abusive qui aurait pu être évitée comme mentionné ci-dessus. Seules sont mentionnées des exceptions pouvant effectivement s'échapper du programme, ainsi que des divisions par zéro et des accès hors des limites de tableaux.

Pour le programme « Knuth-Bendix », qui possède un comportement relativement complexe en matière d'exceptions, 8 exceptions (`Failure` avec 8 arguments chaînes de caractères différents) apparaissant dans le source sont correctement reportées comme ne s'échappant pas.

D'autre part, 7 exceptions (une `Invalid_argument` et 6 `Failure`) sont mentionnées comme s'échappant potentiellement, et peuvent effectivement interrompre le programme dans certaines circonstances.

Sans récursion polymorphe, notre analyse reporte deux *positives fausses* (une exception `Not_found` et une `Failure`), qui correspondent à des cas de fonctions récursives contenant un `try ...with ...` protégeant des appels récursifs (cf. exemple donné en début du chapitre 8). L'ajout de récursion polymorphe, comme discuté précédemment, permet de supprimer l'un de ces cas de reports abusifs, l'autre restant présent car notre algorithme d'inférence en présence de récursion polymorphe n'arrive pas à donner un type suffisamment polymorphe à l'une des fonctions. Toutefois, nous pensons que l'algorithme d'inférence pourrait encore être amélioré pour éliminer ce dernier cas.

Les exemples plus conséquents (8 et 9) montrent une autre source de *positives fausses* qui pourraient être évitées : les structures de données mutables (références et tableaux) contenant des fonctions. Comme mentionné en section 7.3, les variables de rangées apparaissant dans les approximations de structures de données mutables ne sont pas généralisées, et « collectent » ainsi toutes les exceptions apparaissant à leurs sites d'utilisation. Par exemple, dans :

```
let r = ref (λx....) in
let f = λy.if cond then !r y else raise C
in !r 0
```

`r` a le type $\mathtt{int} \xrightarrow{\rho} \mathtt{int}$ où ρ n'est pas généralisée. Lorsque nous typons `f`, l'effet de `raise C` est unifié avec celui de `!r y`. Ainsi ρ devient `C:Pre`; ρ' et l'application `!r 0` semble lever l'exception `C`.

9.2 L'interface utilisateur

Nous avons vu jusqu'à présent que notre analyse fournissait ses résultats sous la forme de types. Ainsi l'extraction de l'information à propos des exceptions pouvant s'échapper d'une expression repose sur la lecture des rangées annotant les types et les effets de chaque expression.

Il est clair que la taille de ces rangées peut rapidement s'avérer imposante, rendant la lecture difficile. Qui plus est, en règle générale l'utilisateur ne sera intéressé que par la forme des rangées étiquetant les types de fonctions et celles apparaissant sur les effets des expressions. En effet, s'il définit une fonction sur les listes, de type `int list` → `bool list`, il sera vraisemblablement peu intéressé par la forme de la rangée des listes argument et résultat de la fonction.

De plus, l'existence de variables (de rangée et de présence) généralisées, n'apparaissant qu'en position positive, qui dénotent en fait l'absence de valeurs (cf. 4.1.5) ne fait que rendre cette lecture plus complexe. En effet, l'utilisateur doit trouver un moyen de vérifier si chaque variable qui l'intéresse représente un ensemble vide en regardant toutes ses occurrences d'apparition. Pour rendre la tâche un peu plus délicate, notons qu'il n'est pas évident de se rendre compte, au sein d'une expression de type compliquée, si une variable apparaît en position négative ou positive.

Enfin, s'apercevoir qu'une exception peut être levée et non rattrapée est important, mais pouvoir la tracer, c'est-à-dire comprendre d'où elle provient, est également primordial. Il est donc nécessaire d'aider l'utilisateur à suivre (au moins à approximer) le chemin que les exceptions peuvent emprunter avant d'interrompre le programme.

Afin d'illustrer ces propos, exhibons une sortie typique des résultats de notre analyseur, sans aucun traitement des informations obtenues :

```
val kb_complete :
   (TY0 * TY0
    -{__Not_found22 :Pre ;
      __Failure13
       (string
        ["Rule numbers not in sequence" :Pre ; "change" :Pre ;
         "kb_completion" :Pre ; "pretty_term : infix arity <> 2" :Pre ;
         "replace" :Pre ; _'a]) ;
      __Invalid_argument14 (string["List.fold_left2" :Pre ; _'b]) ; _'c}->
    bool['d])
   -{'e}->
   TY1 -{'f}->
   TY2
   -{__Not_found22 :Pre ;
     __Failure13
      (string
       ["Rule numbers not in sequence" :Pre ; "change" :Pre ;
        "kb_completion" :Pre ; "pretty_term : infix arity <> 2" :Pre ;
        "replace" :Pre ; _'a]) ;
     __Invalid_argument14 (string["List.fold_left2" :Pre ; _'b]) ; _'c}->
   unit[() :Pre ; 'g]
     where TY0 = term
                 [Term
                   (string["*" :Pre ; "+" :Pre ; "-" :Pre ; "/" :Pre ; _'h] * TY3) ;
                  Var (int[_]) ; _'i]
     where TY1 = (int[_] * (int[_] * (TY0 * TY0))) list
                 [[] :Pre ; : : ((int[_] * (int[_] * (TY0 * TY0))) * TY1) ; _'j]
     where TY2 = ('a * ('b * (TY0 * TY0))) list
                 [[] :Pre ; : : (('a * ('b * (TY0 * TY0))) * TY2) ; 'k]
     where TY3 = TY0 list[[] :Pre ; : : (TY0 * TY3) ; _'l]
kb_complete :\ _'a

unit[() :Pre ; _'a]
  :\ __Not_found22 :Pre ;
   __Failure13
    (string
     ["Rule numbers not in sequence" :Pre ; "change" :Pre ; "kb_completion" :Pre ;
      "lex_ext" :Pre ; "pretty_term : infix arity <> 2" :Pre ; "replace" :Pre ;
      _'a]) ;
   __Invalid_argument14 (string["List.fold_left2" :Pre ; _'b]) ; _'c
```

Ce compte-rendu d'analyse est directement extrait du programme de test Knuth-Bendix. Il montre le type d'une des fonctions principales du programme ainsi que l'application finale qui lance l'exécution proprement dite de ce programme. La phase de typage du compilateur Objective Caml nous aurait simplement donné les deux types suivants :

```
val kb_complete : (term * term -> bool) ->
    (int * (int * (term * term))) list ->
    ('a * ('b * (term * term))) list -> unit
= <fun>

- : unit = ()
```

Cet exemple illustre bien la difficulté qu'il peut y avoir à interpréter les résultats bruts de l'analyse !

Pour toutes ces raisons, nous avons jugé bon de créer un outil, muni d'une interface graphique, devant aider l'utilisateur dans ces tâches. Si un tel outil ne soulève pas de problème théorique particulier, il requiert toutefois un effort d'ingénierie tout à fait conséquent et non négligeable en terme d'effort de programmation.

Afin de donner une idée de ce qu'il est possible de faire, nous allons exposer rapidement ici les concepts fonctionnels de l'outil que nous avons réalisé.

9.2.1 La réalisation

L'outil que nous allons succinctement présenter a été réalisé en Objective Caml grâce à la bibliothèque CamlTk [40]. Il vient en remplacement des phases finales « d'édition de liens » et d'affichage dont nous avons parlé en section 7.6, dont il reprend le principe à ceci près que les résultats sont désormais affichés graphiquement via un système de fenêtrage permettant l'utilisation de la souris.

L'affichage des résultats d'une analyse se décompose principalement en trois parties. En premier lieu, une fenêtre de sélection des modules composant le programme analysé est affichée, dans laquelle l'utilisateur peut choisir quel module il désire examiner.

Une fois ce module choisi, deux fenêtres s'ouvrent, l'une permettant de visualiser l'interface (les types) de ce module, l'autre permettant de parcourir le source de ce dernier.

9.2.2 Les fonctionnalités

Nous allons énumérer dans les grandes lignes ici les fonctionnalités offertes par cette interface utilisateur.

- **Simplification des types « vides »** : comme mentionné précédemment il est fastidieux pour l'utilisateur de rechercher quelles variables dénotent des ensembles vides d'exceptions et quelles variables doivent être considérées comme importantes. L'imprimeur de types inclu une phase chargée d'automatiser ce traitement et supprime ainsi les variables (de rangée, comme de présence) n'apportant aucune information pertinente (c'est-à-dire celles généralisées et n'apparaissant qu'en position positive dans le programme).
- **Enroulement / déroulement des expressions de types** : comme nous l'avons vu les expressions de types peuvent rapidement devenir imposantes du fait des rangées les étiquetant. Notre interface utilisateur fournit donc un mécanisme permettant de rendre apparent ou non le contenu d'une rangée simplement en cliquant sur son texte. On appelle couramment une telle zone cliquable une « ancre ». À l'origine toutes les rangées sont « enroulées », ne laissant en fait voir à l'utilisateur, qu'un type comme ML lui en aurait fourni. Si l'utilisateur est désireux d'en savoir plus sur la structure d'une rangée, il lui suffit alors de cliquer dessus pour la voir s'expanser. Les types étant une structure imbriquée, il est alors possible de cliquer sur les rangées étant apparues afin d'examiner à leur tour leur contenu.
- **Abréviation des types récursifs** : notre algèbre de types étant fortement récursive, l'affichage des type doit permettre l'affichage de cycles de manière simple. À cet effet,

une détermination des cycles est effectuée permettant de nommer des morceaux de types en vue de les réutiliser sans les expanser à nouveau. Notons que ce mécanisme est d'une part voué à une simplification de lecture des types, mais demeure également impératif si nous ne voulons pas que l'afficheur boucle indéfiniment….

- **Élimination des ancres inutiles** : nous avons vu que le mécanisme d'ancres fourni par l'interface permet de contrôler la quantité d'information que l'utilisateur désire voir sur son écran. Toutefois, si une rangée s'avère « vide », comme c'est le cas au dessus des flèches des fonctions ne levant aucune exception, il est inutile de laisser l'utilisateur croire qu'il y a de l'information à regarder. Au mieux, ceci lui occasionne une perte de temps car il doit regarder toutes les rangées pour connaître celles dignes d'intérêt. Au pire, cela rend la quantité d'information trop importante, noie l'utilisateur, et fait perdre toute utilité à notre interface. Ainsi, nous supprimons automatiquement toute ancre sur les rangées vides. Il est ainsi facile et rapide de repérer les fonctions pouvant lever des exceptions, juste en regardant quels sont les types flèches possédant une ancre.
- **Filtrage des exceptions non désirées** : nous permettons à l'utilisateur d'omettre l'affichage de certaines exceptions. Ceci prend son intérêt lorsque ce dernier ne désire se concentrer que sur des exceptions importantes et non des *positives fausses* comme celles que nous avons vu précédemment en section 9.1.2, dans le genre des divisions par zéro ou des accès hors des limites de tableaux, exceptions que de toute façon l'analyse n'est pas apte à traiter précisément.
- **Suivi des exceptions dans l'arbre du programme** : une fois que les exceptions non rattrapées sont identifiées, il est possible de les suivre grâce à la fenêtre affichant le source du programme. L'utilisateur peut alors sélectionner à la souris toute (sous-) expression du programme afin d'en connaître à la fois le type et l'effet. Notons que l'affichage des types et des effets obtenus à partir de cette fenêtre bénéficient également des mécanismes de simplification décrits ci-dessus. Cette fonctionnalité fournit en fait une possibilité de navigation directe dans l'arbre de syntaxe du programme annoté par les types et les effets. Afin que le programmeur s'y retrouve plus aisément, le texte de son programme n'est pas affiché par un « pretty-printer », mais directement issu du fichier que le programmeur a écrit. Cette méthode possède un second avantage d'importante : il n'est ainsi pas nécessaire d'écrire un complexe afficheur d'expressions du langage, afficheur qui de toute façon ne serait pas du goût de tout le monde, chaque programmeur ayant son propre style d'écriture.

9.2.3 Écrans

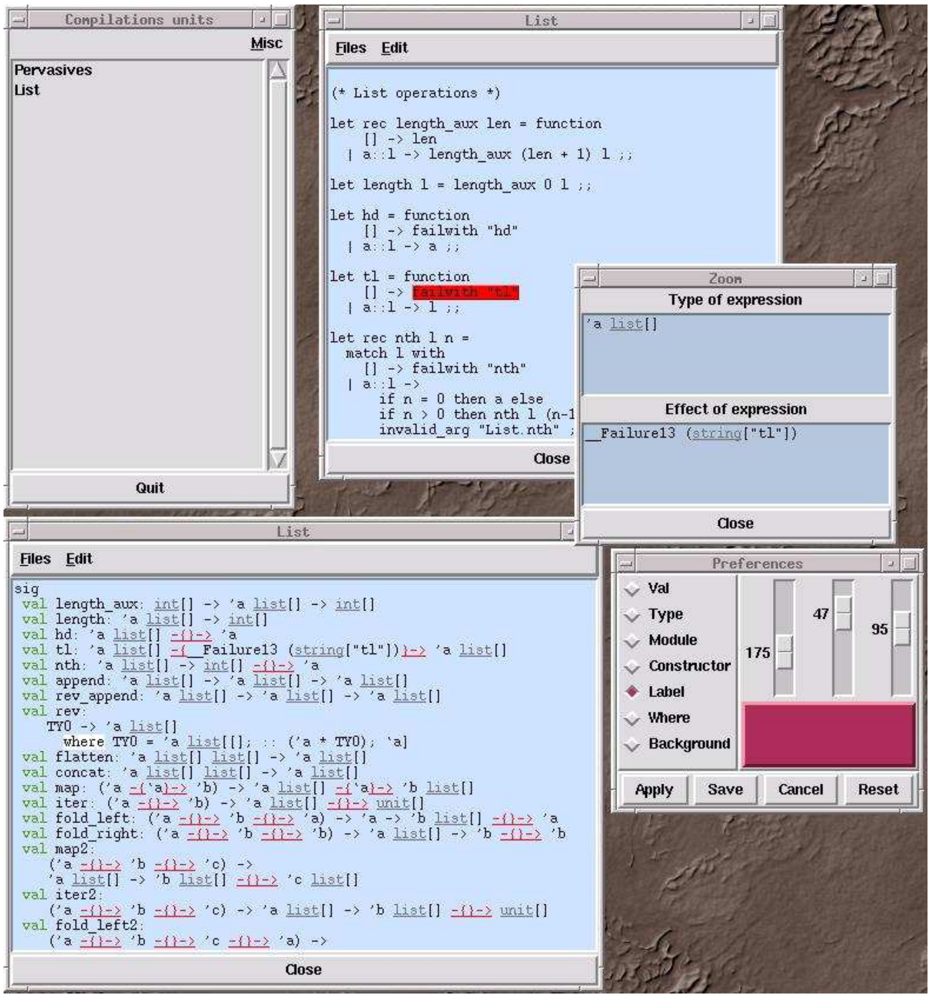
Afin de donner une idée de l'apparence d'un tel outil, nous donnons ici quelques photos d'écran issues de l'utilisation de notre interface utilisateur.

Cette première impression d'écran montre cinq fenêtres.

- La fenêtre de sélection du ou des modules à examiner (en haut à gauche).
- La fenêtre de visualisation du source du module (en haut à droite), dans laquelle une sous-expression de la fonction `tl` a été sélectionnée afin d'en afficher le type et l'effet.
- La fenêtre de visualisation de l'interface (donc des types) du module. Afin de montrer le mécanisme d'enroulement / déroulement des types, nous avons cliqué sur l'ancre de l'effet de la fonction `tl` ainsi que sur celle du premier argument de la fonction `rev`. Ce dernier permet au passage de montrer comment les cycles sont affichés. Nous pouvons également remarquer que certaines fonctions (comme `length` ne portent pas d'ancre sur leur type flèche. Ceci montre que ces fonctions ont un effet vide, et donc ne peuvent pas lever d'exceptions.
- En sur-affichage en haut, une fenêtre détaillant le type et l'effet d'une expression (celle sélectionnée en rouge dans la fenêtre de visualisation du source du module et dont nous avons parlé ci-dessus). On remarquera que nous avons cliqué sur l'ancre `string`

paramétrant le constructeur `Failure` de l'effet de l'expression. Par contre, nous avons laissé volontairement l'approximation du type `list` de cette expression « enroulée ».

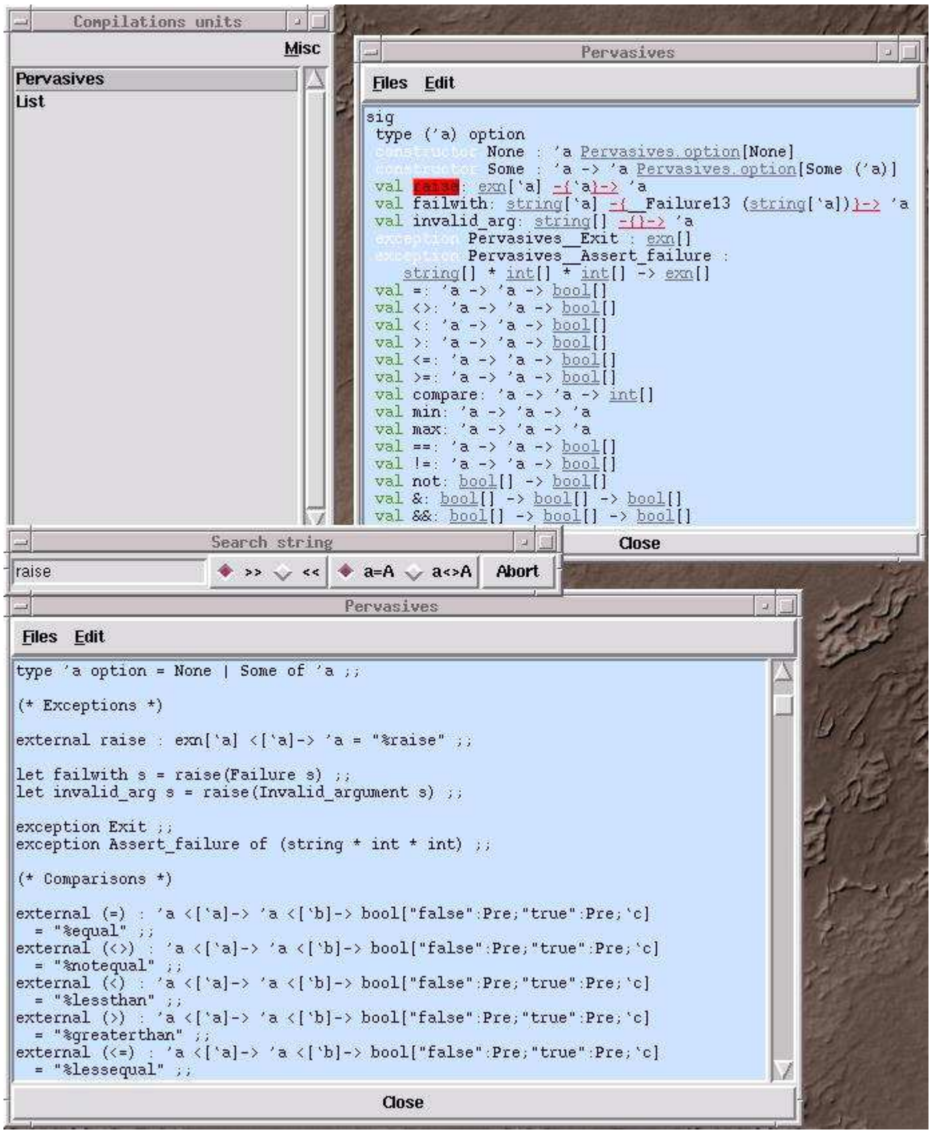
- Et enfin, en bas à droite, la fenêtre de « préférences » qui permet à l'utilisateur de configurer la couleur des éléments syntaxiques s'affichant dans ses fenêtres.



La seconde vue proposée montre un autre module. Les fenêtres sont similaires à la première, à ceci près que nous avons enlevé les fenêtres de « préférences » et de « zoom ». À la place, nous exposons la fenêtre de recherche de texte. Notons que celle-ci nous a permis de trouver la fonction `raise` (en sur-brillance dans la fenêtre de visualisation de l'interface du module, en haut a droite). Nous voyons ainsi clairement que l'effet de cette fonction est lié à son argument. Un examen de la fenêtre de visualisation du source du module nous montre que cette fonction `raise` est une fonction externe au langage. En effet, celle-ci est définie comme « external », et son type est explicitement fourni *avec des annotations* dans le fichier d'interface (`external raise : exn[‘a] <[‘a]-> 'a = "%raise"`).

Nous pouvons également voir le type de la fonction `failwith` qui montre que son argument chaîne de caractères apparaît en tant qu'argument du contructeur `Failure` qu'elle lève dans son effet.

Pour terminer, notons que nous voyons, au travers des noms de constructeurs d'exceptions, le mécanisme de désambiguage des noms de constructeurs dont nous avons parlé au chapitre 7.6.1. Ainsi, l'exception `Exit` étant définie dans le module nommé `Pervasives`, son nom « réel » dans les expressions de type apparaît sous la forme : `Pervasives__Exit`.



Chapitre 10

Conclusion

Il est désormais temps de nous diriger vers le mot de la fin de ce document, en essayant d'adopter une vue d'ensemble du travail effectué tout au cours de cette thèse et de déterminer quelles sont les contributions majeures qu'elle apporte au problème de l'analyse d'exceptions non rattrapées en Objective Caml.

Mais auparavant, nous allons faire un tour d'horizon des travaux antérieurs effectués dans le domaine de l'analyse d'exceptions non rattrapées. Nous décidons sciemment de discuter ce point à cet endroit tardif de cette thèse avec l'idée que le lecteur, ayant lu les précédents chapitres, connaît désormais mieux le problème et comprendra ainsi plus facilement les lignes qui vont suivre.

10.1 Travaux antérieurs

Compile-time Detection of Uncaught Exceptions for Standard ML Programs [66]

Dans ce travail, l'auteur présente une analyse d'exceptions non rattrapées pour le langage SML, basée sur un mécanisme d'interprétation abstraite. Dans un premier temps, les programmes SML sont transformés dans un langage intermédiaire. Une sémantique concrète est donnée pour ce langage, de laquelle une sémantique abstraite est ensuite définie afin de dériver un interpréteur abstrait.

L'implémentation de cet analyseur est effectuée à l'aide du générateur d'analyseurs *Z1* [67] auquel on fournit une spécification de cet interpréteur abstrait. Le résultat de *Z1* est un source en langage C qui deviendra le programme d'analyse lui-même une fois compilé et lié avec les analyseurs syntaxique et lexical du langage cible (SML en l'occurrence).

L'analyseur ainsi généré est apte à traiter SML complet. Il gère également les arguments des exceptions.

Son principal inconvénient réside dans sa lenteur d'analyse qui, selon l'auteur, ne permet pas d'utilisation de manière interactive (6 heures pour analyser le programme ML-LEX dont la taille est de 1229 lignes).

D'autre part, aucun mécanisme d'analyse fichier par fichier n'est disponible (obligeant à ré-analyser tout le programme dès qu'un de ses fichiers a été modifié). Enfin, le langage SML ne fournissant pas d'objets, cet aspect n'est pas envisagé dans ce travail.

Towards A Cost-Effective Estimation of Uncaught Exceptions in SML Programs [68]

Dans ce travail, les auteurs présentent une nouvelle analyse pour SML, basée, non plus sur un interpréteur abstrait, mais sur une analyse de flot. Cette analyse est décomposée en deux parties distinctes : l'approximation du flot de contrôle d'une part ; l'approximation du flot d'exceptions d'autre part. À partir du graphe d'appel généré par la première phase d'analyse, le flot des exceptions est dérivé comme un ensemble de contraintes d'inclusion dont le plus petit modèle est le résultat de leur analyse.

Toutefois, afin d'obtenir une vitesse d'analyse compatible avec une utilisation interactive, un certain nombre de choix simplificateurs sont effectués. En particulier les arguments des exceptions sont ignorés à moins qu'ils ne soient des exceptions. D'autre part, pour l'approximation du graphe d'appel, lorsque les fonctions à appeler sont colportées par des exceptions non rattrapées, une approximation « plus grossière » est décidée : il faut alors regarder les endroits où des exceptions sont construites et collecter les fonctions y apparaissant afin de constituer l'ensemble d'approximation. Il est toutefois possible de raffiner cet ensemble en choisissant seulement les fonctions dont le type est unifiable avec l'argument de l'exception.

L'analyse des programmes fichiers par fichier et les aspects objets restent non disponibles. L'analyseur exposé dans ce travail au moment de sa publication était considéré comme un prototype et ne prétendait pas traiter le langage SML dans sa totalité.

Nous devons également noter que, comme nous l'avons vu au début de cette thèse, le flot de contrôle du programme et le flot des exceptions sont naturellement dépendants. Ainsi le fait de les découpler résulte en une perte de précision qu'une analyse mixant les deux (comme la nôtre) ne subit pas.

Du point de vue de la vitesse d'exécution cette analyse s'avère incomparablement plus rapide que la précédente, analysant désormais le programme ML-LEX en 4.58 secondes.

An Extended Type System for Exceptions [13]

Dans ce travail les auteurs exposent une analyse d'exceptions non rattrapées pour un sous-ensemble de ML, basée sur un système de types à effets. À l'instar de notre système, le type « flèche » des fonctions est annoté par leur effet latent, c'est-à-dire l'ensemble des exceptions pouvant être levées lors d'une application de ces fonctions.

Toutefois, les exceptions ne sont pas ici considérées comme valeurs de première classe, et les exceptions avec argument ne sont pas prises en compte. Puisque les exceptions ne sont pas considérées comme de première classe, il en résulte l'impossibilité d'analyser finement leur parcours dans les arguments des fonctions, d'où une perte de précision de l'analyse.

S'agissant d'un travail plutôt théorique, aucun aspect d'implantation n'est abordé. En particulier, la structure des ensembles d'exceptions reste ouverte, ne mentionnant pas de mécanisme précis comme le sont les rangées dans notre travail. Dans ce cadre, seul le type « flèche » est annoté, contrairement à notre système qui se permet d'annoter tous les constructeurs de type. Ainsi, contrairement à notre analyse dans laquelle les rangées servent à approximer les ensembles d'exceptions mais aussi tous les autres ensembles de valeurs, aucune approximation sur les ensembles d'autres valeurs n'est effectuée.

Ce travail étant basé sur un langage restreint, toutes les constructions habituellement présentes dans les langages de taille réelle ne sont pas prises en compte, les problèmes d'analyse séparée et les objets ne sont pas non plus abordés.

Program Analysis Using Mixed Term and Set Constraints [8]
Tracking down Exceptions in Standard ML Programs [9]

Dans ce travail, les auteur présentent un analyse d'exceptions pour SML, structurée comme un système de types à effets. Pour chaque expression du programme, l'analyseur infère un type et un effet. Le type correspond au type habituel fournit par SML à ceci près que ceux des exceptions, des fonctions et des types construits colportent des paramètres de type supplémentaires représentant des ensembles d'exceptions. Quant à l'effet d'une expression, il représente l'ensemble des exceptions pouvant être levées lors de son évaluation. Toutefois, contrairement à la nôtre, cette analyse ne cherche pas à approximer les autres valeurs que les exceptions. Autrement dit, il n'existe pas d'annotation sur les constructeurs de type comme nous le faisons via nos rangées.

L'implémentation de cet analyseur est effectuée à l'aide du moteur d'analyse *BANE* [11] lui-même basé sur un mécanisme mixant contraintes d'inclusion et contraintes d'unification. Les résultats de l'analyse sont extraits de l'ensemble de contraintes puis mémorisés dans un fichier. La visualisation du résultat cette l'analyse est ensuite effectuée à l'aide d'un mode Emacs dédié nommé *PAM*.

La précision de l'analyse ainsi obtenue est toute à fait satisfaisante dans la majorité des cas, de même que sa vitesse d'exécution (3.5 secondes pour le programme ML-LEX, ce qui reste toutefois bien supérieur aux 0.2 secondes nécessaires à notre analyseur).

Toutefois les arguments des exceptions ne sont pas approximés, c'est-à-dire que les exceptions ne sont connues que par leur constructeur (néanmoins, comme nous l'avons dit ci-dessus, le fait de rajouter des paramètres de type enregistrant les exceptions contenues dans les structures de données permet de mémoriser qu'une exception peut en contenir d'autres, soit directement, soit via des fonctions). Les objets et l'analyse fichier par fichier ne sont pas envisagés.

De plus, afin d'accélérer l'analyse, toutes les exceptions pouvant apparaître dans une structure de données sont fusionnées, d'où une perte de précision qui, dans certains cas dégrade sévèrement l'analyse. Rappelons (cf. chapitre 7.1.3) que nous effectuons également une telle fusion, mais uniquement dans le cas des structures de données récursives. Pour terminer, selon les auteurs, bien que l'analyse en elle-même s'adapte à des programmes de grande taille, la taille des résultats finit par devenir problématique (plusieurs centaines de méga-octets) et rendre sa visualisation impraticable, même avec l'outils *PAM*.

10.2 Notre travail

La première remarque qui me vient à l'esprit est que nous avons aboutit à un formalisme nettement plus compliqué que ce à quoi nous nous attendions lorsque j'ai débuté cette thèse. Nous pensions alors, suivant notre intuition du moment, qu'une analyse de flot simple suivie d'un examen du graphe d'appel suffiraient *grosso modo* à analyser les exceptions de manière suffisamment précise. Autrement dit, il semblait possible que notre analyse soit réalisée à partir de la combinaison de techniques déjà existante, habilement combinées. Or non...

Ceci montre toute la difficulté qui réside en règle générale dans les analyses statiques de programmes. En effet, les paradigmes offerts par ces derniers étant de plus en plus variés et puissants, il en découle que les analyses devant traiter ces langages deviennent également complexes. Rappelons qu'Objective Caml fournit à lui tout seul l'ordre supérieur, les modules, les objets, les exceptions et fonctions valeurs de première classe, des structures de données polymorphes, autant de caractéristiques qui ont contraint notre formalisme à se complexifier afin de les traiter le plus finement possible...

De plus, nous voulons désormais des analyses qui « tournent vite » et qui soient également adaptées à de gros programmes. Ceci ne fait donc que rajouter des contraintes sur les formalismes possibles, nécessitant un examen minutieux de la complexité des algorithmes mis en jeu.

Le but de ce travail de recherche a donc été de réaliser une analyse statique de programmes, non pas pour en améliorer la vitesse d'exécution comme c'est souvent le cas au sein des compilateurs (partage d'expressions communes, détermination de la durée de vie de structures de données, détermination des registres « vivants », recherche d'invariants de boucles, optimisation de code en règle générale, GC statique, etc...), mais pour en améliorer la *sécurité*, la *sûreté*.

Nous avons vu comment, en juxtaposant toutes les exigences que nous avions pour notre analyse, petit à petit nous sommes d'abord passés de l'idée d'une analyse de flot conventionnelle comme celles déjà existantes, à l'idée d'un système de types. Puis, progressivement nous avons vu quelle devait être la forme de l'algèbre de types de ce système, toujours poussés par les exigences que nous avions décidées que notre analyse devait satisfaire.

Nous pouvons noter que notre analyse illustre un cas de plus d'utilisation des rangées qui se sont déjà avérées un formalisme puissant et flexible en de nombreuses occasions.

Il est souvent dit que les analyses de programmes basées sur des contraintes d'unification sont plus rapides mais moins précises que des analyses basées sur des contraintes plus générales (inclusion en particulier) comme la CFA ou la SBA. Pour l'analyse d'exceptions, notre expérience indique que la combinaison de contraintes d'unification, de `let`-polymorphisme et de récursion polymorphe est en pratique presque aussi précise que les techniques citées précédemment. Le seul cas dans lequel notre analyse s'avère sensiblement moins précise que des analyses à base de contraintes d'inclusion est dû à l'utilisation intensive des références (en règle générale, des structures de données mutables, donc dont le type ne peut pas être généralisé).

La vitesse de notre algorithme d'analyse semble très satisfaisante (bien que sa complexité théorique soit au moins aussi élevée que celle de l'inférence de types de ML).

Grâce à cette efficacité, nous pouvons nous permettre de conserver plus d'informations sur les arguments de exceptions que ne le peuvent les autres analyses, augmentant ainsi grandement la précision de l'analyse sur certains programmes ML. Ainsi, il semble que nous nous trouvions dans un cas où le « moins fait le plus », c'est-à-dire dans lequel une technique *a priori* imprécise (l'unification) nous permet finalement d'accroître la précision de l'analyse.

Ainsi, notre analyse est capable de gérer finement plusieurs aspects complexes liés aux exceptions, à savoir en particulier :

- les exceptions avec arguments,
- les structures de données paramétrées,
- les exceptions comme valeurs de première classe,
- les structures de données encapsulant des exceptions à une profondeur arbitraire,
- l'ordre supérieur,
- les traits impératifs et fonctionnels du langage,
- le langage de modules (avec structures et foncteurs),
- potentiellement les objets.

De plus, ayant réussi à mettre au point un système d'analyse module par module, nous avons ainsi pu réaliser une implantation « grandeur nature », c'est-à-dire non pas seulement un prototype démontrant la faisabilité du principe théorique, mais un véritable outil utilisable dans le cycle de développement d'applications d'envergure réelle. Cet outil sera d'ailleurs à terme distribué parmi les contributions disponibles pour l'environnement de développement d'Objective Caml. Quelques utilisateurs privilégiés existent d'ores et déjà, qui ont pu nous rapporter des échos favorables quand à l'utilisabilité de notre système.

Un autre aspect intéressant de ce travail réside dans le fait que notre analyse a réussi à s'inscrire harmonieusement dans la philosophie de ML. Autrement dit, elle repose sur un système de types dans lequel l'inférence demeure possible, tout comme ML infère le type des programmes avec sa propre algèbre. De plus, nous avons vu qu'à l'exception de quelques rugosités relevant principalement de problèmes plutôt techniques que théoriques, nous avons réussi à intégrer le langage de modules complet dans notre cadre de travail. Quant aux objets, même si nous ne les avons pas décrits dans le détail dans cette thèse, leur prise en compte s'avère très simple puisqu'elle ne nécessite qu'une très mineure modification de notre algèbre de types (nous devons seulement inclure l'algèbre traitant le type des objets et des classes à la nôtre, en leur ajoutant des effets).

Il est reconnu que les types sont d'une grande aide au programmeur puisqu'ils lui permettent de s'assurer qu'il fait un usage cohérent des structures qu'il manipule dans ses programmes. Il est encore plus agréable que ces types soient inférés automatiquement par le compilateur afin de décharger le programmeur du fastidieux travail de déclaration. Notre analyse illustre une autre facette des types. Comme en ML, les types que notre système fournit sont (nous l'espérons) très utiles au programmeur. Néanmoins, étant donné leur forme, l'inférence qui était un confort en ML habituel, devient désormais une obligation. En effet, comme nous l'avons vu au cours des exemples cités dans cette thèse, il est hors de question que le programmeur doive écrire les annotations à la main, celles-ci étant trop compliquées. De même est-il nécessaire d'aider le programmeur à déchiffrer (décrypter devrions-nous peut-être dire) ces types.

Ainsi dans notre analyse, les types, bien qu'utiles, peuvent dans certains cas tendre à devenir plus « encombrants » que déchiffrables. C'est le revers des systèmes de types riches, qui fournissent beaucoup d'information sur les programmes.

C'est la raison pour laquelle nous avons décidé de rester coûte que coûte dans une logique d'inférence et de mettre au point un outil aidant à la lecture des résultats de l'analyse.

Cet aspect du travail est intéressant, non seulement pour rendre notre analyseur utilisable en pratique, mais également afin de comprendre les liens qu'il existe entre le modèle théorique sous-jacent à l'analyse et ce que désirerait « voir » un utilisateur de notre analyseur.

Parallèlement au principe de l'analyse elle-même, cette thèse nous a permis de mettre au point un nouvel algorithme d'inférence de types en présence de récursion polymorphe. Les principaux atouts de cet algorithme résident en sa rapidité, sa robustesse, son intégration aisée au sein de notre typeur (analyseur) d'origine (et en général au sein d'un typeur conventionnel de ML utilisant un mécanisme de niveaux) et enfin sa capacité à être adapté à des algèbres de types *modulo* une théorie équationnelle (nous ne clamons pas pour autant que toute théorie équationnelle permet l'utilisation de notre algorithme, mais au moins certaines, et ce sans impact sur l'algorithme lui-même, comme le montre l'exemple de notre analyse).

Le problème de la règle μ semble pourtant de prime abord bien éloigné de l'analyse d'exceptions. D'ailleurs, nous avons nous-mêmes été très surpris de nous retrouver à chercher une solution (une approximation, bien entendu) à ce problème, en partant initialement de nos exceptions non rattrapées ! Ceci montre qu'il existe des relations surprenantes entre des questions apparemment totalement dépourvues liens.

Il est très satisfaisant de voir que l'ajout de cet algorithme dans notre analyse aura permis un gain de précision appréciable au prix d'une perte de vitesse très modique. Nous avons vu au cours de cette thèse que la nécessité de « plus de polymorphisme » est un problème déjà rencontré dans plusieurs autres travaux et analyses de programmes. Aussi est-il permis d'espérer que ce nouvel algorithme pourra être utilisé dans ces cadres afin d'en améliorer les performances.

Nous pensons d'ailleurs qu'il est possible de raffiner cet algorithme de façon à lui faire accepter une plus large classe de programmes, en particulier, ceci nous permettrait sans doute d'éliminer la dernière exception abusivement reportée dans le test « Knuth-Bendix » que nous avons eu l'occasion d'étudier.

Il serait intéressant de chercher à savoir si les techniques et le cadre théorique mis en œuvre ce travail pourraient être étendus et à nouveau utilisés pour d'autres styles d'analyses. Par exemple, il est envisageable de constituer une approximation du graphe d'appel d'un programme en annotant les types « flèche » des fonctions par les noms de ces fonctions. D'autres analyses sont sans doutes possibles et restent à trouver.

Comme mentionné précédemment, l'algorithme permettant l'inférence en présence de récursion polymorphe n'est pas parfait (par la nature même du problème qui est indécidable). Nous avons vu qu'il laissait une dernière exception abusivement rapportée sur un des programmes de test. D'une part, nous pouvons envisager d'améliorer cet algorithme, et d'autre part, il serait possible de rechercher dans quelle mesure celui-ci est adaptable à d'autres algèbres de types que celles que nous avons étudié dans cette thèse. En particulier l'introduire de concert avec un mécanisme de sous-typage reste à ma connaissance un problème ouvert.

Nous avons commencé à examiner, au chapitre 4.4.2, le problème de conservativité de notre système de types par rapport à celui de ML habituel, plus particulièrement dans le cadre de la généralisation de types.

Rappelons que le problème était de savoir si une dérivation en ML conventionnel de la forme :

$$E \oplus (x : \texttt{int} \rightarrow \texttt{int}) | - e : \alpha$$

pouvait avoir pour équivalent, dans notre système, une dérivation de la forme :

$$E \oplus (x : \texttt{int} \xrightarrow{\ldots\alpha\ldots} \texttt{int}) | - e : \alpha$$

Dans le premier cas, la variable α était généralisable, puisque non libre dans l'environnement de typage, alors que dans le second cas elle ne l'était plus car elle apparait désormais dans cet environnement.

Si nous arrivions à prouver que toute variable de type apparaissant dans les rangées apparaissent également dans les types ML sous-jacents, alors ceci nous assurerait qu'aucun problème ne se pose. En effet, dans un tel cas, soit la variable n'existe dans aucun des deux systèmes, soit elle elle existe dans les deux et dans les deux alternatives, la généralisation s'effectuerait de façon identique dans chaque système de types. Ceci correspondrait à deux dérivations de la forme :

$$E \oplus (x : \alpha \xrightarrow{\ldots\alpha\ldots} \alpha) | - e : \alpha$$

$$E \oplus (x : \texttt{int} \xrightarrow{\ldots} \texttt{int}) | - e : \alpha$$

Malheureusement, nous avons vu précédemment (cf. chapitre 7.6) que ce n'était pas le cas (dû au mécanisme de soustraction dans le cas d'un motif absorbant). Toutefois, dans ce cas, nous avions intuitivement noté que ces variables n'étant pas contraintes, elles n'apparaissaient nulle part ailleurs que dans la rangée et donc en particulier pas dans l'environnement, ce qui n'empêche pas de généralisation.

Tous ces arguments ne constituent certes pas une preuve ! C'est pourquoi nous laissons ce problème comme ouvert pour des travaux ultérieurs. Toutefois, jusqu'à présent, lors de nos tests, il s'est avéré pratiquement que cette propriété de conservativité semblait être respectée : nous n'avons eu aucun programme bien typé par ML habituel et rejeté par notre système de types. Mais encore une fois, ceci est une interrogation non résolue, et reste à approfondir...

Enfin, nous pouvons nous demander dans quelle mesure notre analyse pourrait avoir un lien avec DCC ([1]), « A Core Calculus of Dependency ». En effet, ce dernier présente un formalisme général pour les analyses de flot, tentant d'extraire un dénominateur commun, en particulier dans le cadre d'analyses ayant trait à une notion de *dépendance*. Peut-être notre analyse peut-elle se plonger dans ce cadre théorique. Peut-être ce cadre théorique pourrait-il nous fournir, une fois spécialisé, un certain nombre d'exemples d'analyses pour lesquelles les techniques développées dans cette thèse pourraient s'appliquer.

Bibliographie

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, et Jon G. Riecke. A core calculus of dependency. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 147–160. ACM, 1999.

[2] Alexander Aiken et Brian. R. Murphy. Implementing regular tree expressions. *Proceedings of the ACM conf. on Functional Programming Languages and Computer Architecture*, pp. 427–447, 1991.

[3] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, et Gilles Kahn. A simple applicative language : Mini-ML. Rapport technique 529, INRIA, 1986.

[4] Luis Damas et Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pp. 207–212. ACM Press, 1982.

[5] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, et James B. Saxe. Extended static checking. Rapport de recherche 159, Compaq SRC Research, 1998.

[6] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. Thèse de doctorat, Department of Computer Science, University of Copenhagen, janvier 1999.

[7] Manuel Fähndrich et Alexander Aiken. Making set-constraint based program analyses scale. Rapport technique 96–917, University of California at Berkeley, Computer Science Division, 1996.

[8] Manuel Fähndrich et Alexander Aiken. Program analysis using mixed term and set constraints. In *Static Analysis Symposium '97*, numéro 1302 in Lecture Notes in Computer Science, pp. 114–126. Springer-Verlag, 1997.

[9] Manuel Fähndrich, Jeffrey S. Foster, Alexander Aiken, et Jason Cu. Tracking down exceptions in Standard ML programs. Rapport technique, University of California at Berkeley, Computer Science Division, 1998.

[10] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, et Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Programming Language Design and Implementation 1998*, pp. 85–96. ACM Press, 1998.

[11] Manuel Fähndrich, Alexander Aiken, et al. Bane : The berkeley analysis engine for constructing constraint-based program analyses. Logiciel et documentation disponibles à `http://http.cs.berkeley.edu/Research/Aiken/bane.html`, University of California, Berkeley, CA 94720-1776 USA, 1998.

[12] Cormac Flanagan et Matthias Felleisen. Componential set-based analysis. In *Programming Language Design and Implementation 1997*. ACM Press, 1997.

[13] Juan Carlos Guzmán et Ascánder Suárez. An extended type system for exceptions. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, numéro 2265 in INRIA Research Reports, pp. 127–135. INRIA, BP 105, 78153 Le Chesnay Cedex, France, juin 1994. URL : `http://www.ldc.usb.ve/~suarez/PAPERS/except.ps`.

[14] Nevin Heintze. Practical aspects of set based analysis. Extended abstract, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, mars 1992.

[15] Nevin Heintze. *Set Based Program Analysis.* Thèse de doctorat CMU-CS-92-201, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, octobre 1992.

[16] Nevin Heintze. Set based analysis and arithmetic. Extended abstract, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, décembre 1993.

[17] Nevin Heintze. Set-based analysis of ML programs. In *Lisp and Functional Programming '1994*, pp. 306–317. ACM Press, 1994.

[18] Nevin Heintze et Joxan Jaffar. Set-based program analysis. Extended abstract, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, janvier 1991.

[19] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2) :253–289, 1993.

[20] Paul Hudak, Simon Peyton Jones, et Philip Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[21] Suresh Jagannathan et Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *22st symposium Principles of Programming Languages*, pp. 393–407. ACM Press, 1995.

[22] Suresh Jagannathan et Andrew Wright. Polymorphic splitting : An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1) :166–207, 1998.

[23] N.D. Jones, C.K. Gomard, et P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ : Prentice Hall, 1993.

[24] P.C. Kanellakis, H.G. Mairson, et J.C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassex et G.D. Plotkin (rédacteurs), *Computational Logic, Essays in Honor of Alan Robinson*, pp. 444–478. MIT Press, 1991.

[25] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Rapport technique Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, mars 1965.

[26] A. J. Kfoury, J. Tiuryn, et P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2) :290–311, 1993.

[27] K. Rustan M. Leino et Greg Nelson. An extended static checker for modula-3. In *Kai Koskimies, editor, Compiler Construction, 7th International Conference, CC'98 - Lecture Notes in Computer Science*, volume 1383, pp. 302–305. Springer-Verlag, avril 1998.

[28] Xavier Leroy. *Typage polymorphe d'un langage algorithmique.* Thèse de doctorat, Université Paris VII, juin 1992. URL : `http://pauillac.inria.fr/~xleroy/publi/these-doctorat.ps.gz`.

[29] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pp. 109–122. ACM Press, 1994.

[30] Xavier Leroy. A modular module system. Rapport de recherche 2866, INRIA, avril 1996.

[31] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5) :667–698, 1996.

[32] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Logiciel et documentation disponibles sur le Web, `http://caml.inria.fr/ocaml/`, 1996.

[33] Mark Lillibridge. *Translucent sums : a foundation for higher-order module systems.* Thèse de doctorat CMU-CS-97-122, School of Computer Science, Carnegie Mellon University, mai 1997.

[34] John M. Lucassen et David K. Gifford. Polymorphic effect systems. In *15th symposium Principles of Programming Languages*, pp. 47–57. ACM Press, 1988.

[35] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, 1978.

[36] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, numéro 167 in Lecture Notes in Computer Science, pp. 217–228. Springer-Verlag, 1984.

[37] F. Nielson, H. R. Nielson, et C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[38] Flemming Nielson et Hanne Riis Nielson. Infinitary control flow analysis : a collecting semantics for closure analysis. In *24th symposium Principles of Programming Languages*. ACM Press, 1997.

[39] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6) :844–895, 1995.

[40] François Pessaux et François Rouaix. *The CamlTk Interface*. INRIA Rocquencourt, Projet Cristal. Logiciel et documentation disponibles à `http://pauillac.inria.fr/~rouaix/camltk-readme.html`.

[41] John Plevyak et Andrew A. Chien. Iterative flow analysis. In *Proceedings of the 1994 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1994.

[42] François Pottier. A framework for type inference with subtyping. In *International Conference on Functional Programming 1998*, pp. 228–238. ACM Press, 1998.

[43] Didier Rémy. Records and variants as a natural extension of ML. In *16th symposium Principles of Programming Languages*, pp. 77–88. ACM Press, 1989.

[44] Didier Rémy. *Algèbres touffues. Application au typage polymorphe des objets enregistrements dans les langages fonctionnels.* Thèse de doctorat, Université Paris VII, 1991.

[45] Didier Rémy. Extending ML type system with a sorted equational theory. Rapport de recherche 1766, INRIA, 1992.

[46] Didier Rémy. Syntactic theories and the algebra of record terms. Rapport de recherche 1869, INRIA, 1993.

[47] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter et John C. Mitchell (rédacteurs), *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1993.

[48] Didier Rémy et Jérôme Vouillon. Objective ML : A simple object-oriented extension of ML. In *24th symposium Principles of Programming Languages*, pp. 40–53. ACM Press, 1997.

[49] Didier Rémy et Jérôme Vouillon. Objective ML : An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1) :27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[50] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pp. 513–523. North-Holland, 1983.

[51] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1) :23–41, 1965.

[52] M. Serrano. Control Flow Analysis : a Functional Languages Compilation Paradigm. In *10th Symposium on Applied Computing*, Nashville, Tennessee, USA, février 1995.

[53] Zhong Shao. Parameterized signatures and higher-order modules. Rapport technique YALEU/DCS/TR-1161, Department of Computer Science, Yale University, août 1998.

[54] Olin Shivers. Control-flow analysis in Scheme. In *Programming Language Design and Implementation 1988*, pp. 164–174. ACM Press, 1988.

[55] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* Thèse de doctorat CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, mai 1991.

[56] Jean-Pierre Talpin et Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2) :245–296, 1994.

[57] M. Tofte et J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *21st symposium Principles of Programming Languages*, pp. 188–201. ACM Press, 1994.

[58] Mads Tofte. Operational semantics and polymorphic type inference. Thèse de doctorat CST-52-88, University of Edinburgh, 1988.

[59] Mads Tofte. Principal signatures for higher-order program modules. In *19th symposium Principles of Programming Languages*, pp. 189–199. ACM Press, 1992.

[60] Philip Wadler. Theorems for free ! In *Functional Programming Languages and Computer Architecture 1989.* ACM Press, 1989.

[61] Mitchell Wand. Complete type inference for simple objects. In *Logic in Computer Science 1987*, pp. 37–44. IEEE Computer Society Press, 1987.

[62] Pierre Weis et Xavier Leroy. *Le langage Caml.* Dunod, 1999.

[63] Andrew K. Wright. Typing references by effect inference. In *European Symposium on Programming*, *Lecture Notes in Computer Science*, volume 582. Springer-Verlag, 1992.

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

[65] Andrew K. Wright et Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, 1994.

[66] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science*, volume 864, pp. 238–254. Springer-Verlag, proceedings of the first international static analysis symposium édition, 1994.

[67] Kwangkeun Yi et Luddy Harrison. System z1 programming manual. CSRD Report N^o 1283, Center for Supercomputing Research and Development, University of Illinois at Urbana-Chaimpaign, 465 CSRL, 1308 West Main Street, Urbana, IL 61801-2307, janvier 1993.

[68] Kwangkeun Yi et Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Static Analysis Symposium '97*, numéro 1302 in Lecture Notes in Computer Science, pp. 98–113. Springer-Verlag, 1997.

[69] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Inform. Contr. 10*, pp. 189–208, 1967.

Table des matières

Index