View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL Université de Savoie

Preuves, types et sous-types

Frédéric Ruyer

To cite this version:

Frédéric Ruyer. Preuves, types et sous-types. Mathématiques [math]. Université de Savoie, 2006. Français. <tel-00414653>

HAL Id: tel-00414653

https://tel.archives-ouvertes.fr/tel-00414653

Submitted on 9 Sep 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

pour obtenir le grade de docteur de l'Université de Savoie

Spécialité : Logique mathématique

présentée par Frédéric RUYER

Titre :

Preuves, Types et Sous-Types

---

soutenue le 30 Novembre 2006 devant le jury composé de

Emmanuel CHAILLOUX	Rapporteur
René DAVID	Directeur de thèse
Gilles DOWEK	Rapporteur
Pascal MANOURY	Rapporteur
Michel PARIGOT	
Christophe RAFFALLI	Directeur de thèse
Laurent REGNIER	

Université de Savoie

REMERCIEMENTS...

A mes maîtres : tout d'abord mes directeurs de thèse, René David et Christophe Raffalli, ainsi que mes professeurs de D.E.A., spécialement Paul Rozière et Chantal Berline pour leurs intuitions, leur exigence et leur accueil, ainsi qu'aux rapporteurs de cette thèse pour leur lecture attentive.
A ma famille : mes parents et grands-parents pour tout ce que le leurs dois, mes beaux-parents spécialement pour nous avoir accueillis lors de notre arrivée en Savoie, ma femme et mes enfant, de m'avoir soutenu et supporté pendant quatre longues années trop souvent perdu dans mes pensées !
Aux amis : Thomas et Yasmina pour m'avoir poussé vers cette aventure, à Laurent et Yoyo pour leurs encouragements, aux doctorants et camarades Thomas, Patrick, Serge, Yves, Khelifa, Didier,Gwenaël . . .pour la bonne humeur et les coups de mains.

Table des matières

Chapitre 1

Introduction

[...] ô mathématiques concises, par l'enchaînement rigoureux de vos propositions tenaces et la constance de vos lois de fer, vous faites luire, aux yeux éblouis, un reflet puissant de cette vérité suprême dont on remarque l'empreinte dans l'ordre de l'univers.
I. Ducasse, Comte de Lautréamont *"Les chants de Maldoror"*

La mathématique semble jouir d'un prestige immense : elle est la "reine des sciences", la matrice féconde d'où jaillissent des vérités indubitables, la pierre d'achoppement d'un pan gigantesque de la connaissance humaine. C'est à elle, à ses progrès et à sa capacité merveilleuse de synthétiser à l'aide de quelques signes des idées complexes, que l'on doit les progrès de nombreuses autres sciences.
Cependant, pour reprendre Bertrand Russell, *"Les mathématiques sont une science dans laquelle on ne sait jamais de quoi on parle, et où l'on ne sait jamais si ce que l'on dit est vrai"*; cette citation, non dénuée d'humour, traite d'un point à la mode au tournant du XIX^{e} siècle, et qui a hanté le XX^{e} siècle : le problème des fondements. Si l'on veut fonder les mathématiques sur un enchaînement de propositions rigoureux basé sur une axiomatique, il importe tout d'abord de se pencher justement sur cette axiomatique, et de la rendre la plus convaincante possible. La théorie des ensembles de Zermelo-Fraenkel [31], base des mathématiques modernes, semble correspondre à cet objectif : on n'a jusqu'à présent pas réussi à en dériver de paradoxe[1]. Ensuite, il faut se doter également d'une formalisation rigoureuse du raisonnement : c'est l'objet de la théorie de la démonstration. Celle ci peut sembler relever de la métamathématique[2], mais sa formalisation, son développement important, les problèmes complexes qu'elle pose, ainsi que ses implications pratiques en lien avec l'informatique, en font une branche à part entière des mathématiques.
Ainsi, tout comme les mécaniques classique, quantique et relativiste interagissent avec les développements de l'analyse et de l'algèbre, l'informatique fournit-elle à la logique et à la théorie de la démonstration des problèmes fondamentaux. Cette comparaison pourrait sem-

[1] Mais cela n'exclut pas qu'il en existe, puisque nous n'avons pas non plus de modèle de cette théorie
[2] Dans le sens de discours sur les mathématiques

bler audacieuse, car les mécaniques ont trait directement à nos tentatives de modélisation de la nature (c'est l'objet de la physique), alors que l'informatique n'est qu'un outil ; ceci dit, un outil sophistiqué ne remplit pas qu'une fonction utilitaire, mais porte à s'interroger sur les phénomènes que sa fonction permet d'observer, qu'elle nous fait apparaître. Par exemple, le microscope a permis d'observer quelques animalcules insoupçonnés, la lunette astronomique l'avance du périhélie de Mercure, etc...On objecterait encore que l'informatique semble, elle, en circuit fermé : à bien y regarder, un ordinateur n'est jamais qu'une sorte d'automate sophistiqué qui permet de faire fonctionner vite et bien des algorithmes, et par lui il ne s'agit d'observer que ce que l'on a créé par notre raison raisonnante, chose sans intérêt puisque l'on sait bien ce que l'on veut créer et comment le créer. Mais c'est justement ici le point intéressant : observer nos raisonnements pousse à regarder comment on peut les modéliser, comment exprimer ce que l'on veut, et si il n'y aurait pas derrière tout ça quelques principes ou lois à découvrir[3]. Une des découvertes surprenantes du domaine est justement qu'en un sens, c'est l'activité de raisonner sur des objets qui crée elle même les objets dont elle parle[4].
Se situant dans ce paradigme, cette thèse s'intéresse à l'aspect immatériel (on dit plus souvent logiciel, par opposition à matériel) de l'informatique : elle porte sur l'étude d'un langage d'algorithmes, sorte de langue rudimentaire servant à décrire des algorithmes que l'on souhaite éventuellement par la suite voir exécuter par une machine (le faisant passer du statut de langage d'algorithmes à celui de langage de programmation). Plus précisément, cette thèse étudie un système formel de démonstration associé à un langage d'algorithmes, explore son expressivité et le rattache à des concepts issus du monde de la programmation "réelle". Nous allons donc présenter ici brièvement quels sont les liens de la théorie de la preuve avec les mathématiques et l'informatique, avant d'aborder plus précisément la théorie de la démonstration, situer notre travail dans son contexte, et donner le plan de la thèse.

1.1 Introduction informelle

1.1.1 Preuves et preuves formelles

Preuve : Ce qui sert à établir la véracité d'un acte, d'une chose ou l'exactitude d'une allégation ou d'un fait.[5]
Une preuve, dans le cadre du droit, s'applique au délicat problème de la détermination de la responsabilité, la culpabilité ou l'innocence d'un sujet de droit. En tant qu'êtres raisonnables et épris d'équité, nous nous refusons à considérer qu'un faisceau de présomptions ou l'intime conviction d'un juge suffise à fournir une preuve : nous attendons un enchaînement rigoureux de faits et d'arguments.

[3]Il serait d'ailleurs presqu'ironique que de constater qu'un des aboutissements technologiques les plus fins du génie humain porte à s'interroger sur les plus élémentaires briques du raisonnement.
[4]cf. infra la correspondance de Curry-Howard.
[5]source : http ://www.taq.gouv.qc.ca/quelques-definitions/quelques-definitions.jsp

De même, en mathématiques, on pourrait penser que l'on est capable de se forger une opinion personnelle sur la véracité d'un théorème. Si cela reste possible pour ce qui est de certains théorèmes bien établis et vérifiés par quelques générations (comme le théorème fondamental de l'arithmétique), il devient humainement extrêmement délicat de le vérifier pour des résultats qui mettent en oeuvre de grandes quantités de savoirs pointus, et des calculs qui nous échappent (l'article [4] de Henk Barendregt et Freek Wiedjik donne un bon aperçu de cette évolution). D'individuelle, la preuve devient collective et repose à la fois sur la confiance entre des agents humains, et sur des ordinateurs. Citons pèle-mêle :

- La preuve de la conjecture de Kepler par Thomas Hales ([15] pour une présentation générale de la preuve), qui utilise des calculs informatiques, a été examinée pendant 4 ans par une équipe de 12 référés qui sont persuadés à 99% de la correction de la preuve. La vérification de cette preuve par des méthodes formelles est un sujet de recherches actif dans le cadre du projet *Flyspeck*[6].
- Les preuves du théorème des 4 couleurs, la dernière en date étant celle de Georges Gonthier qui utilise l'assistant de preuves CoQ, reposent sur la confiance attribuée aux machines et à leur implémentation.
- "L'énorme théorème" de classification des groupes finis simples repose sur de nombreuses publications (plusieurs milliers de pages au total).

La vérification de telles preuves requiert donc un effort qu'un humain ne peut effectuer seul. Les machines deviennent alors de précieux auxiliaires qui ne sont pas de simples calculatrices, mais des "penseurs" automatiques, nous déchargeant de vérifications fastidieuses.

Un problème analogue se trouve dans le développement tentaculaire de l'informatique : on ne compte plus les "bogues" qui ne se résument pas à de simples plantages d'ordinateurs personnels, mais peuvent affecter le fonctionnement de collectivités. Citons les plus célèbres :

- Panne du système de contrôle aérien de l'aéroport de Londres Heathrow (2004,2000, ...)
- Panne du système de réservation de la SNCF en 2004.
- Panne du réseau d'un grand opérateur français de la téléphonie mobile en 2004.
- Crash de la fusée Ariane 5 en 1996.
- Erreur de calcul d'un célèbre microprocesseur en 1994.

Le besoin de développer des méthodes pour vérifier le bon fonctionnement des programmes, spécialement dans les domaines dits critiques où des vies humaines ou des intérêts collectifs sont en jeu, se fait donc clairement sentir. Par bon fonctionnement, il s'agit de réaliser deux étapes : premièrement, il faut définir précisément ce qu'on veut que le programme fasse (ce qu'on appelle les *spécifications*), et ensuite il faut vérifier que le programme satisfait ces spécifications.

Pour cela, différentes approches sont utilisées, et nous distinguons les méthodes informelles des méthodes formelles. Les méthodes informelles servent à guider un développeur ou une équipe de développeurs dans l'élaboration d'un logiciel :

- Les règles de génie logiciel qui concernent les méthodologies de développement, les règles

[6] http ://www.math.pitt.edu/ thales/flyspeck/index.html

d'écriture de programmes, le management ...
- Les tests servent à vérifier le bon fonctionnement dans la majorité des cas.

On voit que ces méthodes, bien qu'apportant certainement une meilleure qualité dans le développement, n'apportent aucune *certitude*.
Les méthodes formelles utilisent, elles, des méthodes mathématiques :
- Le model checking étudie l'évolution de systèmes dans le temps à l'aide de la logique temporelle.
- L'interprétation abstraite, qui permet de dégager des propriétés générales de programmes.
- La méthode B [1], issue de la logique de Hoare [27], permet de certifier des programmes écrits dans un style impératif.
- Les assistants de preuve (PVS,CoQ, Nqthm, Mizar, Isabelle-HOL, PhoX, ...) permettent de décrire des langages et prouver formellement des propriétés de programmes (voir [5] pour une présentation des assistants de preuve).

Notre approche se situe dans la dernière catégorie : nous élaborons un langage de démonstration avec son système logique, de façon à prouver des propriétés de programmes. Elle se situe dans la continuité de l'histoire de la théorie de la démonstration, que nous présentons maintenant.

1.1.2 Théorie de la démonstration, λ-calcul et types

L'histoire de la logique et de l'étude du raisonnement est très ancienne, elle remonte à Aristote (-384 - -322) qui a inventé la figure du syllogisme (voir par exemple [44] ou [20] pour un aperçu de l'histoire de la logique). Cependant, son étude symbolique est assez récente. Le premier à avoir donné un formalisme pour le raisonnement est David Hilbert (1862-1943) (voir [17] pour une présentation), mais la formulation la plus "naturelle" en tant que séparant les hypothèses des conclusions est la déduction naturelle de Gerhard Gentzen (1909-1945) (voir [19] pour une présentation de la logique naturelle et du calcul des séquents). Ces recherches étaient notamment motivées par le point 2 du programme de 23 questions que Hilbert avait formulées en 1900 :

> *est-il possible de donner une preuve formelle de la consistance des axiomes de l'arithmétique ?*

Dans le domaine de ce qui était alors la recherche de l'expressivité des langages formels, et dans la continuité du programme de Hilbert, cherchant à caractériser les fonctions récursives - identifiées[7] avec ce qui est possible de calculer -, Alonzo Church (1903 - 1995) invente dans les années 30 le λ-calcul, sorte de langage d'algorithmes élémentaire. Avec l'invention des premiers ordinateurs, la théorie de la récursivité est devenue une branche de la théorie des langages de programmation, le λ-calcul devenant un de ceux-là.
La représentation physique des données dans la mémoire des machines, a apporté la notion de *type* de données qui permet de distinguer, par exemple, les booléens des entiers et des

[7] C'est ce qu'on appelle la "Thèse de Church", constatation empirique de la convergence de diverses façons de modéliser le calcul (machines de Turing, λ-calcul, fonctions récursives ...)

chaînes de caractères. Ceux-ci permettent de décrire des types plus complexes : un test de parité aura par exemple le type noté "entiers→booléens", prenant en entrée un entier, et renvoyant le booléen correspondant à sa parité.
Il fallut apparemment attendre les années 60 pour que les mathématiciens Haskell Curry (1900-1982) et William Alvin Howard s'aperçoivent qu'il est en fait possible d'interpréter les programmes comme des preuves et les types comme des propositions : c'est ce qu'on appelle la correspondance (ou analogie, ou isomorphisme) de Curry-Howard[28]. Cette correspondance est un véritable athanor[8] : elle effectue la transmutation d'une preuve écrite en déduction naturelle en un programme dont le type (la spécification) "est" l'énoncé que l'on a prouvé. Le résultat n'est pas anodin, car il réconcilie la forme et le fond du discours : alors que l'étude du raisonnement *per se* semblait ne pas se soucier des dénotations des objets sur lequel il raisonne (comme une figure de syllogisme fait abstraction de son contenu), on retombe par Curry-Howard sur une description des objets dont on parle par le fait même d'en avoir parlé. C'est ce qu'on appelle aussi le *contenu algorithmique* d'une preuve.
Cette correspondance permet ainsi de programmer sans en avoir conscience : les assistants de preuve permettent d'extraire d'un raisonnement le programme correspondant, sans efforts et en toute confiance. Cependant, la démarche usuelle, pragmatique, de la programmation est inverse : on programme un algorithme tout en ayant en tête la "preuve" de sa correction[9].
Notre approche se situe dans ce sens : nous bâtissons un langage fonctionnel, fortement inspiré du langage Caml[14] de la famille ML, tel que l'on puisse d'une part écrire les algorithmes, énoncer leurs spécifications, puis vérifier qu'ils les satisfont.
Après cette présentation générale de l'histoire et des idées qui concernent notre domaine, nous allons préciser un peu nos propos en présentant plus précisément notre domaine de recherche et en détaillant notre contribution.

1.2 État de l'art

1.2.1 Systèmes de types primitifs

Comme nous l'avons vu précédemment, l'objectif d'un système de type est d'exprimer le sens des algorithmes.
Un système de type est associé à une logique, aussi nous pouvons donner en exemple celui d'une logique où le seul connecteur serait le "ET" ($\wedge$). Le langage d'algorithme associé serait limité à la paire (x, y) et aux premières et secondes projections (π_1 et π_2). Un type est donc ici :
– Soit une constante atomique (P,Q,...)
– Soit une conjonction ($A \wedge B$) de deux types A et B

[8] casserole des alchimistes.

[9] On peut penser aux formules de Ramanujan(1887-1920), mathématicien indien qui fournissait ses résultats sans preuve et dont les travaux fournissent encore du travail pour les justifier.

Un programme est donc ici :
- Soit une constante atomique (x_A,x_B,...), dont on suppose qu'elles appartiennent aux types sous-notés.
- Soit une paire (u, v) de deux programmes u et v
- Soit une première (resp. une seconde) projection d'un programme $\pi_1(u)$ (resp. $\pi_2(u)$)

Ceci nous donne les règles suivantes :

$$\frac{}{x_A : A}\, Ax \qquad \frac{p : A \qquad q : B}{(p, q) : (A \wedge B)}\, \wedge_I$$

$$\frac{p : (A \wedge B)}{\pi_1(p) : A}\, \wedge_E^g \qquad \frac{p : (A \wedge B)}{\pi_2(p) : B}\, \wedge_E^d$$

Les processus de calcul sous-jacents, qui correspondent à une simplification de ma preuve sont :

$$\pi_1((u, v)) \succ u \quad \text{et} \quad \pi_2((u, v)) \succ v$$

Les deux propriétés, faciles à établir, de ce système de type, sont :

1. Préservation du type par réduction : si on peut dériver $p : A$ et si $p \succ q$, alors on peut dériver $q : A$
2. Normalisation : si on peut dériver $p : A$, alors la réduction de p s'arrête.

Si on s'autorise de plus à aller réduire sous les paires, la deuxième propriété devient une propriété de normalisation forte, c'est à dire que toute réduction de p s'arrête.

Un système de type plus usuel et plus intéressant est basé sur le constructeur $\rightarrow$, qui symbolise l'implication. Dans ce cas, le langage sous-jacent est le λ-calcul dont les termes sont :
- Soit une variable (x, y, ...)
- Soit un terme t dans lequel on a abstrait sur une variable x : $\lambda x.t$.
- Soit un terme u appliqué à un autre terme v : $u(v)$.

Pou écrire les règles de typage, une façon commode est de raisonner sur des séquents de la forme $\Gamma \vdash t : A$, Γ étant un contexte formé d'une liste de déclarations de types de variables ($x : A, y : B, \ldots$), le signe $\vdash$ un séparateur, et $t : A$ la conclusion :

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\, Ax$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \to_I \qquad \frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{u(v) : B} \to_E$$

La règle de réduction associée est :

$$\lambda x.t(u) \succ t[x := u]$$

Le système , bien que beaucoup plus expressif que le précédent, en conserve les propriétés de préservation du type par réduction et de forte normalisation.

1.2.2 Polymorphisme, types dépendants et PTS

Le langage de types restreint au seul constructeur $\to$ est assez pauvre, et d'autres systèmes ont été élaborés pour permettre d'exprimer :
- des types dépendants de types, comme les types paramétrés `List[nat]` (listes d'entiers), `List[Bool]` (listes de booléens), ...
- des termes dépendants de types
- types dépendants de termes, comme les tableaux de taille un entier donné `Array[n]`, ...

On se donne un ensemble de "genres" (noté $Kind$ dans la littérature), et on suppose que "l'ensemble des types" est un genre (ce qui est noté $Type : Kind$).
Le principe est de remplacer la flèche $\to$ par des produits $\Pi x : A.B$, exprimant intuitivement le domaine des fonctions qui à un objet de A associent un objet de B, x pouvant apparaître dans B :

1. Si A et B sont des types, cela correspond à la flèche usuelle (termes dépendants de termes) : on considère que $\Pi x : A.B$ est un type : c'est la règle $(Type, Type, Type)$
2. Si A est un genre, et B est un type, cela correspond au polymorphisme (types dépendants de types pour le système F), on considère encore que $\Pi x : A.B$ est un type : c'est la règle $(Kind, Type, Type)$.
3. Si A est un type et B est un genre, cela correspond aux types dépendants de termes, et on considère que $\Pi x : A.B$ est un genre : c'est la règle $(Type, Kind, Kind)$.
4. Si A est un genre et B est un genre, on considère que $\Pi x : A.B$ est un genre (types dépendants de types si A et B sont $Type$) : c'est la règle $(Kind, Kind, Kind)$.

Si on se donne le premier point, les trois points suivants peuvent être vus comme les trois dimensions d'un cube appelé "cube de Barendregt" (voir [37] pour une présentation).
Il est possible de rajouter d'autres formations de termes, en ajoutant d'autres espèces d'objets que $Kind$ et $Type$, on obtient alors des Systèmes de Types Purs ("PTS"), mais il

est important de bien en cerner les règles pour éviter de tomber dans l'inconsistance ; c'est ce qui se passe avec le système U^-, qui comporte les règles :
- $Sort : Kind$ et $Type : Sort$
- Possibilités de produits : $(Sort, Sort, Sort)$, $(Kind, Sort, Sort)$, $(Type, Type, Type)$, et $(Sort, Type, Type)$.

Le système le plus riche du cube est le calcul des constructions. Cependant, le système ne permet pas de séparer les parties d'une preuve informatives (correspondant au type de donnée désiré) de celles non-informatives (correspondant à la preuve que l'objet désiré a la propriété voulue).

1.2.3 Omission de contenu non-algorithmique, et sous-typage

Christine Paulin [42] a remédié à ce problème en ajoutant au PTS du calcul des constructions un nouvel habitant $Prop : Kind$. On a donc les mêmes règles de formation des produits, mais on peut distinguer dans un terme de preuve la "partie qui calcule" de type un objet de type $Type$, de celle qui "ne calcule pas", de type un objet de type $Prop$.
Par exemple, si on souhaite décrire le type paramétré par par A et P des objets de type A qui ont une propriété P, soit qui habite le type de CC :

$$\lambda A : Type.\lambda P : A \to Type.\Pi K : Type.(A \to (Px) \to K) \to K$$

il devient, si l'on ne s'intéresse qu'au type de données A :

$$\lambda A : Type.\lambda P : A \to Prop.\Pi K : Type.(A \to (Px) \to K) \to K$$

Une autre approche consiste à prendre directement un terme de preuve issu d'un système de typage, puis à l'élaguer pour n'en garder que l'information intéressante : ce sont les techniques de "pruning" développées par Stefano Berardi [6] et élargies au système F dans [9].
Le sous-typage consiste à modéliser la relation d'inclusion entre types. Dans les systèmes de types avec sous-typage, on distingue les relations de typage (de conclusion $t : A$) de celles de sous-typage (de conclusion $A \subset B$), et la règle permettant d'utiliser une relation de sous-typage est :

$$\frac{\Gamma \vdash u : A \qquad \Gamma \vdash A \subset B}{t : B}$$

Citons le système $F_{<:}$ (ou $F - sub$) [12] qui étudie les relations de sous-typage dans le système F. Ce système a été étendu dans [13] en ajoutant au système de types des enregistrements. Cette relation est largement exploitée dans les langages de programmation avec les concepts de modules et d'objets : nous y reviendrons dans cette introduction.

1.2.4 Types inductifs

Les langages de programmation, et les mathématiques, utilisent également les types inductifs définis comme les plus petits ensembles vérifiant certaines propriétés, ou clos par certaines fonctions, tels les entiers, les ordinaux ...
Ajouter un système de types inductifs de façon analogue à celle des langages de programmation fonctionnels nécessite l'utilisation d'un récurseur généralisé, ce qui n'est pas possible dans un cadre de normalisation forte ou faible d'un système.
Christine Paulin [42] a résolu ce problème dans le calcul des constructions inductives où un récurseur est construit pour chaque type défini, ces derniers étant codés dans le calcul des constructions.
Afin d'éviter le problème de l'inefficacité des types inductifs codés au second ordre, Michel Parigot [41] a proposé un système dans lequel les types inductifs sont des types de base du système. C'est ce qui est également proposé dans la thèse de Christophe Raffalli[48].

1.2.5 Modules, types nommés et types abstraits

Les langages de programmation de la famille ML utilisent la notion de modules (voir [14], [46]) pour structurer les développements : ce sont des enregistrements appelés structures dans lesquels chaque champs dépend des champs précédents, et qui sont associés à des types qu'on appelle leurs signatures. Le système de modules permet également de nommer des types et d'utiliser des types abstraits qui sont dans ce contexte des types nommés dont la signature masque la définition. Les modules ne sont pas vus comme des programmes ordinaires, mais comme des façons d'organiser les programmes.
La modélisation des types abstraits en tant que types existentiels [38] permet d'intégrer ceux-ci dans les programmes ordinaires, mais le nommage des des types est un problème plus délicat, traité dans le cas des variables (i.e. sans véritable nommage, mais avec des projections restreintes aux cas où les programmes sont des variables du contexte) dans [11] où il ne pose pas d'ambiguïté.
Cependant, l'utilisation des modules comme des programmes ordinaires est un trait assez intéressant car il permet de les paramétrer par des programmes ordinaires, et donc de les modifier en cours de calcul. Claudio Russo [52] a proposé un tel système, mais son système a été prouvé inconsistant (via un contre-exemple dans l'implémentation de Moscow ML) par Derek Dreyer [21], qui propose avec Karl Crary et Robert Harper [22] un formalisme pour résoudre ce problème.
Les signatures des modules à la ML sont déjà une ébauche de spécification, permettant de s'assurer d'une certaine façon de la correction des programmes, mais ne permettant pas d'écrire des spécifications fines.
Dans le formalisme "EML" (extended ML) développé à Edinburgh (voir par exemple la présentation de Kahrs, Sanella et Tarlecki [29]), il est possible d'ajouter dans les signatures des propriétés des programmes, et de les prouver. Ce formalisme souffre cependant d'une limitation, en ce sens qu'il ne dispose pas de fondements théoriques pour prouver sa consistance.

1.3 Contribution et plan de la thèse

Les systèmes de types issus de la logique (F [24], $AF2$ [32], CCI [42],...) présentent cet avantage de cerner précisément les types de données, mais par contre n'autorisent que des programmes fortement terminants (en particulier, ils ne tolèrent pas d'opérateur de point fixe général) : ils sont donc trop "rigides". Les systèmes de types issus de l'informatique (pour les langages fonctionnels) sont, par contre, trop "laxistes" : certains (ML,Haskell,...) permettent de donner, par exemple, le type "entier" à des expressions qui ne terminent pas, ou à des exceptions.
Nous souhaitons combiner dans notre approche les avantages des deux techniques : d'une part, pouvoir être certain de la forme d'un résultat lorsqu'il est typé, et d'autre part avoir une expressivité proche des langages de programmation réels.
Nous nous basons sur le système ST (initiales de sous-typage) de Christophe Raffalli ([50],[49]), qui utilise l'omission de contenu algorithmique et le sous-typage, et dont la particularité est de considérer un jugement de sous-typage comme une propriété logique, permettant ainsi de l'intégrer dans les types.
Le système ST satisfait aux critères précédents : d'une part, on est sûr de la forme d'un résultat lorsqu'il est typé, et d'autre part on sait que son expressivité est très forte (notamment via le théorème 47 de [49] qui établit sa complétude vis à vis de la réalisabilité formelle [32]). Les exemples développés dans les articles et dans cette thèse permettront d'étayer ce propos.
Notre contribution se situe sur deux plans :

- Sur le plan théorique, nous dégageons une sémantique plus générale de ST, complétons la preuve de préservation du type, et affinons l'étude de son expressivité en y exprimant des propriétés nouvelles que nous détaillons ci-dessous.
- Sur le plan des applications, nous montrons comment obtenir sur la base de ST un langage fonctionnel avec spécifications, comportant de nombreux traits de programmation :
  - Un riche ensemble de types de données : paires, types sommes, enregistrements, types inductifs et co-inductifs, types abstraits, polymorphisme.
  - Un langage de modules du premier ordre comprenant des formules dans les signatures.

  De plus, la correction de ce langage est en très grande partie certifiée sur machine.

Au niveau des types inductifs et co-inductifs, l'intérêt est que les codages sont faits en interne dans le système, sans recourir à une extension.
Les 4 premiers chapitres concernent l'étude théorique du système ST :

- Le chapitre 2 présente la sémantique qui sous-tend le système ST.
- Le chapitre 3 donne les règles du système, et établit sa correction par adéquation dans les modèles vus au chapitre de sémantique.
- Le chapitre 4 contient notre preuve de préservation du type dans ST.
- Nous étudions dans le chapitre 5 l'expressivité de ce système de types sous deux angles :
  - une extension simple du système (non-trivialité) permet d'y montrer son incompatibilité avec la logique classique, et une forme d'incomplétude.
  - bien que le système ne soit pas normalisable, nous montrons qu'il permet tout de même d'exprimer *en interne*, par le sous-typage, les propriétés de normalisabilité et

de résolubilité.

Les 4 derniers chapitres concernent le langage de programmation proprement dit :

- Nous commençons par donner dans le chapitre 6 une extension du système ST, et donnons la représentation de quelques types de données.
- Le chapitre 7 est une description d'un langage de programmation fonctionnel typé proche de ML, avec des spécifications.
- Nous décrivons dans le chapitre 8 un langage de modules avec spécifications qui ont un statut du premier ordre (c'est à dire qu'ils peuvent se manipuler comme des programmes)[10].
- Nous donnons ensuite la preuve des énoncés de correction concernant le langage, et terminons par un chapitre donnant quelques pistes de recherche.

[10]Les chapitres 7 et 8 peuvent se lire indépendamment

Chapitre 2

Préliminaires sémantiques

Nous présentons ici les notions mathématiques générales qui sous-tendent les notions informatiques présentées dans les chapitres suivants. L'objectif est de rendre plus généraux les raisonnements qui portent sur l'étude de notre langage, qui pourraient ainsi se généraliser à l'étude d'autres objets. Après une introduction situant notre approche sémantique par rapport aux autres, nous présentons dans une première section des résultats "classiques" sur les treillis (notamment l'induction et la coinduction, voir par exemple [34]), en mettant l'accent sur les notions d'éléments premiers et d'éléments compacts, ce qui nous semble nouveau. Nous voyons également comment le schéma de compréhension peut s'exprimer dans ce cadre. Dans une seconde section, nous voyons comment ces résultats peuvent s'appliquer dans le cadre d'espaces de parties d'un ensemble saturées par une relation. Enfin, dans une troisième section, qui cerne plus précisément ce dont on aura besoin par la suite, nous voyons comment construire dans une certaine généralité, à partir des notions vues dans la première section, des modèles du λ-calcul qui ont à la fois un sens dénotationnel et opérationnel (car la réduction y est interprétée). Bien que l'introduction se réfère à des notions propres à l'informatique, les deux sections suivantes (2.2 et 2.3) sont d'un niveau assez élémentaire, et sont lisibles par tout mathématicien même dépourvu de culture dans ce domaine.

2.1 Approches sémantiques de l'algorithmique

La froide et pure logique, s'intéressant au calcul de la valeur de vérité d'énoncés construits dans tel ou tel langage formalisé, s'incarne dans la réalité (mathématique) par le biais de modèles dans lesquels on donne un sens aux énoncés. Précisons. Un ensemble, souvent fini - tout au moins engendrable mécaniquement - d'énoncés, duquel on déduit d'autres énoncés s'appelle une *théorie* (théorie des ensembles, théorie des groupes, arithmétique, ...). Un objet mathématique dans lequel on interprète les énoncés d'une théorie par des faits vrais, s'appelle un *modèle* de cette théorie (voir par exemple [17] ou [19] pour une présentation précise) : c'est ainsi que, comme en linguistique - d'où provient d'ailleurs le terme - on parle de *sémantique* d'une théorie.

Effectivement, dans le cadre de la linguistique, on distingue la *syntaxe* (étude des mots et des phrases, ou signifiants) de la *sémantique* (étude des concepts, ou signifiés). Cette analogie tient à ce que l'intuition mathématique repose sur une collection de signifiés, objets mentaux dont nous admettons "l'existence" (1, 2, 3 ..., un point, une droite, une sphère, ...) ou plutôt dont nous partageons, par un accord mutuel basé sur notre culture et notre expérience, une connaissance tacite.
L'approche sémantique apporte donc à la pure logique :
- un solide étayage pour les théories, établissant leur consistance par le biais des modèles.
- un précieux guide dans les démonstrations : notre esprit interprète en permanence les énoncés qu'il manipule.
- des preuves d'impossibilité de prouver certains énoncés, en bâtissant des modèles qui les réfutent.

Dans le domaine qui nous intéresse ici, l'objectif est d'une part de construire un langage d'algorithmes[1], c'est à dire :
- Un ensemble de mots, qu'on appelle programmes.
- Des règles de réécriture sur ces mots, qui décrivent comment les programmes s'évaluent.

D'autre part, nous voulons être capable d'exprimer des propriétés des programmes dans un langage lui aussi formalisé. Les approches sémantiques du domaine consistent :
- à interpréter le langage de programmation et ses lois.
- à interpréter le langage des propriétés de programmes.

Nous présentons à présent ces deux approches.

2.1.1 Sémantiques des langages

Remarque: Cette approche n'étant pas directement liée à notre travail, nous en faisons une présentation succinte.
On distingue usuellement deux sémantiques : l'*opérationnelle*, qui étudie les lois d'évaluation du langage, l'objet mathématique étant le langage lui même, et la *dénotationnelle*, qui cherche à interpréter les mots du langage dans un autre ensemble mathématique, de manière à pouvoir en cerner certaines propriétés.
Nous présentons ici ce qui concerne ce langage simple qu'est le λ-calcul. Rappelons succintement sa syntaxe : on se donne un ensemble dénombrable de variables V, et un terme du λ-calcul est :
- soit une variable $x \in V$
- soit *l'application* de deux termes u et v, notée $(u \quad v)$
- soit *l'abstraction* d'une variable $x \in V$ dans un terme t, notée $\lambda x.t$

On dit qu'une occurrence d'une variable x est liée si elle apparaît dans un terme de la forme $\lambda x.t$.

Notation 1 *On notera Λ l'ensemble des λ-termes, quotienté par l'α-équivalence qui est la relation décrivant le renommage correct des variables liées.*

[1] on rencontre plus usuellement le terme de langage informatique, ou de programmation.

Sa règle de réécriture de base consiste en la contraction d'un radical (ou redex[2]), qui est une application dont le membre de gauche est une abstraction :

$$\underbrace{(\lambda x.t \quad u)}_{\text{radical}} \succ_{rad} \underbrace{t[x := u]}_{\text{contracté}}$$

$t[x := u]$ représentant la substitution correcte de x par u à toutes les occurrences de x liées par le λ (pour une présentation plus précise, notamment des notions de variables liées, de renommage et de substitutions, voir par exemple [32], [34], [7], [3] ...).
La contraction d'un radical est à rapprocher de l'application d'une fonction à un argument, qui "donne" un résultat. Par exemple, soit f la fonction qui à un entier n associe $n+2$. On la note usuellement $n \mapsto (n+2)$, et on peut aussi la noter $\lambda n.n+2$. L'application de f à un argument est "égale" à son réduit, dans le sens où $f(E) = (n+2)[n := E] = E+2$ pour toute expression arithmétique E.
Tout comme à l'intérieur d'un calcul algébrique on s'autorise à remplacer une sous-expression par son résultat, on étend la contraction en définissant les notions de β_0- et de β-réduction :

Définition 1 *La β_0-réduction, notée $\succ_{\beta_0}$ est définie par :*

$$\frac{t \succ_{rad} t'}{t \succ_{\beta_0} t'}$$

$$\frac{t \succ_{\beta_0} t'}{\lambda x.t \succ_{\beta_0} \lambda x.t'} \qquad \frac{t \succ_{\beta_0} t'}{(t \quad u) \succ_{\beta_0} (t' \quad u)} \qquad \frac{t \succ_{\beta_0} t'}{(u \quad t) \succ_{\beta_0} (u \quad t')}$$

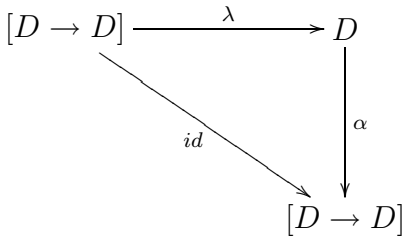
La β-réduction, notée $\succ_{\beta}$ est définie par :

$$\frac{}{t \succ_{\beta} t}$$

$$\frac{t \succ_{\beta_0} t'}{t \succ_{\beta} t'} \qquad \frac{t \succ_{\beta} t' \qquad t' \succ_{\beta} t''}{t \succ_{\beta} t''}$$

Partant, il est ainsi naturel de chercher des modèles où la β réduction s'interprète comme l'égalité, et où les termes peuvent s'interpréter comme des fonctions.
Dana Scott [53] a introduit la *théorie des domaines*, qui permet d'interpréter les λ-termes à la fois comme des éléments d'un domaine D et comme des fonctions de $D \to D$. On utilise en fait un sous-ensemble $[D \to D]$ de $D \to D$, qui sera la partie de $D \to D$ correspondant aux fonctions "codables" par des termes, et se caractérise en tant qu'espace de fonctions continues sur D pour une topologie (D sera un ensemble ordonné muni de certaines propriétés, nous renvoyons à [34],[32] pour de plus amples détails).
L'idée de la représentation est la suivante :

[2]Contraction de "reducible expression"



On définit deux fonctions $\alpha : D \to [D \to D]$ et $\lambda : [D \to D] \to D$ telles que $\alpha o \lambda = id$.
Et on définit par induction une interprétation I_v des λ-termes, qui dépend d'une valuation $v : V \to D$ des variables :
– $I_v(x) = v(x)$ si $x \in V$
– $I_v(t(u)) = \alpha(I_v(t))(I_v(u))$
– $I_v(\lambda x.t) = \lambda(d \mapsto I_{v[x:=d]}(t))$
Modulo la vérification que $d \mapsto I_{v[x:=d]}(t)$ est continue, cela définit une interprétation, et on a, par $\alpha o \lambda = id$, et par des lemmes techniques sur les substitutions :

$$\begin{aligned} I_v(\lambda x.t(u)) &= (\alpha o \lambda(d \mapsto I_{v[x:=d]}(t)))(I_v(u)) \\ &= I_{v[x:=I_v(u)]}(t) \\ &= I_v(t[x := u]) \end{aligned}$$

ce qui montre que l'on obtient bien un modèle de la β-équivalence.
Citons les modèles P_ω, D_∞. Les applications connues de cette théorie sont l'étude d'extensions consistantes du λ-calcul (déductions supplémentaires ...), et des arguments de réfutation (il n'existe pas de terme qui a telle ou telle propriété ...) : voir par exemple [32], [7] pour des précisions.
Nous allons maintenant présenter la sémantique des énoncés, et comment nous parvenons à retrouver une forme de sémantique des termes par son biais.

2.1.2 Sémantique des énoncés

Les propositions définies dans un langage formalisé sont interprétées usuellement par leur valeur de vérité : Vrai ou Faux. La sémantique dite de Brouwer-Heyting-Kolmogorov ([30], [26],...) propose une autre approche de la valeur de vérité : la valeur d'une proposition est l'ensemble de ses preuves, du moment que ces preuves sont formalisées. Une technique liée à ces idées est la réalisabilité de Kleene (voir par exemple [42] pour une présentation). Donnons un exemple dans le cadre restreint où le langage est le sous-ensemble des formules du calcul propositionnel avec pour seul connecteur $\to$, une formalisation arborescente simple des preuves en déduction naturelle étant donnée par les règles du λ-calcul simplement typé (qui correspondent à la logique minimale) :

$$\frac{}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i}\, Ax \quad (i \in \{1; \ldots; n\})$$

$$\frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : F}{x_1 : A_1, \ldots, x_{n-1} : A_{n-1} \vdash \lambda x_n.t : A_n \to F} \to_I$$

$$\frac{x_1 : A_1, \ldots, x_n : A_n \vdash t : F \to G \qquad x_1 : A_1, \ldots, x_n : A_n \vdash u : F}{x_1 : A_1, \ldots, x_n : A_n \vdash (t \quad u) : G} \to_E$$

L'idée est qu'une preuve de $F \to G$ est un procédé de calcul permettant d'obtenir, à partir de toute preuve de F, une preuve de G, la β-réduction devenant un procédé d'élimination de coupures (succession de $\to_I$ et de $\to_E$). On voit de plus que, les preuves étant codées par des λ-termes, suivre l'idée d'identifier une formule avec ses preuves, revient donc à interpréter une formule par un ensemble de λ-termes.
Ainsi, si on se donne une valuation v de l'ensemble des constantes propositionnelles dans les parties de λ, une notion d'interprétation I_v des formules dans l'ensemble des parties de Λ peut être défini par :
– $I_v(A) = v(A)$ si A est une constante.
– $I_v(F \to G) = \{t; \forall u \in F, (tu) \in G\}$.
On définit pour un λ-terme t le prédicat "t réalise F", noté $t \vDash F$, par $t \in I_v(F)$, et on veut le résultat :

$$\text{si} \vdash t : F \quad \text{alors} \quad t \vDash F$$

Une condition suffisante pour cela est que l'interprétation des constantes soit dans un ensemble de parties E tel que :

$$\begin{array}{c}\text{pour tout } A, B \in E, \text{ pour tout } t \in B \\ \text{si } u[x := t] \in A, \text{ alors } (\lambda x.u \quad t) \in A\end{array}$$

(c'est ce qu'on appelle une propriété de saturation).
Le résultat reste vrai pour des sémantiques analogues dans de nombreux systèmes de typage (systèmes T de Gödel [24], F de Girard [24], AF_2 de Krivine [32], ...).
Dans le système ST, la sémantique joue un rôle central, et comme il s'agit de la base sur laquelle nous bâtissons notre travail, nous avons choisi de l'étudier un peu précisément : de propriétés de termes et de formules, ce qui nous intéresse devient les propriétés d'ensembles de termes.
Souhaitant nous dégager au maximum des propriétés propres aux ensembles de λ-termes, nous allons nous placer dans un cadre plus abstrait, qui concerne les ensembles ordonnés. Nous nous intéresserons donc aux propriétés d'ensembles ordonnés aptes à être des candidats pour raisonner sur la réalisabilité, et en particulier qui contiennent des éléments irréductibles (que nous appellerons premiers). Dans une première partie, nous nous penchons sur le schéma de compréhension et les ensembles inductifs et coinductifs auxquels nous apportons un regard orienté vers les éléments premiers. Nous montrons ensuite que

notre catégorie d'ensembles ordonnés correspond en fait aux ensembles de parties saturées. Enfin, dans une troisième partie, nous voyons comment ajouter certaines propriétés à ces ensembles, pour obtenir, d'une part, une interprétation de $\rightarrow$, et d'autre part, ce qui est pour nous une surprise, une interprétation des λ-termes eux mêmes, les notions de β et d'η réductions devenant des inégalités.

2.2 Treillis complets premiers

2.2.1 Rappels sur les ensembles ordonnés

Définition 2 *Un ensemble ordonné est un couple $(E, \leq)$ tel que $\leq$ soit une relation :*
- *réflexive : pour tout $x \in E$, $x \leq x$.*
- *transitive : pour tous $x, y, z \in E$, si $x \leq y$ et $y \leq z$, alors $x \leq z$.*
- *antisymétrique : pour tous $x, y \in E$, si $x \leq y$ et $y \leq x$, alors $x = y$.*

Définition 3 *Soit $(E, \leq)$ un ensemble ordonné, et $A \subset E$.*
- *Un élément $x \in E$ **minore** (ou est un minorant de) A si pour tout $a \in A$, $x \leq a$.*
- *Un élément $x \in E$ **majore** (ou est un majorant de) A si pour tout $a \in A$, $a \leq x$.*
- *$a \in A$ est un **plus petit élément** de A si a minore A.*
- *$a \in A$ est un **plus grand élément** de A si a majore A.*
- *A a une **borne supérieure** (ou un sup) si l'ensemble de ses majorants est non vide et a un plus petit élément, que l'on note $\vee A$.*
- *A a une **borne inférieure** (ou un inf) si l'ensemble de ses minorants est non vide et a un plus grand élément, que l'on note $\wedge A$.*

Définition 4
- *Un **sup-treillis complet** est un ensemble ordonné $(E, \leq)$ qui possède un plus petit élément (noté $\bot$) et tel que tout sous ensemble a un sup.*
- *Un **inf-treillis complet** est un ensemble ordonné $(E, \leq)$ qui possède un plus grand élément (noté $\top$) et tel que tout sous ensemble a un inf.*

Les deux définitions d'inf- et sup- treillis complet étant équivalentes, nous donnons la définition suivante pour un treillis complet :

Définition 5 (Treillis complet) *Un **treillis complet** est un quadruplet $(E, \leq, \bot, \top)$ tel que :*
- *$(E, \leq)$ soit un ensemble ordonné,*
- *tout sous ensemble de E ait un sup et un inf,*
- *$\bot = \wedge E$ et $\top = \vee E$.*

Exemple 1
- *$(P(D), \subseteq, \emptyset, D)$ est un treillis complet pour tout ensemble D.*
- *L'ensemble des ouverts (ou des fermés) d'une topologie définie sur un ensemble X est un treillis complet (on remarque que dans ce dernier exemple, l'inf d'une famille d'ouverts n'est pas l'intersection de cette famille -qui peut être un fermé- mais son intérieur).*

2.2.2 Éléments premiers

Définition 6 (Éléments premiers, éléments compacts) *Soit $(E, \leq)$ un ensemble ordonné. On dit que $p \in E$ est **premier** si :*
– $p \neq \bot$
– pour tout $A \subset E$, si $p \leq \vee A$, alors il existe $a \in A$ tel que $p \leq a$.
Si $e \in E$, on note $P_e = \{p; p$ est premier et $p \leq e\}$.
*On dit que $k \in E$ est **compact** si :*
– k est premier
– Pour tout p premier, si $k \leq p$, alors $k = p$
Si $e \in E$, on note $K_e = \{k; k$ est compact et $k \leq e\}$.

Remarque: La notion d'élément premier est à rapprocher de la notion de primalité dans l'ensemble ordonné $(\mathbb{N}, divise)$: si $A = \{e_1; \ldots; e_n\}$ est un ensemble fini d'entiers, $\vee A = PPCM(e_1, \ldots, e_n)$. Il est clair que p est premier si et seulement si pour tout A fini, si $p \leq \vee A$, alors il existe $a \in A$ tel que $p \leq a$.
La notation suivante sert à exprimer des raccourcis pour les prédicats restreints aux éléments premiers ou compacts :

Notation 2
– On note $\forall x : a.P(x)$ pour $\forall x \in P_a.P(x)$
– On note $\forall x :_k a.P(x)$ pour $\forall x \in K_a.P(x)$

Définition 7 *Un **ensemble ordonné** est **premier** si pour tout $e \in E$, $e = \vee P_e$.*
*Un **ensemble ordonné** est **compact** si il est premier et si tout élément premier est compact.*

Comme nous aurons souvent à parler de treillis compacts ou de treillis premiers, nous avons la notation suivante :

Notation 3 *Un treillis complet premier (resp. compact) est noté TCP (resp. TCK). Dans un treillis complet, on notera P_E (resp. K_E) l'ensemble de ses éléments premiers (resp. compacts) (à la place de $P_\top$ (resp. $K_\top$)).*

Dans un TCP, la relation d'ordre jouit de la propriété suivante :

Proposition 1 *Soit E un TCP, et $a, b \in E$. On a :*

$$a \leq b \Longleftrightarrow \forall x : a.(x \leq b)$$

preuve:
– $\Longrightarrow$: est immédiat, car si $x \leq a$, alors $x \leq b$
– $\Longleftarrow$: si $\forall x : a.(x \leq b)$, alors $\{x; x \in P_a \wedge x \leq b\} = P_a$. Or, $\vee\{x; x \in P_a \wedge x \leq b\} \leq b$ par définition de la borne supérieure, et donc $\vee P_a \leq b$. Comme E est un TCP, $a = \vee P_a$, ce qui nous donne le résultat.

□

2.2.3 Schéma de compréhension dans les treillis complets

On appellera dans cette section prédicat sur E tout énoncé à un paramètre dans E.

Définition 8 *Soit E un treillis complet, et Φ un prédicat sur E. On appelle* ***schéma de compréhension*** *de Φ, noté $SC(\Phi)$, l'élément de E défini par :*

$$SC(\Phi) = \vee\{x \in P_E; \Phi(x)\}$$

Le fait suivant justifie la dénomination de schéma de compréhension :

Fait 2 *Soit E un treillis complet, et Φ un prédicat sur E. Alors :*

$$\forall k :_k \top.(k \leq SC(\Phi) \Longrightarrow \Phi(k))$$

preuve: Si k est compact, alors k est premier, donc si $k \leq \vee\{x \in P_E; P(x)\}$, alors il existe a tel que $a \in P_E$ et $\Phi(a)$ et $k \leq a$. Comme k est compact, on en déduit que $k = a$ et donc $\Phi(k)$. □

Un corollaire de ce fait est la proposition suivante, qui est très intuitive :

Proposition 3 *Soit E un treillis complet, $a \in E$ et Φ un prédicat sur E.*

1. *Si E est un TCP, alors : $\forall x : a.\Phi(x) \Longrightarrow a \leq SC(\Phi)$*
2. *Si E est un TCK, alors : $\forall x : a.\Phi(x) \Longleftrightarrow a \leq SC(\Phi)$*

preuve:

1. Est un corollaire de la proposition 1 page 19 : en effet, si $\forall x : a.\Phi(x)$, alors il est clair que $\forall x : a.x \leq SC(\Phi)$.
2. Le 1. nous donne déjà un sens, l'autre sens nous est donné par le fait que dans un TCK, $\forall x : a.\Phi(x) \Leftrightarrow \forall x :_k a.\Phi(x)$, le fait précédent nous donnant immédiatement que $a \leq SC(\Phi) \Rightarrow \forall x :_k a.\Phi(x)$.

□

2.2.4 Axiome du choix dans les treillis complets

Notation 4 *Nous adoptons dans cette section les notations suivantes :*
- *Si A et B sont deux ensembles, on notera $A \rightarrow B$ l'ensemble des applications de A dans B*
- *Si $f \in A \rightarrow B$, on notera aussi $x \mapsto f(x)$ pour f.*

On rappelle maintenant la définition de l'opérateur de description définie, ou fonction de choix :

Définition 9 (Fonction de choix) *Soit F un ensemble et $\mathcal{P}(F)$ l'ensemble des prédicats sur F. On appelle fonction de choix, notée Δ, toute fonction de $\mathcal{P}(F)$ dans F telle que :*

$$\textit{pour tout } P \in \mathcal{P}(F), \textit{ s'il existe un } x \in F \textit{ tel que } P(x) = 1, \textit{ alors } P(\Delta(P)) = 1$$

Sous forme de diagramme sagittal, on a donc :

$$\begin{array}{rccl} \Delta : & \mathcal{P}(F) & \rightarrow & F \\ & P & \mapsto & \textit{un } f \in F \textit{ tel que s'il existe un } x \in F \textit{ tel que } P(x) = 1, \textit{ alors } P(f) = 1 \end{array}$$

Dans la suite, on suppose fixée une fonction de choix Δ_F pour tout ensemble F. Le fait suivant illustre l'utilité potentielle d'une fonction de choix dans un treillis complet :

Fait 4 *Soit E un treillis complet et $(F_i)_{i\in I}$ une famille d'ensembles. Alors, pour tout $i \in I$, il existe une fonction, notée $._i$ telle que :*
– $._i$ a pour domaine $(F_i \rightarrow E) \times P_E$ et pour codomaine F_i
– Pour tout $\phi \in F_i \rightarrow E$, pour tout $e \in P_E$:

$$\textit{si} \quad e \leq \vee\{\phi(f); f \in F_i\}, \quad \textit{alors} \quad e \leq \phi(._i(\phi, e))$$

Sous forme de diagramme sagittal, on a donc :

$$\begin{array}{rccl} ._i : & (F_i \rightarrow E) \times P_E & \rightarrow & F_i \\ & (\phi, e) & \mapsto & \textit{un } f \in F_i \textit{ tel que } e \leq \phi(f) \textit{ si } e \leq \vee\{\phi(f); f \in F_i\} \end{array}$$

preuve: Soit $\phi \in (F_i \rightarrow E)$ et $e \in P_E$. On pose $P_{(\phi,e)}$ le prédicat $e \leq \phi(f)$. Si $e \leq \vee\{\phi(f); f \in F_i\}$, alors comme e est premier, il existe $f \in F_i$ tel que $e \leq \phi(f)$. Il suffit donc de poser $._i(\phi, e) = \Delta_{F_i}(P_{(\phi,e)})$. □

Notation 5 *On notera $e._i\phi$ pour $._i(\phi, e)$.*

Le fait précédent nous amène à donner un nom à l'opérateur . :

Définition 10 (Projection) *Soit E un treillis complet et $(F_i)_{i\in I}$ une famille d'ensembles. Alors, pour tout $i \in I$, la fonction $._i$ définie par le fait précédent est appelée projection d'indice i.*

Le lecteur averti est certainement conscient à ce point que notre but est de généraliser les types abstraits afin d'être capable de "projeter" n'importe quelle composante définie abstraitement dans un type (qu'elle soit un type, un type paramétré, un prédicat, un programme, etc ...). Cependant, si de multiples composantes sont définies abstraitement, on veut pouvoir les projeter toutes "simultanément", c'est pourquoi nous portons une attention particulière aux produits.

Projections dans les produits

Nous commençons par faire un rapide rappel sur un "isomorphisme" bien connu :

Fait 5 *Pour toute famille finie d'ensembles $(A_i)_{1\leq i\leq n}$ et pour tout ensemble B, $(A_1 \times \ldots \times A_n) \to B$ et $A_1 \to \ldots A_n \to B$ sont en bijection. On note $Curry((A_i)_{1\leq i\leq n}, B)$ et $Uncurry((A_i)_{1\leq i\leq n}, B)$ les bijections canoniques associés.*

preuve: On a simplement : $Curry((A_i)_{1\leq i\leq n}, B)(f) = a_1 \mapsto \ldots \mapsto a_n \mapsto f(a_1, \ldots, a_n)$, et : $Uncurry((A_i)_{1\leq i\leq n}, B)(g) = a \mapsto g(a_1)(a_2)\ldots(a_n)$.

□

On utilisera les notations suivantes :

Notation 6 – *Soit E et A deux ensembles, et m un entier strictement positif. On note : $A^m \to E$ pour $\underbrace{A \to \ldots \to A}_{m} \to E$.*

– *Soit E un ensemble, $(A_i)_{1\leq i\leq n}$ une famille finie d'ensembles, et m un entier strictement positif. La notation :*

$$(A_i)^m_{1\leq i\leq n} \to E \qquad \textit{représente :} \qquad A_1^m \to \ldots \to A_n^m \to E$$

Le fait précédent va nous servir pour montrer la proposition suivante :

Proposition 6 *Soit E un treillis complet, $(A_i)_{1\leq i\leq n}$ une famille finie d'ensembles, et m un entier strictement positif. Alors, il existe $n \times m$ fonctions notées $(.(i,j))_{1\leq i\leq n, \leq j\leq m}$ telles que :*

– *Pour tout (i, j) :*

$$.(i,j) : (((A_i)^m_{1\leq i\leq n} \to E) \times P_E) \to A_i$$

– *Pour tout $\psi \in (A_i)^m_{1\leq i\leq n} \to E$, pour tout $e \in P_e$, si :*

$$e \leq \vee\{\psi(a_{(1,1)})\ldots(a_{(n,m)}); a_{(i,j)} \in A_i \textit{ pour tous } i, j \textit{ tels que } 1 \leq i \leq n \textit{ et } 1 \leq j \leq m\}$$

alors :

$$e \leq \psi(.(1,1)(\psi, e))\ldots(.(n,m)(\psi, e))$$

preuve: Posons $F = \underbrace{A_1 \times \ldots \times A_1}_{m} \times \ldots \times \underbrace{A_n \times \ldots A_n}_{m}$, $\phi = Uncurry(\psi) : F \to E$, et $f = (a_{(1,1)}, \ldots, a_{(n,m)} \in F$ Prenons dans le fait 4, la famille (F_i) réduite à un élément qui est F. On a donc $e \leq \phi(.(\phi, e))$. Posons $.(i,j)(\psi, e) = (.(\phi, e))_{m\times(i-1)+j}$. On a $.(\phi, e) = (.(1,1)(\psi, e), \ldots, .(n,m)(\psi, e))$, et donc $e \leq Curry(\phi)(.(1,1)(\psi, e))\ldots(.(n,m)(\psi, e))$, et comme $Curry$ et $Uncurry$ sont des paires d'isomorphismes inverses :

$$e \leq \psi(.(1,1)(\psi, e))\ldots(.(n,m)(\psi, e))$$

□

Dans cette proposition, qui peut paraître artificielle, les ensembles $A_1, \ldots, A_n$ joueront plus tard le rôle des (interprétations de) sortes, et les indices $1, \ldots, m$ celui des noms sous lesquels on abstraira des objets.

2.2.5 Induction dans les treillis complets

Nous définissons l'élément inductif associé à une fonction sans condition aucune sur cette fonction.

Définition 11 (Elément inductif associé à une fonction) *Soit E un treillis complet et F une fonction de E dans E. On appelle* ***élément inductif*** *associé à F, noté μF, l'élément de E défini par :*

$$\mu F = \wedge\{x; \forall a \in E(a \leq x \Rightarrow F(a) \leq x)\}$$

Le fait suivant donne des propriétés basiques de l'ensemble inductif :

Fait 7 *Soit E un treillis complet et F une fonction de E dans E. On a :*

1. $\forall x(x \leq \mu F \Rightarrow F(x) \leq \mu F)$
2. $F(\mu F) \leq \mu F$
3. *Si F est croissante, alors μF est le plus petit point fixe de F*

preuve:

1. On suppose que $x \leq \mu F$. Par définition de la borne inférieure, cela signifie que x minore tout y qui vérifie $\forall a \in E(a \leq y \Rightarrow F(a) \leq y)$. Il suffit, pour montrer que $F(x) \leq \mu F$, de montrer que $F(x)$ a aussi cette propriété. Soit donc un y tel que $\forall a \in E(a \leq y \Rightarrow F(a) \leq y)$. Par hypothèse, x minore y, donc en substituant a par x dans la formule précédente, on en déduit que $F(x)$ minore y.
2. C'est immédiat en prenant $x = \mu F$ dans 1.
3. Il est immédiat de voir que si F est croissante, alors μF minore tous les points fixes de F, car pour tout point fixe i de F, pour tout x, si $x \leq i$, alors $F(x) \leq F(i) = i$. Il reste à montrer que μF est bien un point fixe, et d'après 2, il reste donc à montrer que $\mu F \leq F(\mu F)$. Il suffit pour cela de montrer que $\forall a \in E(a \leq F(\mu F) \Rightarrow F(a) \leq F(\mu F)$, ce qui est immédiat en utilisant 2.

□

Le troisième point du fait précédent peut laisser à penser que notre présentation n'apporte rien par rapport à la présentation traditionnelle du résultat de Tarski (voir par exemple [34]). Cependant, on constate dans le fait suivant que la croissance de F n'a pas d'influence sur le principe d'induction, si on se place dans un espace suffisamment restreint.

Proposition 8 (Principe d'induction) *Soit E un TCK, F une fonction de E dans E et P un prédicat sur E. On a :*

$$\forall a(\forall x : a.P(x) \Rightarrow \forall x : F(a).P(x)) \quad \Rightarrow \quad \forall x : \mu F.P(x)$$

preuve: On suppose que $\forall a(\forall x : a.P(x) \Rightarrow \forall x : F(a).P(x))$. En utilisant la proposition 3 page 20, on en déduit que $\forall a(a \leq SC(P) \Rightarrow F(a) \leq SC(P))$, d'où vient par le fait 7 page 23 $\mu F \leq SC(P)$, et finalement en réutilisant la proposition 3 page 20, $\forall x : \mu F.P(x)$. □

2.2.6 Co-induction dans les treillis complets

Nous procédons comme pour les ensembles inductifs : on définit d'abord l'élément co-inductif associé à une fonction.

Définition 12 (Elément co-inductif associé à une fonction) *Soit E un treillis complet et F une fonction de E dans E. On appelle* ***élément co-inductif*** *associé à F, noté νF, l'élément de E défini par :*

$$\nu F = \vee\{x; \forall a \in E(x \leq a \Rightarrow x \leq F(a))\}$$

On montre les propriétés basiques de l'élément co-inductif :

Fait 9 *Soit E un treillis complet et F une fonction de E dans E. On a :*

1. $\forall x(\nu F \leq x \Longleftrightarrow \forall y(\forall a(y \leq a \Rightarrow y \leq F(a)) \Rightarrow y \leq x))$
2. $\forall x(\nu F \leq x \Rightarrow \nu F \leq F(x))$
3. $\nu F \leq F(\nu F)$
4. *Si F est croissante, alors νF est le plus grand point fixe de F.*

preuve:

1. C'est évident par définition de la borne supérieure.
2. On suppose $\nu F \leq x$, ce qui donne par 1. $\forall y(\forall a(y \leq a \Rightarrow y \leq F(a)) \Rightarrow y \leq x))$. Il faut montrer, par 1. encore $\forall y(\forall a(y \leq a \Rightarrow y \leq F(a)) \Rightarrow y \leq F(x)))$. On suppose que $\forall a(y \leq a \Rightarrow y \leq F(a))$, on a donc $y \leq x$, qu'on réinjecte dans l'hypothèse, ce qui nous donne $y \leq F(x)$.
3. Immédiat par 2.
4. On suppose que F est croissante. Pour montrer que νF est un point fixe, il suffit de montrer que $F(\nu F) \leq \nu F$, ce qui est immédiat car alors $F(\nu F)$ vérifie que $\forall a \in E(F(\nu F) \leq a \Rightarrow F(\nu F) \leq F(a))$, qui est trivial par 2. et par croissance.

□

La proposition suivante donne deux principes de co-induction : le premier semble avoir peu d'intérêt pratique, et le second correspond à l'utilisation usuelle que l'on attend d'une co-induction dans le cas croissant :

Proposition 10 (Principes de co-induction) *Soit E un TCK, F une fonction de E dans E, et P un prédicat sur E. On a :*

1. $\forall y(\forall a(y \leq a \Rightarrow y \leq F(a)) \Rightarrow \forall x : y.P(x)) \Rightarrow \forall x : \nu F.P(x)$
2. *si F est croissante, alors :* $\forall y(y \leq F(y) \Rightarrow \forall x : y.P(x)) \Rightarrow \forall x : \nu F.P(x)$

preuve:

1. Est une application du fait précédent 1. à et de la proposition 3 page 20 2. à $SC(P)$.
2. Il suffit de vérifier que si F est croissante, $y \leq F(y) \Longleftrightarrow \forall a(y \leq a \Rightarrow y \leq F(a))$, ce qui est trivial.

□

2.3 Parties saturées

La notion de partie saturée, bien connue dans le domaine de l'informatique théorique, sert, pour parler grossièrement, à identifier un élément d'un ensemble avec tous les éléments qui sont en relation avec lui pour une certaine relation R ; dans le cas où R est une relation d'équivalence, un sous-ensemble usuel des parties saturées par R est celui formé par les classes d'équivalences. Nous présentons ici le cas général, et montrons qu'en fait les notions présentées dans la section précédente sont "les mêmes" que (c'est à dire sont isomorphes à) celles dont nous allons parler maintenant.

2.3.1 Définition et premières propriétés

Définition 13 (Parties saturées, singletons) *Soit X un ensemble et R une relation sur X. L'ensemble des **parties de X saturées par** R, noté $\mathcal{P}_R(X)$ est le sous ensemble de $\mathcal{P}(X)$ défini par :*

$$P \in \mathcal{P}_R(X) \iff \forall x \in P, \forall y \in X : x\ R\ y \Rightarrow y \in P$$

Si $A \subset X$, on note $Sat_R(A)$ la plus petite partie saturée par R qui contient A.
*Si $x \in X$, on appelle **singleton** associé à x l'ensemble $Sat_R(\{x\})$, que l'on notera aussi $\{x\}_R$.*
On note $\mathcal{S}_R(X)$ l'ensemble des singletons de $\mathcal{P}_R(X)$

Remarque: Dans le cas où R est une relation d'équivalence, on remarque immédiatement que l'ensemble des singletons s'identifie avec l'ensemble des classes d'équivalence.
On observe également une propriété élémentaire des saturés de parties, qui consiste à dire que le saturé d'une partie est atteint au bout d'un "nombre dénombrable" d'itérations :

Fait 11 *Soit E un ensemble, R une relation sur E et $A \subset E$. On définit la suite $(A_n)_{n\in\mathbb{N}}$ par :*
– $A_0 = A$
– $A_{n+1} = \{e; \exists a \in A_n : a\ R\ e\}$
Alors, $Sat_R(A) = \cup_{n\in\mathbb{N}} A_n$

preuve:

1. $\cup_{n\in\mathbb{N}} A_n$ est saturée : effectivement, si $x \in \cup_{n\in\mathbb{N}} A_n$, il existe $n \in \mathbb{N}$ tel que $x \in A_n$. Soit y tel que $x\ R\ y$. Par définition, $y \in A_{n+1}$, et donc $y \in \cup_{n\in\mathbb{N}} A_n$.
2. Soit B une partie saturée qui contient A. Il est clair que, pour tout $n \in \mathbb{N}$, $A_n \subset B$, donc $\cup_{n\in\mathbb{N}} A_n \subset B$.

Il suit de 1. et 2. que $\cup_{n\in\mathbb{N}} A_n$ est la plus petite partie saturée qui contient A. □

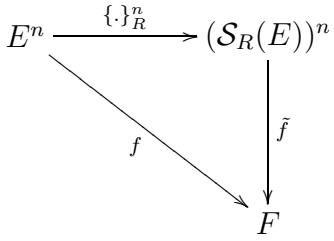
Passage aux singletons

Nous observons ici l'analogue du passage au quotient sur les classes d'équivalence.

Définition 14 (R-compatibilité) *Soit E et F deux ensembles, R une relation sur E, n un entier strictement positif et f une fonction de E^n dans F. On dit que f est R-compatible si pour tout $(a_1, \ldots, a_n)$, $(b_1, \ldots, b_n) \in E^n$ si pour tout $i = 1, \ldots, n$, $a_i \; R \; b_i$, alors $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$.*

Le fait suivant permet de définir des fonctions sur l'ensemble des singletons :

Fait 12 (Passage aux singletons) *Soit E et F deux ensembles, R une relation sur E, n un entier strictement positif et f une fonction R-insensible de E^n dans F. Alors, il existe une fonction $\tilde{f}$ de $(\mathcal{S}_R(E))^n$ dans F telle que le diagramme ci-dessous commute :*



$\{.\}_R^n$ étant la fonction $(a_1, \ldots, a_n) \mapsto (\{a_1\}_R, \ldots, \{a_n\}_R)$

preuve: Montrons que pour tout a, pour tout b, si pour tout $i = 1, \ldots, n$, $b_i \in \{a_i\}_R$, $f(b) = f(a)$. Cela est immédiat en utilisant la caractérisation de $\{a_i\}_R$ du fait 11 page 25 : si $b_i \in \{a_i\}_R$, alors il existe n_i tel que $b_i \in \{a_i\}_{R_{n_i}}$, et donc $f(b) = f(a)$. □

2.3.2 Isomorphismes avec les treillis complets premiers ou compacts

On prend une notation pour les suites d'éléments en relation les uns avec les autres :

Notation 7 *Soit E un ensemble, R une relation sur E, $x, y \in E$, et n un entier. On note $x \; R^n \; y$ s'il existe une suite finie $(x_i)_{0 \leq i \leq n}$ telle que :*
– $x = x_0$ et $y = x_n$.
– Pour tout entier $i \leq n$, $x_i \; R \; x_{i+1}$.
On a, en particulier, toujours $x \; R^0 \; x$.

Le fait suivant établit que tout ensemble de parties saturées est un TCP, et donne une caractérisation de la compacité :

Fait 13 *Soit X un ensemble et R une relation sur X.*

1. *$\mathcal{T}_R(X) = (\mathcal{P}_R(X), \subseteq, \emptyset, X)$ est un TCP, et $P_{\mathcal{T}_R(X)} = \mathcal{S}_R(X)$.*

2. $\{x\}_R$ *est un élément compact de* $\mathcal{T}_R(X)$ *si et seulement si*

$$\forall y(\exists n \in \mathbb{N} : y\ R^n\ x \Longrightarrow \exists p \in \mathbb{N} : x\ R^p\ y)$$

3. $\mathcal{T}_R(X)$ *est un TCK si et seulement si la clôture transitive de* R *est symétrique.*

preuve:

1. Le fait que cet ensemble soit un treillis complet est immédiat. On vérifie que les éléments premiers s'identifient avec les singletons :
   - $\mathcal{S}_R(X) \subset P_{\mathcal{T}_R(X)}$: soit $s = \{x\}_R$ un singleton, et $A \subset \mathcal{P}_R(X)$, tel que $s \subset \vee A = \cup_{a\in A} a$. On a donc $x \in \vee A$, donc il existe $a \in A$ tel que $x \in a$, ce qui entraîne $s \subset a$ car a est saturée.
   - $P_{\mathcal{T}_R(X)} \subset \mathcal{S}_R$. Soit $p \in P_{\mathcal{T}_R(X)}$, $A = \{\{\alpha\}_R; \alpha \in p\}$, et $a = \vee A$. Pour tout $\alpha \in p$, $\alpha \in a$ donc $p \subset a$, et donc il existe $\beta \in p$ tel que $p \subset \{\beta\}_R$. Il est immédiat que $\{\beta\}_R \subset p$ (car p est saturée), donc $p = \{\beta\}_R$.
2. – Soit $\{x\}_R$ compact. Soit y tel que $\exists n : y\ R^n\ x$. On a, par le fait 11 page 25, que $\{x\}_R \subset \{y\}_R$, et donc par définition, $\{x\}_R = \{y\}_R$, ce qui entraîne $\{y\}_R \subset \{x\}_R$, et donc en réutilisant le fait 11 page 25, que $\exists p : y\ R^p\ x$.
   - Soit x qui vérifie $\forall y(\exists n : y\ R^n\ x \Rightarrow \exists p : x\ R^p\ y)$, et y tel que $\{x\}_R \subset \{y\}_R$. Par le fait 11 page 25, on a $\exists n : y\ R^n\ x$, et donc $\exists p : y\ R^p\ x$, et donc encore par le fait 11 page 25, on a $\{y\}_R \subset \{x\}_R$, ce qui entraîne $\{x\}_R = \{y\}_R$.
3. Immédiat par 2.

□

Remarque: Le 3. du fait précédent revient aussi à dire que la clôture transitive de R est une relation d'équivalence partielle (PER).
Il faut cependant prendre garde à ne pas avoir de vision trop naïve des TCP. En particulier, dans un TCP, on peut avoir des éléments compacts qui ont des minorants stricts, comme le montre l'exemple suivant : $E = \{a; b; c\}$, avec $R = \{(a, b); (a, c)\}$. $\{a\}_R = E$ est compact, mais est strictement minoré par $\{b\}_R$, $\{c\}_R$ (qui ne sont pas compacts), et $\vee\{\{b\}_R; \{c\}_R\} = \{b; c\}$.
Il vient naturellement la question inverse du fait précédent : un TCP ou un TCK est il toujours isomorphe à un ensemble de parties saturées ? La réponse, positive, sera précisée par la proposition suivante, après que nous ayons précisé ce que nous entendons dans ce contexte par isomorphe.

Définition 15 *Soit* $(E, \leq, \bot, \top)$ *et* $(E', \leq', \bot', \top')$ *deux treillis complets, et* ϕ *une application de* E *dans* E'. *On dit que* ϕ *est un* ***morphisme de treillis complets*** *si :*

$$\phi(\vee A) = \vee\{\phi(a); a \in A\} \text{ pour tout } A \subset E$$

On dit que ϕ *est un* ***isomorphisme de treillis complets*** *si c'est un morphisme de treillis complets, qu'il est bijectif et que son inverse est un morphisme de treillis complets.*
S'il existe un morphisme de treillis complets entre $(E, \leq, \bot, \top)$ *et* $(E', \leq', \bot', \top')$, *on dit qu'ils sont* ***isomorphes****.*

Remarque: Le fait que ϕ soit un morphisme de treillis complets entraîne naturellement que ϕ est croissante, que $\phi(\bot) = \bot'$ et $\phi(\top) = \top'$.

Proposition 14 *Les TCP et les TCK sont complètement caractérisés par les ensembles de parties saturées :*

1. *Tout TCP est isomorphe à un ensemble de parties saturées.*
2. *Tout TCK est isomorphe à un ensemble de parties saturées par une relation d'équivalence.*

preuve: Soit $(E, \leq, \bot, \top)$ un treillis complet. On définit les fonctions ϕ et ψ par :

$$\begin{array}{rccl} \phi: & E & \rightarrow & \mathcal{P}_{\geq}(P_E) \\ & e & \mapsto & Sat_{\geq}(P_e) \end{array}$$

$$\begin{array}{rccl} \psi: & \mathcal{P}_{\geq}(P_E) & \rightarrow & E \\ & P & \mapsto & \vee\{p; p \in P\} \end{array}$$

Il est immédiat que ces fonctions sont bien définies. Il faut vérifier que ce sont bien des morphismes de treillis complets (1.) entre $(E, \leq, \bot, \top)$ et $\mathcal{T}_{\geq}(P_E)$, puis qu'ils sont inverses l'un de l'autre (2.) :

1. – ϕ est un morphisme de treillis complets : il suffit de montrer que $\phi(\vee A) = \cup\{\phi(a); a \in A\}$ pour tout $A \subset E$.
   - $\phi(\vee A) \subset \cup\{\phi(a); a \in A\}$: soit $x \in \phi(\vee A) = Sat_{\geq}(P_{\vee A})$. Par le fait 11 page 25, et par transitivité de $\geq$, il existe $\alpha \in P_{\vee A}$ tel que $x \leq \alpha$. α étant premier, par définition de $P_{\vee A}$, il existe $a \in A$ tel que $\alpha \leq a$. Il s'ensuit que $x \leq a$, et donc $x \in Sat_{\geq}(P_a) = \phi(a) \subset \cup\{\phi(a); a \in A\}$.
   - $\cup\{\phi(a); a \in A\} \subset \phi(\vee A)$: soit $x \in \cup\{\phi(a); a \in A\}$. Il existe $a \in A$ tel que $x \in \phi(a)$, donc il existe $p \in P_a$ tel que $x \leq p$. Or, comme $a \in A$, on a $a \leq \vee A$ et donc $P_a \subset P_{\vee A}$. On a donc $p \in P_{\vee A}$, et donc $x \in Sat_{\geq}(P_{\vee A}) = \phi(\vee A)$.

   – ψ est un morphisme de treillis complets : il suffit de montrer que $\psi(\cup A) = \vee\{\psi(a); a \in A\}$ pour tout $A \subset \mathcal{P}_{\geq}(P_E)$.
   - $\psi(\cup A) \leq \vee\{\psi(a); a \in A\}$: soit $p \in \cup A$. Il existe $a \in A$ tel que $p \in a$, et donc $p \leq \vee\{p; p \in a\} = \psi(a)$, d'où $p \leq \vee\{\psi(a); a \in A\}$. Comme $\psi(\cup A) = \vee\{p; p \in \cup A\}$, il s'ensuit bien que $\psi(\cup A) \leq \vee\{\psi(a); a \in A\}$.
   - $\vee\{\psi(a); a \in A\} \leq \psi(\cup A)$ D'après la proposition 1 page 19, il suffit de montrer $\forall x : \vee\{\psi(a); a \in A\}.x \leq \psi(\cup A)$. Soit $x \in P_{\vee\{\psi(a); a \in A\}}$. Par définition, $x \leq \vee\{\psi(a); a \in A\}$, et donc il existe $a \in A$ tel que $x \leq \psi(a) = \vee\{p; p \in a\}$. Donc, il existe $p \in a$ tel que $x \leq p$. Or, comme $p \in a$, $p \in \cup A$, et donc il existe $p \in \cup A$ tel que $x \leq p$, il s'ensuit que $x \leq \vee\{p; p \in \cup A\} = \psi(\cup A)$.

   – Il s'agit maintenant de montrer que les deux morphismes sont des morphismes inverses :
   - Soit $e \in E$, on a $\psi(\phi(e)) = \vee\{p; p \in Sat_{\geq}(P_e)\}$. Comme pour tout $x \in Sat_{\geq}(P_e)$, il existe $p \in P_e$, tel que $x \leq p$ il s'ensuit que $\vee\{p; p \in Sat_{\geq}(P_e)\} \leq \vee\{p; p \in P_e\}$. Comme $P_e \subset Sat_{\geq}(P_e)$, il s'ensuit que $\vee\{p; p \in Sat_{\geq}(P_e)\} \geq \vee\{p; p \in P_e\}$. D'où $\psi(\phi(e)) = \vee\{p; p \in P_e\} = e$.

- Soit $P \in \mathcal{P}_{\geq}(P_E)$, on a $\phi(\psi(P)) = Sat_{\geq}(P_{\vee\{p;p\in P\}})$. Si $p \in P$, $p \leq \vee\{p; p \in P\}$, et donc $p \in P_{\vee\{p;p\in P\}} \subset Sat_{\geq}(P_{\vee\{p;p\in P\}})$: $P \subset Sat_{\geq}(\vee\{p; p \in P\})$. Si $q \in Sat_{\geq}(P_{\vee\{p;p\in P\}})$, on a donc qu'il existe $p \in P_{\vee\{p;p\in P\}}$ tel que $q \leq p$. Comme $p \leq \vee\{p; p \in P\}$, q aussi, et donc il existe $p \in P$ tel que $q \leq p$. Il s'ensuit par saturation de P que $q \in P$:$Sat_{\geq}(P_{\vee\{p;p\in P\}}) \subset P$.

2. Il suffit de vérifier que si E est un TCK, alors $\geq$ restreinte à P_E est une relation d'équivalence. La réflexivité et la transitivité sont triviales, et la symétrie découle de la définition de compact : en fait, cette relation est simplement l'égalité sur les compacts !

□

2.3.3 Un résultat de compacité pour les classes de λ -termes

On donne ici un petit résultat qui donne une caractérisation élémentaire des termes normaux et des termes qui "bouclent". On dit que $t \in \Lambda$ boucle si $t \succ_\beta t$ et si t n'est pas normal. La proposition suivante donne une caractérisation des singletons compacts dans l'ensemble des parties de Λ saturées par la β-expansion $\prec_\beta$ (noté $\mathcal{P}_{\prec_\beta}(\Lambda)$).

Proposition 15 *Soit $E = \mathcal{P}_{\prec_\beta}(\Lambda)$, et $t \in \Lambda$:*

$$\{t\}_{\prec_\beta} \text{ est compact } \iff t \text{ est normal ou } t \text{ boucle}$$

preuve: D'après le fait 13 page 26, $\{t\}_{\prec_\beta}$ est compact si et seulement si pour tout y, si $t \succ_\beta y$, alors $y \succ_\beta t$. Ceci est évidemment vrai pour les termes normaux. Si t n'est pas normal, alors par définition, t boucle. □

2.4 Treillis applicatifs

Définition 16 (Treillis applicatif) *Soit $(E, \leq, \bot, \top)$ un treillis complet, et @ une fonction de $E \times E$ dans E. On dit que @ commute aux sups si pour tout $A \subset E$ et pour tout $b \in E$:*
– $\vee\{a@b; a \in A\} = \vee A@b$
– $\vee\{b@a; a \in A\} = b@ \vee A$
*On appelle **treillis applicatif** un quintuplet $(E, \leq, \bot, \top, @)$ tel que $(E, \leq, \bot, \top)$ soit un treillis complet et @ commute aux sups. On note TA un treillis applicatif.*

On constate immédiatement que dans un treillis applicatif, @ est croissante, plus précisément :

Fait 16 *Soit $(E, \leq, \bot, \top, @)$ un treillis applicatif, et $a, b, c \in E$ tels que $a \leq b$. Alors :*

$$a@c \leq b@c \quad \text{et} \quad c@a \leq c@b$$

preuve: $a@c \leq \vee\{x@c; x \in \{a; b\}\} = (\vee\{x; x \in \{a; b\}\})@c = b@c$. □

L'objectif de cette section est de montrer qu'un treillis applicatif est une structure suffisamment riche pour y exprimer la β-expansion et la η-réduction. Plus précisément, nous allons montrer le résultat suivant :

Proposition 17 *Tout treillis applicatif contient un modèle de* $\Lambda_{\prec_\beta \cup \succ_\eta}$.

Nous allons définir ce que l'on entend par "un modèle de $\Lambda_{\prec_\beta \cup \succ_\eta}$", et avant cela, donner la définition de la η-réduction. Tout comme la β-réduction, il faut tout d'abord définir le contracté d'un radical.
La règle de réécriture consiste en la contraction d'un η-radical (ou η-redex), qui est une abstraction dont le sous-terme est une application, le membre de droite de l'application étant la variable abstraite, et son membre de gauche un terme dans lequel la variable abstraite n'est pas libre :

$$\underbrace{\lambda x.(t \quad x)}_{\text{radical } (x \text{ non libre dans } t)} \succ_{\eta-rad} \underbrace{t}_{\text{contracté}}$$

Définition 17 *La η_0-réduction, notée $\succ_{\eta_0}$ est définie par :*

$$\frac{t \succ_{\eta-rad} t'}{t \succ_{\eta_0} t'}$$

$$\frac{t \succ_{\eta_0} t'}{\lambda x.t \succ_{\eta_0} \lambda x.t'} \qquad \frac{t \succ_{\eta_0} t'}{(t \quad u) \succ_{\eta_0} (t' \quad u)} \qquad \frac{t \succ_{\eta_0} t'}{(u \quad t) \succ_{\eta_0} (u \quad t')}$$

La η-réduction, notée $\succ_\eta$ est définie par :

$$\frac{}{t \succ_\eta t} \qquad \frac{t \succ_{\eta_0} t'}{t \succ_\eta t'} \qquad \frac{t \succ_\eta t' \qquad t' \succ_\eta t''}{t \succ_\eta t''}$$

Définition 18 *Un ensemble ordonné $\mathcal{M} = (E, \leq)$ est un modèle de $\Lambda_{\prec_\beta \cup \succ_\eta}$ si il existe une fonction I de Λ dans $\mathcal{M}$ telle que :*
- *Si $u \succ_\beta v$, alors $I(u) \leq I(v)$*
- *Si $u \succ_\eta v$, alors $I(u) \geq I(v)$*

Il faut pour cela pouvoir interpréter l'abstraction, ce qu'on va faire après avoir interprété la flèche $\rightarrow$, qui correspond à la flèche des types :

Définition 19 *Soit $(E, \leq, \bot, \top, @)$ un treillis applicatif. Soit $a, b \in E$ et f une fonction de E dans E.On définit :*
- $a \rightarrow b = \vee\{y; y@a \leq b\}$
- $\Lambda f = \wedge\{x \rightarrow f(x); x \in E\}$

Lemme 18 *Pour tous $a, b, c \in E$, pour toute fonction f de E dans E :*

1. $a \leq (b \to c) \quad \Longleftrightarrow \quad (a@b) \leq c.$
2. $\Lambda f@a \leq f(a).$
3. $a \leq \Lambda(x \mapsto a@x).$

preuve:

1. – $\Leftarrow$: est évident, car si $a@b \leq c$, alors $a \in \{y; y@b \leq c\}$, et donc $a \leq \vee\{y; y@b \leq c\}$.
   – $\Rightarrow$: soit $a \leq b \to c$. Posons $Y = \{y; y@b \leq c\}$. On a $a \leq \vee Y$, et par le fait 16 page 29, on en déduit $a@b \leq \vee Y@b$. Comme @ commute aux sups, $\vee A@b = \vee\{y@b; y \in Y\}$. Or, pour tout $y \in Y$, $y@b \leq c$, et donc $\vee\{y@b; y \in Y\} \leq c$, d'où $a@b \leq c$.
2. Par 1., il suffit de montrer que $\Lambda f \leq a \Rightarrow f(a)$, ce qui est immédiat par définition de Λ.
3. Encore par 1., il suffit de montrer que pour tout x, $a@x \leq a@x$, ce qui est pour le moins clair.

□

Ce lemme est la clé pour montrer la proposition 17 page 30. On démontre cependant deux lemmes techniques avant de faire la preuve finale. Il faut tout d'abord définir les interprétations :

Définition 20 (Interprétation des λ-termes dans un treillis applicatif)
Soit E un treillis applicatif.
On appelle valuation sur E une fonction de l'ensemble des variables du λ-calcul dans E. Si $\mathcal{V}$ est une valuation, on définit $\mathcal{V}[x := e]$ la valuation égale à $\mathcal{V}$ sur les variables distinctes de x, et qui vaut e sur x.
Pour chaque valuation $\mathcal{V}$, on définit une interprétation $\mathcal{I}_\mathcal{V}$ des termes dans E par :
– $\mathcal{I}_\mathcal{V}(x) = \mathcal{V}(x)$ pour toute variable x.
– $\mathcal{I}_\mathcal{V}(uv) = \mathcal{I}_\mathcal{V}(u)@\mathcal{I}_\mathcal{V}(v)$.
– $\mathcal{I}_\mathcal{V}(\lambda x.t) = \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(t))$.
($e \mapsto f(e)$ désignant la fonction qui à e associe $f(e)$)

Le premier lemme technique est la propriété de substitutivité des interprétations :

Lemme 19 *Soit t,u et v des λ-termes, x,y deux variables distinctes, $e, f \in E$, et $\mathcal{V}$ une valuation. On a :*

1. $\mathcal{V}[x := e][y := f] = \mathcal{V}[y := f][x := e]$
2. *Si y n'est pas libre dans u,* $\mathcal{I}_{\mathcal{V}[y:=e]}(u) = \mathcal{I}_\mathcal{V}(u)$
3. $\mathcal{I}_\mathcal{V}(t[x := u]) = \mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_\mathcal{V}(u)]}(t)$

preuve:

1. Évident par définition.

2. Par induction sur la structure de u :
   - Si u est une variable, alors c'est évident.
   - Si u est une application, c'est immédiat par définition de l'interprétation et hypothèse d'induction.
   - Si u est une abstraction : $u = \lambda z.u'$:
     - Si $z = y$, alors

$$\begin{aligned} \mathcal{I}_{\mathcal{V}[y:=e]}(u) &= \Lambda(f \mapsto \mathcal{I}_{\mathcal{V}[y:=e][y:=f]}(u')) \\ &= \Lambda(f \mapsto \mathcal{I}_{\mathcal{V}[y:=f]}(u')) \end{aligned}$$

     - Si $y \neq z$, y n'est pas libre dans u' et :

$$\begin{aligned} \mathcal{I}_{\mathcal{V}[y:=e]}(u) &= \Lambda(f \mapsto \mathcal{I}_{\mathcal{V}[y:=e][z:=f]}(u')) \\ \text{par 1.} &= \Lambda(f \mapsto \mathcal{I}_{\mathcal{V}[z:=f][y:=e]}(u')) \\ \text{par H.I.} &= \Lambda(f \mapsto \mathcal{I}_{\mathcal{V}[z:=f]}(u')) \end{aligned}$$

3. Par induction sur la structure de t :
   - Si t est une variable, c'est immédiat.
   - Si t est une application, c'est immédiat par H.I.
   - Si t est une abstraction : $t = \lambda y.t'$
     - Si $y = x$:

$$\begin{aligned} \mathcal{I}_{\mathcal{V}}(t[x := u]) &= \mathcal{I}_{\mathcal{V}}(\lambda x.t'[x := u]) \\ &= \mathcal{I}_{\mathcal{V}}(\lambda x.t') \\ &= \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(t')) \\ &= \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_{\mathcal{V}(u)}][x:=e]}(t')) \\ &= \mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_{\mathcal{V}(u)}]}(t) \end{aligned}$$

     - Si $y \neq x$; on renomme éventuellement y pour éviter la capture de variables, ce qui donne $\mathcal{I}_{\mathcal{V}[y:=e]}(u) = \mathcal{I}_{\mathcal{V}}(u)$ (r) par 2. et :

$$\begin{aligned} \mathcal{I}_{\mathcal{V}}(t[x := u]) &= \mathcal{I}_{\mathcal{V}}(\lambda y.(t'[x := u])) \\ &= \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[y:=e]}(t'[x := u])) \\ \text{par H.I.} &= \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[y:=e][x:=\mathcal{I}_{\mathcal{V}[y:=e]}(u)]}(t')) \\ \text{par 1.} &= \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_{\mathcal{V}[y:=e]}(u)][y:=e]}(t')) \\ \text{par (r)} &= \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_{\mathcal{V}}(u)][y:=e]}(t')) \\ &= \mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_{\mathcal{V}}(u)]}(t) \end{aligned}$$

□

Le second lemme est un lemme de croissance :

Lemme 20 *Pour tout treillis applicatif E, pour toutes fonctions f et g de E dans E :*

$$\textit{Si } \forall e \in E(f(e) \leq g(e)) \textit{ alors } \Lambda f \leq \Lambda g$$

preuve: On suppose $\forall e \in E(f(e) \leq g(e))$. Soit $x \in E$. $x \to f(x) = \vee\{y; y@x \leq f(x)\}$ Or, si $y@x \leq f(x)$, on a par hypothèse $y@x \leq g(x)$. On en déduit que $\vee\{y; y@x \leq f(x)\} \leq \vee\{y; y@x \leq g(x)\}$, et donc $x \to f(x) \leq x \to g(x)$. D'où $\wedge\{x \to f(x); x \in E\} \leq \wedge\{x \to g(x); x \in E\}$. □

On fait maintenant la preuve de la proposition 17 page 30 :
preuve: On va montrer en fait : pour toute valuation V, pour tous termes u et v :
– Si $u \succ_{\beta_0} v$, alors $\mathcal{I}_{\mathcal{V}}(u) \leq \mathcal{I}_{\mathcal{V}}(v)$
– Si $u \succ_{\eta_0} v$, alors $\mathcal{I}_{\mathcal{V}}(u) \geq \mathcal{I}_{\mathcal{V}}(v)$
Le résultat pour pour leurs clôtures transitives réflexives résultant des propriétés de l'ordre.
On fait la preuve dans chaque cas par induction sur la structure de u.
Pour β_0 :
– Si u est une variable, $u \succ_{\beta_0} v$ est faux.
– Si u est une abstraction. $u = \lambda x.u'$ et $v = \lambda x.v'$ avec $u' \succ_{\beta_0} v'$. Par H.I., pour toute valuation V, $\mathcal{I}_V(u') \leq \mathcal{I}_V(v')$. On a donc, pour tout $e' \in E$ $(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(u'))(e') \leq (e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(v'))(e')$. On a donc, par le lemme 20 : $\Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(u')) \leq \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(v'))$, et donc $\mathcal{I}_V(u) \leq \mathcal{I}_V(v)$.
– Si u est une application : $u = (ab)$.
  – Si $v = (a'b)$ ou $v = (ab')$ avec $a \succ_{\beta_0} a'$ ou $b \succ_{\beta_0} b'$, ce résultat suit immédiatement de l'hypothèse d'induction et de la croissance de l'application.
  – Si $a = \lambda x.a'$ et $v = a'[x := b]$:

$$\begin{array}{rcl} \mathcal{I}_V(u) & = & \mathcal{I}_V(a)@\mathcal{I}_V(b) \\ & = & \Lambda(e \mapsto \mathcal{I}_{V[x:=e]}(a'))@\mathcal{I}_V(b) \\ \text{par le lemme 18.2} & \leq & \mathcal{I}_{V[x:=\mathcal{I}_V(b)]}(a') \\ \text{par le lemme 19.3} & = & \mathcal{I}_V(a'[x := b]) \\ & = & \mathcal{I}_V(v) \end{array}$$

Pour η_0 :
– Si u est une variable, $u \succ_{\eta_0} v$ est faux.
– Si u est une abstraction.
  – Si $u = \lambda x.u'$ et $v = \lambda x.v'$ avec $u' \succ_{\eta_0} v'$. Par H.I., pour toute valuation V, $\mathcal{I}_V(u') \geq \mathcal{I}_V(v')$. On a donc, pour tout $e' \in E$ $(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(u'))(e') \geq (e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(v'))(e')$. On a donc, par le lemme 20 : $\Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(u')) \geq \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(v'))$, et donc $\mathcal{I}_V(u) \geq \mathcal{I}_V(v)$.
  – Si $u = \lambda x.(u'x)$ avec x non libre dans u' et $v = u'$.

$$\begin{array}{rcl} \mathcal{I}_V(u) & = & \mathcal{I}_V(\lambda x.(u'x)) \\ & = & \Lambda(e \mapsto \mathcal{I}_{V[x:=e]}(u'x)) \\ \text{par définition} & = & \Lambda(e \mapsto \mathcal{I}_V(u')@e) \\ \text{par le lemme 18.3} & \geq & \mathcal{I}_V(u') \\ & = & \mathcal{I}_V(v) \end{array}$$

– Si u est une application : $u = (ab)$, $v = (a'b)$ ou $v = (ab')$ avec $a \succ_{\eta_0} a'$ ou $b \succ_{\eta_0} b'$, ce résultat suit immédiatement de l'hypothèse d'induction et de la croissance de l'application.

Si on pose $\mathcal{M} = \mathcal{I}_V(\Lambda)$, on a donc bien que $\mathcal{M}$ est un modèle de $\Lambda_{\prec_\beta;\succ_\eta}$. Ceci clôt la preuve de la proposition. □

Nous ne présentons pas d'application directe de la proposition 17 page 30. Nous constatons simplement qu'elle permet de construire, par des procédés élémentaires, une grande classe de modèles de la β-expansion et de la η-réduction. Les modèles les plus intuitifs peuvent être construits comme suit :

Soit X un ensemble, et f une fonction de $X \times X$ dans X. On pose @ la fonction de $\mathcal{P}(X)$ dans $\mathcal{P}(X)$ définie par :

$$a@b = \{f(\alpha, \beta); \alpha \in a, \beta \in b\}$$

On vérifie que @ commute au sup : Soit $A \subset \mathcal{P}(X)$, on a $\vee A = \cup_{a \in A} a$.

$$\begin{aligned}
\vee A@b &= \{f(\alpha,\beta); \alpha \in \vee A, \beta \in b\} \\
&= \{f(\alpha,\beta); \exists a \in A : \alpha \in a, \beta \in b\} \\
&= \{x; \exists\alpha\exists a\exists\beta(a \in A, \alpha \in a, \beta \in b, x = f(\alpha,\beta)\} \\
&= \{x; \exists a(a \in A, \exists\alpha\exists\beta(\alpha \in a, \beta \in b, x = f(\alpha,\beta)\} \\
&= \{x; \exists a(a \in A, x \in a@b)\} \\
&= \cup_{a \in A}(a@b) \\
&= \vee\{a@b; a \in A\}
\end{aligned}$$

On montre par exemple aisément que $I(\lambda x.x) = \{1\}$ dans le modèle où $X = \mathbb{N}$ et $f(n,m) = n \times m$ ...

2.5 Treillis λ-compacts

Nous allons terminer cette section sur les préliminaires sémantiques en introduisant une famille de structures qui va capturer l'essentiel de ce dont on aura besoin par la suite.

2.5.1 Définitions et premières propriétés

Définition 21 *Un treillis applicatif est λ-compact si il est non réduit à $\perp$ et si pour toute valuation qui interprète les variables du λ-calcul par des éléments compacts, pour tout terme t du λ-calcul, $\mathcal{I}_\mathcal{V}(t)$ est compact.*

Fait 21 *Tout treillis applicatif λ-compact contient un modèle de $\Lambda_{=_\beta,=_\eta}$*

preuve: C'est immédiat par définition : si $\mathcal{I}(u) \leq \mathcal{I}(v)$ et si $\mathcal{I}(u)$ et $\mathcal{I}(v)$ sont compacts, alors $\mathcal{I}(u) = \mathcal{I}(v)$ par définition de la compacité. □

2.5.2 Exemples

Treillis trivial

$E = \{\bot; \top\}$, muni de l'application définie par :

@	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

est un treillis λ-compact. La table de l'application est d'ailleurs également celle de la conjonction et de l'intersection.
Il est immédiat de vérifier que c'est un treillis applicatif. On montre par induction que l'image de tout λ-terme dans la valuation qui à toute variable associe $\top$ est compacte :
- Cas variable : c'est clair
- Cas application : c'est clair
- Cas abstraction : on montre plus généralement que si f est une fonction qui à $\top$ associe $\top$, alors $\Lambda(f) = \top$. En effet, $\Lambda(f) = (\top \to \top) \cap (\bot \to f(\bot))$. Or, $\top \leq (\top \to \top)$ et $\top \leq \bot \to x$ pour tout x donc $\top \leq \Lambda(f)$.

Supposons maintenant que E contienne au moins un élément compact entre $\top$ et $\bot$. Si cet élément est unique, alors on n'a pas $\top = \vee\{a\}$, et donc $\top$ n'est pas égal à la borne supérieure des compacts qui le minore, ce qui est absurde. Donc il y a au moins un autre élément distinct de $\top$.

Parties saturées de λ-termes

La nécessité que tout élément premier soit compact impose de se restreindre aux relations d'équivalence d'après la proposition 14 page 28. On s'intéresse donc aux ensembles de la forme $E = \mathcal{P}_R(\Lambda)$, avec R une relation d'équivalence, et on pose :

$$a@b = \{(uv); u \in a, v \in b\}_R$$

L'ordre est l'inclusion, on a un treillis complet par le fait 14 page 28.
Montrons que cela nous donne bien un treillis applicatif :
soit $A \subset E$. $\vee A = \cup_{a \in A} a$. soit $b \in E$
- Prenons $x \in (\vee A)@b$. Il existe $u \in \vee A$, $v \in b$ tels que $x \in \{(uv)\}_R$. Comme $u \in \vee A$, il existe $a \in A$ tel que $u \in a$, et donc il existe $a \in A$ tel que $x \in a@b$.
- Soit $x \in \vee\{a@b; a \in A\}$. Il existe $a \in A$ tel que $x \in a@b \leq \vee A@b$ par croissance.

On a bien un treillis applicatif.
Si on prend $R = \beta$, on n' a pas la compacité pour les images de λ-termes : effectivement, si on prend a une constante, on a $\{a\}_R \subset \mathcal{I}(\lambda x.(a)x)$ et $\{\lambda x.(a)x\}_R \subset \mathcal{I}(\lambda x.(a)x)$. Ainsi, l'interprétation de $\lambda x.(a)x$ n'est pas compacte.
Vérifions que cela fonctionne pour $R = \beta\eta$:
Il reste à montrer que l'interprétation de tout λ-terme t, sous l'hypothèse que la valuation des variables est dans les compacts, est compacte.
On procède par induction.

- si t est une variable, c'est trivial.
- si t est une application, c'est trivial aussi.
- si t est une abstraction : on suppose que $t = \lambda x.u$. $\mathcal{I}(t) = \Lambda(e \mapsto \mathcal{I}_{\mathcal{V}[x:=e]}(u)$ Soit a un compact. Par le lemme 18 page 31 $\mathcal{I}(t)@a \leq \mathcal{I}_{\mathcal{V}[x:=a]}(u)$, et donc est compacte par hypothèse d'induction. Prenons t_1 et $t_2 \in \mathcal{I}(t)$, et α une variable non libre dans t_1 et t_2, et posons $a = \{\alpha\}_R$. On a $(t_1\alpha)$ et $(t_2\alpha) \in \mathcal{I}_{\mathcal{V}[x:=a]}(u)$, qui est compact donc $(t_1\alpha) =_{\beta\eta} (t_2\alpha)$,et donc $\lambda\alpha(t_1\alpha) =_{\beta\eta} \lambda\alpha(t_1\alpha)$, d'où $t_1 =_{\beta\eta} t_2$. Et donc $\mathcal{I}(t)$ est compact.

Définition 22 (Modèle Standard) *On appellera modèle standard* $(\mathcal{P}_{\beta\eta}(\Lambda), \subset)$.

Autres modèles

Nous avons montré en fait que toute relation d'équivalence qui contient $\beta\eta$ donne un treillis λ-compact : le premier cas de modèle trivial en est un cas particulier. On peut aussi prendre un modèle de facilité (cf. par exemple la thèse d'Yves Bertini [8]) : R est dans ce cas la relation d'équivalence contextuelle engendrée par $\beta\eta$ et l'équation $\lambda x.x = (\lambda x.(x \quad x) \quad \lambda x.(x \quad x))$, et le modèle est non-trivial.

2.6 Conclusion

Nous avons présenté dans ce chapitre les notions sémantiques, assez naïves mais claires nous l'espérons qui vont sous-tendre l'étude théorique du système ST que nous présentons dans le chapitre suivant. Il sera utile au lecteur d'avoir en tête le modèle "standard" de réalisabilité qui correspond aux parties saturées par $\beta\eta$.

Chapitre 3

Le système ST

3.1 Présentation informelle

Le système ST est un formalisme de typage du λ-calcul pur qui permet de distinguer les parties d'une preuve avec ou sans contenu algorithmique et de faire du sous-typage. Nous proposons pour ce système une présentation légèrement différente de l'originale [50], [49] :

- nous utilisons des contextes ordonnés, qui permettent un contrôle simultané de la construction des termes et des preuves.
- pour faciliter la lecture, nous utiliserons deux notations :
  - Une λ-notation ($\lambda x : s.t$, $u(v)$, ...) pour les termes destinés à être interprétés sémantiquement (les types, les formules,...).
  - Une notation style machine à écrire, similaire à l'écriture d'un programme ML (`fun` x `->` t, `(`u v`)`, ...) pour les programmes[1] .
- nous séparons les règles des axiomes.
- nous remplaçons certains axiomes.
- l'adéquation est montrée pour des modèles de réalisabilité abstraits.

Nous commençons (section 3.2) par présenter les règles de bonne formation de contexte et de construction de termes. Puis, nous présentons les règles et axiomes dépourvus d'opérateurs (section 3.3), c'est à dire dont l'écriture ne fait apparaître que les constantes de base. Après avoir défini les opérateurs utiles, nous donnons les règles et axiomes qui utilisent ces opérateurs (3.4).Nous donnons ensuite une preuve de consistance du système par adéquation dans un modèle abstrait de réalisabilité tel que vu au chapitre 2 (section 3.4). Enfin, nous donnons une série de faits qui peuvent être qualifiés de "techniques", pour l'essentiel déjà énoncés dans [50] et [49], dont nous détaillons un peu plus les preuves et dont la lecture est profitable pour mieux saisir le fonctionnement du système.

[1] Nous attirons ici l'attention du lecteur sur les différentes notations :
- Ce qui s'appelle ici les termes, pour lesquels nous écrivons l'application sous la forme $u(v)$, généralisent les types (éléments du treillis) du chapitre précédent.
- Ce qui s'appelle ici les programmes (écrits en style machine à écrire) correspondent aux λ-termes du chapitre précédent.

3.2 Sortes, termes et contextes

3.2.1 Sortes

Les jugements les plus simples concernent les sortes ; on distingue deux sortes de base : τ (la sorte des types, ou propositions avec contenu algorithmique) et o (la sorte des formules, ou propositions sans contenu algorithmique), à partir desquelles on construit les autres sortes à l'aide de la flèche $\rightarrow$, ce qui donne les règles élémentaires suivantes :

$$\frac{}{\vdash \tau : *} \qquad \frac{}{\vdash o : *}$$

$$\frac{\vdash s_1 : * \qquad \vdash s_2 : *}{\vdash (s_1 \rightarrow s_2) : *}$$

Le jugement $\vdash s : *$ peut se lire simplement "s est une sorte".

3.2.2 Termes et contextes

Commençons par définir les termes, qui sont des λ-termes avec abstraction annotée par une sorte, soit "à la Curry" :

Définition 23 (Termes) *On se donne un ensemble de variables de termes V_t. Un **terme** est :*
- *Si $x \in V_t$, x est un terme.*
- *Si u et v sont deux termes, $u(v)$ est un terme.*
- *Si $x \in V_t$ et u est un terme et $\vdash s : *$, $\lambda x : s.u$ est un terme.*

Nous ne définissons pas les notions de variables liées, libres, et de substitution qui peuvent se trouver dans toute référence commune sur le λ-calcul ([32], [3], ...).

Définition 24 (Contexte) *On se donne un ensemble $\mathcal{V}$ variables appelées variables de programme. Un **contexte** est défini par :*
- *$[]$ est un contexte.*
- *Si Γ est un contexte, $x \in V_t$ et $\vdash s : *$, alors $\Gamma, x : s$ est un contexte.*
- *Si Γ est un contexte, $x \in \mathcal{V}$ et t est un terme, alors $\Gamma, x : t$ est un contexte.*
- *Si Γ est un contexte, et t est un terme, alors Γ, t est un contexte.*

En d'autres termes, pour éviter de jargonner inutilement, un contexte sera une liste ordonnée de paires *variable* : *sorte* ou *variable* : *terme* ou *terme*. Si Γ est un contexte, on appelle $V(\Gamma)$ l'ensemble des variables qui apparaissent dans Γ comme membre gauche d'une paire *variable* : *sorte* (ou *variable* : *terme*).
Nous définissons parmi les contextes et parmi les termes ceux qui sont licites : respectivement les contextes bien formés et les termes typés. Nous présentons progressivement

les règles de formation des contextes bien formés (qu'on note WF). Les premières règles servent à rajouter à un contexte une déclaration de variable sortée :

$$\frac{}{WF([])}$$

$$\frac{WF(\Gamma) \quad \vdash s : * \quad x \notin V(\Gamma)}{WF(\Gamma, x : s)}$$

On peut maintenant bâtir les termes *typés* ; on se donne un ensemble infini de couples *constante* : *sorte*, noté *Const* qui contient les éléments suivants :

- Constantes de propositions :
  - $\Rightarrow^o : o \to o \to o$
  - $\forall^o_s : (s \to o) \to o$ pour chaque sorte s
  - $\subset : \tau \to \tau \to o$
- Constantes de types :
  - $\Rightarrow^\tau : \tau \to \tau \to \tau$
  - $\forall^\tau_s : (s \to \tau) \to \tau$ pour chaque sorte s
  - $\twoheadrightarrow : o \to \tau \to \tau$

Notation 8 *On adopte les conventions d'écriture suivantes :*
- *Tout connecteur binaire $a \in \{\Rightarrow^o; \Rightarrow^\tau; \twoheadrightarrow\}$ sera noté en position infixe i.e. $a(x)(y)$ sera noté $(x \quad a \quad y)$.*
- *Un quantificateur $\xi \in \{\forall^o_s; \forall^\tau_s\}$ appliqué pourra être noté $\xi x.A(x)$ au lieu de $\xi \lambda x.A(x)$.*
- *Lorsque la sorte de $\Rightarrow$ sera implicite, on l'omettra pour alléger les écritures.*

Les règles de constructions des termes typés sont les règles du λ-calcul simplement typé :

$$\frac{WF(\Gamma) \quad x : s \in \Gamma}{\Gamma \vdash x : s} \qquad \frac{WF(\Gamma) \quad c : s \in Const}{\Gamma \vdash c : s}$$

$$\frac{\Gamma \vdash u : s_1 \to s_2 \quad \Gamma \vdash v : s_1 \quad \vdash s_1 \to s_2 : *}{\Gamma \vdash u(v) : s_2} \qquad \frac{\Gamma, x : s_1 \vdash t : s_2 \quad \vdash s_1 \to s_2 : *}{\Gamma \vdash \lambda x : s_1.t : s_1 \to s_2}$$

Les termes étant construits, on étend la notion de contexte bien formé avec les règles suivantes :

$$\frac{WF(\Gamma) \quad \Gamma \vdash P : o}{WF(\Gamma, P)} \qquad \frac{WF(\Gamma) \quad \Gamma \vdash A : \tau \quad x \notin V(\Gamma) \quad x \in \mathcal{V}}{WF(\Gamma, x : A)}$$

Un contexte sera donc composé :
– de déclarations de sortage (de la forme $x : s$ avec $s : *$)
– de déclarations logiques (de la forme P avec $P : o$)
– de déclarations de typage (de la forme $x : A$ avec $A : \tau$)
Nous allons voir maintenant comment on utilise les déclarations logiques et de typage dans les déductions qui nous intéressent : la formation de programmes et la preuve de formules.

3.3 Règles et axiomes purs

Nous présentons dans cette section les règles et axiomes les plus "purs" dans le sens où ils ne font pas intervenir d'opérateurs définis. Disposant de formules logiques et de types, nous aurons les règles logiques et les règles de typage, ainsi que des règles mixtes. Les axiomes serviront à donner les propriétés du sous-typage.

3.3.1 Règles logiques

Les règles logiques sont les règles usuelles du calcul des prédicats d'ordre supérieur :

$$\frac{WF(\Gamma) \qquad P \in \Gamma}{\Gamma \vdash P} Ax^o$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow^o Q} \Rightarrow^o_I \qquad \frac{\Gamma \vdash P \Rightarrow^o Q \qquad \Gamma \vdash P}{\Gamma \vdash Q} \Rightarrow^o_E$$

$$\frac{\Gamma, x : s \vdash Q}{\Gamma \vdash \forall^o_s x\, Q} \forall^o_I \qquad \frac{\Gamma \vdash \forall^o_s Q \qquad \Gamma \vdash t : s}{\Gamma \vdash Q(t)} \forall^o_E$$

3.3.2 Règles de typage

Les règles de typage sont analogues aux règles logiques, sauf qu'elles ont un "contenu algorithmique", c'est à dire qu'elles servent à bâtir des programmes. On se donne donc un "deuxième étage" de λ-termes, que l'on va noter comme des programmes dans le style ML :

$$\mathcal{P} = \quad \mathcal{V} \quad | \quad \texttt{fun}\ \mathcal{V}\ \texttt{->}\mathcal{P} \quad | \quad (\ \mathcal{P}\ \mathcal{P}\ )$$

Puis on se donne les règles suivantes :

$$\frac{WF(\Gamma) \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\, Ax^{\tau}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ \ x\ \ \texttt{->}\ \ t : A \Rightarrow^{\tau} B}\, \Rightarrow^{\tau}_{I} \qquad \frac{\Gamma \vdash u : A \Rightarrow^{\tau} B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\ u\ \ v\ ) : B}\, \Rightarrow^{\tau}_{E}$$

$$\frac{\Gamma, X : s \vdash t : A}{\Gamma \vdash t : \forall^{\tau}_{s} X.A}\, \forall^{s}_{I} \qquad \frac{\Gamma \vdash t : \forall^{\tau}_{s} A \qquad \Gamma \vdash v : s}{\Gamma \vdash t : A(v)}\, \forall^{s}_{E}$$

On remarque que ces règles sont les règles du système F, la quantification étant étendue à toutes les sortes.

3.3.3 β-règles

Il s'agit des règles de β-conversion :

$$\frac{\Gamma \vdash t : A \qquad A \succ_{\beta} A'}{\Gamma \vdash t : A'}\, \beta^{\tau}_{red} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A' : \tau \qquad A' \succ_{\beta} A}{\Gamma \vdash t : A'}\, \beta^{\tau}_{exp}$$

$$\frac{\Gamma \vdash P \qquad P \succ_{\beta} P'}{\Gamma \vdash P'}\, \beta^{o}_{red} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P' : o \qquad P' \succ_{\beta} P}{\Gamma \vdash P'}\, \beta^{o}_{exp}$$

La relation $\succ_{\beta}$ étant définie comme dans la définition 1 page 15.

3.3.4 Règles mixtes

Il s'agit des règles qui font intervenir à la fois des jugements de typage et des jugements (ou hypothèses) logiques.
La première des règles mixtes est la règle de sous-typage :

$$\frac{\Gamma \vdash A \subset B \qquad \Gamma \vdash t : A}{\Gamma \vdash t : B}\, \subset$$

L'introduction et l'élimination de la flèche spéciale sont les pendants des règles correspondantes pour les $\Rightarrow$:

$$\frac{\Gamma, P \vdash t : A}{\Gamma \vdash t : P \twoheadrightarrow A}\, \twoheadrightarrow_{I} \qquad \frac{\Gamma \vdash t : P \twoheadrightarrow A \qquad \Gamma \vdash P}{\Gamma \vdash t : A}\, \twoheadrightarrow_{E}$$

Il y a de plus deux règles structurelles :

– La règle de renforcement :

$$\frac{\Gamma, x : A \vdash A \subset B}{\Gamma \vdash A \subset B}\, Renf$$

– La règle de coupure, qui est une règle gauche :

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash A' \subset A}{\Gamma, x : A' \vdash t : B}\, Coup$$

3.3.5 Axiomes

Il s'agit d'un ensemble de formules $\mathcal{A}$ tel qu'on ait la règle suivante :

$$\frac{WF(\Gamma) \qquad P \in \mathcal{A}}{\Gamma \vdash P}\, Axiome$$

Ces axiomes servent à exprimer les propriétés du sous-typage. Ils peuvent être vus comme des propriétés d'une relation binaire qui est presque un "ordre", sans l'antisymétrie :

Axiome 1 (Les axiomes de base)

1. *Réflexivité :* $\forall A(A \subset A)$
2. *Transitivité :* $\forall A, B, C((A \subset B) \Rightarrow (B \subset C) \Rightarrow (A \subset C))$
3. *Borne inférieure (pour chaque sorte s) :*
   (a) *Minorant :* $\forall^o_{(s\to\tau)} B \forall^o_s x(\forall^\tau_s B \subset B(x))$
   (b) *Plus grand Minorant :* $\forall^o_\tau A \forall^o_{(s\to\tau)} B(\forall^o_s x(A \subset B(x)) \Rightarrow (A \subset \forall^\tau_s B))$
4. *Contra- et Co-variance de* $\Rightarrow^\tau$ *:*
   $\forall A, B, C, D((A \subset B) \Rightarrow (C \subset D) \Rightarrow ((B \Rightarrow^\tau C) \subset (A \Rightarrow^\tau D)))$
5. *Flèche spéciale :*
   (a) *Gauche :* $\forall P, \forall A(P \Rightarrow ((P \twoheadrightarrow A) \subset A))$
   (b) *Droit :* $\forall P, \forall A, B((P \Rightarrow (A \subset B)) \Rightarrow (A \subset (P \twoheadrightarrow B)))$
   (c) *Permutation :* $\forall P \forall A, B((P \twoheadrightarrow (A \Rightarrow B)) \subset (A \Rightarrow (P \twoheadrightarrow B)))$
6. *Axiome de Mitchell :* $\forall A, B(\forall x(A(x) \Rightarrow B(x)) \subset (\forall x A(x) \Rightarrow \forall x B(x)))$

Faisons ici quelques commentaires sur ces axiomes pour en donner le sens intuitif. Il est bon d'avoir en tête que les types "représentent" des ensembles de programmes, et que si A et B sont deux types, $A \Rightarrow^\tau B$ représente l'ensemble formé des programmes f tels que pour tout $a \in A$, $(f\ a) \in B$. L'inclusion $\subset$ représente simplement l'inclusion entre ensembles, et le $\forall^\tau_s$ l'intersection. La flèche spéciale $\twoheadrightarrow$ a le sens suivant : si A est un type et P une proposition :
– si P est vraie, $P \twoheadrightarrow A$ est A.
– si P est fausse, $P \twoheadrightarrow A$ est tout.
Muni de ce guide, il est plus aisé de comprendre les axiomes :
– Les axiomes 1 et 2 donnent les propriétés d'une relation d'ordre, sauf l'antisymétrie.
– Les axiomes 3(a) et (b) caractérisent l'intersection en tant que borne inférieure.
– L'axiome 4 est clair à comprendre avec le sens intuitif de $\Rightarrow^\tau$.
– Les axiomes 5(a) et (b) peuvent être compris comme caractérisant le comportement intuitif de $\twoheadrightarrow$. L'axiome 5(c) permet de permuter les deux types de flèches : munis du tiers exclu (fait 29.3, page 53), il est en fait équivalent à l'axiome suivant qui concerne uniquement $\Rightarrow^\tau$:

$$\top^\tau \subset (A \Rightarrow \top^\tau)^2$$

– L'axiome 6 se comprend, comme l'axiome 4, avec le sens intuitif de $\Rightarrow^\tau$.
Nous en avons terminé ici avec les règles et axiomes de base. Nous allons maintenant définir les opérateurs qui permettront d'une part de mieux apprécier l'expressivité du système, et d'autre part de donner d'autres axiomes qui constitueront avec les précédents le système complet.

3.4 Opérateurs

3.4.1 Opérateurs logiques

Les opérateurs logiques sont les codages usuels des connecteurs au second ordre (sauf pour l'absurde) :

Définition 25 (opérateurs logiques)
– *Absurde :* $\bot^o := \forall^o_\tau X, Y(X \subset Y)$.
– *Conjonction :* $P \wedge Q := \forall K((P \Rightarrow Q \Rightarrow K) \Rightarrow K)$.
– *Disjonction :* $P \vee Q := \forall K((P \Rightarrow K) \Rightarrow (Q \Rightarrow K) \Rightarrow K)$.
– *Existence :* $\exists^s x P(x) := \forall K(\forall^o_s x(P(x) \Rightarrow K) \Rightarrow K)$.
– *Négation :* $\neg P := P \Rightarrow \bot^o$.

Nous verrons à la section 3.7, fait 29, que l'absurde tel qu'il est défini entraîne l'absurde usuel[3] $(\forall^o_o X.X)$.

[2]Nous laissons au lecteur, à titre d'exercice, le choix d'une définition de $\top^\tau$ en tant que plus grand élément, et la vérification que les axiomes sont équivalents

[3]cela n'apparaissait pas dans les articles originaux, mais en est une conséquence relativement facile

3.4.2 Opérateurs de types

Les opérateurs de types servent tout d'abord à définir les opérations ensemblistes usuelles. Nous commençons par les plus simples :

Définition 26 (opérateurs de types (1))
- *Plus petit type :* $\bot^\tau := \forall^\tau X X$.
- *Restriction :* $A \upharpoonright P := \forall X((A \subset (P \twoheadrightarrow X)) \twoheadrightarrow X)$.
- *Réunion binaire :* $A \cup B := \forall X((A \subset X) \twoheadrightarrow (B \subset X) \twoheadrightarrow X)$.
- *Réunion :* $\cup^s x A(x) := \forall X(\forall^o_s x(A(x) \subset X) \twoheadrightarrow X)$.

Noter la similarité entre les opérateurs logiques Conjonction, Disjonction et Existence et leurs contrepartie respectives en types Restriction, Réunion binaire et Réunion.
Donnons ici quelques commentaires et conventions :
- Le plus petit type représente intuitivement l'ensemble vide.
- Si A est un type et P une proposition :
  - si P est vraie, $A \upharpoonright P$ est A.
  - si P est fausse, $A \upharpoonright P$ est vide.
  Des raisonnements élémentaires utilisant les éléments intuitifs fournis page 43 permettent d'établir cela.
- La réunion binaire est bien nommée.
- Pour la réunion $\cup^s$, tout comme pour l'existentiel $\exists^s$, nous omettrons la sorte lorsqu'il est inutile de la préciser.
- Pour faciliter la compréhension, il peut être utile au lecteur de visualiser la réunion de la façon suivante :

$$\cup^s x A(x) \text{ signifie } \bigcup_{x:s} A(x)$$

Nous donnons ensuite les opérateurs plus élaborés :

Définition 27 (opérateurs de types (2))
- *Intersection binaire :* $A \cap B := \cup^\tau X(X \upharpoonright ((X \subset A) \wedge (X \subset B)))$.
- *Complémentaire :* $A^c := \cup^\tau X(X \upharpoonright (X \cap A \subset \bot^\tau))$.
- *Application de types :* $A[B] := \forall X((A \subset (B \Rightarrow X)) \twoheadrightarrow X)$.

Les deux premiers opérateurs se passent de commentaires : leurs noms comme leurs notations suggèrent leurs dénotations intuitives.
Le dernier opérateur peut surprendre à première vue : sémantiquement, il représentera l'ensemble des termes qui "sont" des termes de type A appliqués à des termes de type B. Il revêtira une importance particulière dans le chapitre suivant, car il est la première brique qui permettra de représenter les programmes dans les types, ce qui est une des clés du fonctionnement de la preuve de réduction du sujet.

3.4.3 Prédicats de types

Il s'agit d'opérateurs décrivant des propriétés de types.

Définition 28 (prédicats de types) *Opérateurs ensemblistes :*
– *Égalité :* $A =_\subset B := (A \subset B \wedge B \subset A)$
– *Non vacuité :* $A \neq \emptyset := \neg(A =_\subset \bot_\tau)$
– *Singleton :* $S(A) := (A \neq \emptyset \wedge \forall^o_{\tau\to\tau} B((A \subset \cup x B(x)) \Rightarrow \exists x(A \subset B(x))))$

Tout comme dans les articles originaux, nous prenons pour l'égalité une version plus faible que l'égalité de Leibnitz ($A =_L B := \forall X(X(A) \Rightarrow X(B))$), l'égalité $=_\subset$ étant suffisante pour montrer tout ce dont nous aurons besoin jusqu'au chapitre où nous introduirons le schéma de compréhension (chapitre 6). Ceci justifie a posteriori que le troisième axiome caractérisant la relation d'ordre (l'antisymétrie, qui sera $\forall A, B(A =_\subset B \Rightarrow A =_L B)$) n'ait pas été donné.
La non vacuité signifie, avec la sémantique intuitive, le fait d'être distinct du vide.
Le dernier prédicat sera également éclairé par la sémantique : l'interprétation des types se fera en terme de parties saturées, être un singleton signifiant être une classe d'équivalence pour la relation choisie. Cela est à rapprocher de la notion d'élément premier vue dans le chapitre 2. L'importance des singletons apparaîtra notamment dans la preuve de préservation du type (chapitre 4), puis dans la construction du langage de programmation où l'on s'en servira pour dénoter des programmes, dans les types. C'est cette approche orientée vers les spécifications qui avait été soulignée par D. Aspinall dans son article [2] sur le sous-typage et les singletons. Une des différences dans le système ST réside dans le fait de pouvoir définir les singletons de manière *interne*.

3.5 Règles et Axiomes avec opérateurs

3.5.1 Règles

Il y a deux règles qui utilisent les opérateurs :
– La règle classique :

$$\frac{\Gamma \vdash t : (P \twoheadrightarrow \bot^\tau) \Rightarrow \bot^\tau}{\Gamma \vdash P}\, RC$$

– Règle gauche de la réunion :

$$\frac{\Gamma, y : s, x : F(y) \vdash t : A \qquad \Gamma \vdash A : \tau}{\Gamma, x : \cup^s y F(y) \vdash t : A}\, UG$$

La règle classique est un avatar "mixte" de la Loi de Pierce, dont le lecteur se convaincra de la validité à l'aide de la sémantique intuitive.

3.5.2 Axiomes

Axiome 2

1. *Le faux est vide :* $\forall A, B(A \subset (\bot^\tau \Rightarrow B))$
2. *Atomiticité :* $\forall A(A \subset \cup X(X \upharpoonright (S(X) \wedge X \subset A)))$
3. *Complément :* $\forall A, B(A \subset B \cup B^c)$
4. *Axiomes de l'application :*
   (a) *Continuité à droite :* $\forall A, B(A[\cup x B(x)] \subset \cup x(A[B(x)]))$
   (b) *Conservation des singletons :* $\forall A, B(S(A) \Rightarrow S(B) \Rightarrow S(A[B]))$
   (c) *Extensionnalité :* $\forall A, B(S(B) \Rightarrow \forall X(A[X] \subset B[X]) \Rightarrow (A \subset B))$

Donnons ici encore quelques commentaires sur ces axiomes :
- L'axiome 1, tout comme l'axiome $5(c)$ page 42, précise le comportement de $\Rightarrow^\tau$, cette fois par rapport au "faux" ou "vide" $\bot^\tau$.
- L'axiome 2 donne le sens d'inclusion qui manque pour pouvoir dire que tout type est égal à la réunion des singletons qui le minore. L'autre sens est dérivable.
- Les axiomes de l'application se comprennent à l'aide de la sémantique intuitive.

3.5.3 Choix des axiomes

Par rapport aux articles originaux [50] et [49], nous avons modifié la présentation des règles et axiomes :
- Nous avons changé l'ordre des règles, en regroupant les règles logiques, les règles de typage et les règles mixtes, et nommé les règles, ce qui est une pure modification de forme.
- Nous avons identifié ce qui pouvait se regrouper sous la règle *Axiome*, ce qui est encore une modification de forme (jouant sur le lien entre $\Rightarrow$, $\vdash$ et $\frac{\text{hypothèse}}{\text{conclusion}}$).
- Enfin, nous avons remplacé les axiomes *Axiome de la réunion* (*Union Axiom* dans [49]) et *Axiome de commutation des singletons* (*Commutation of singletons* de [49]) par respectivement les axiomes de *Continuité à droite* et de *Conservation des singletons* : nous discutons ici ce dernier point.

L'axiome de *Conservation des singletons* est équivalent à celui de *Commutation of singletons*, comme c'était remarqué dans [49]. Nous verrons dans la section 3.7 que *Axiome de la réunion* est entraîné par la *Continuité à droite*, et cette dernière apparaissait dans un fait (fait 19 de [49]). Nous avons vérifié que les deux énoncés étaient équivalents sous les mêmes axiomes[4]. Ils semblent donc être de "force" égale.
Ainsi, pour ces deux paires d'axiomes, le critère qui nous guidé notre choix était celui de la proximité avec les modèles, donc de la facilité avec laquelle on pouvait montrer la consistance par interprétation dans une sémantique : c'est le sujet que nous abordons maintenant.

[4] vérification faite dans PhoX

3.6 Sémantique

3.6.1 Définition de la sémantique

On se donne $(E, \bot, \top, \leq, @)$ un treillis applicatif λ-compact (définition 21 page 34).

Interprétations des sortes

On définit l'interprétation $\overline{s}$ d'une sorte par induction sur la structure de s :
- $\overline{\tau} = E$
- $\overline{o} = \{\bot; \top\}$
- $\overline{s_1 \rightarrow s_2} = s_2^{s_1}$, ensemble des fonctions de s_1 dans s_2.

Interprétation des termes et des programmes

Définition 29 (Valuation) *On appelle **valuation** toute fonction de domaine l'ensemble des variables de termes et de programmes et de codomaine la réunion des interprétations de sortes telle que l'interprétation de toute variable de programme soit un compact de E.*

Soit t un terme et $\mathcal{V}$ une valuation. On définit l'interprétation de t sous la valuation $\mathcal{V}$, si elle existe, notée $\mathcal{I}_\mathcal{V}(t)$ par :
- si t est une variable, $\mathcal{I}_\mathcal{V}(t) = \mathcal{V}(t)$
- si t est une constante :
  - $\mathcal{I}_\mathcal{V}(\Rightarrow^o) = p \mapsto q \mapsto$ si $p = \top$ et $q = \bot$ alors $\bot$, sinon $\top$
  - $\mathcal{I}_\mathcal{V}(\forall_s^o) = f \mapsto \wedge\{f(x); x \in \overline{s}\}$
  - $\mathcal{I}_\mathcal{V}(\Rightarrow^\tau) = a \mapsto b \mapsto (a \rightarrow b)$, (définition 19 page 30).
  - $\mathcal{I}_\mathcal{V}(\forall_s^\tau) = f \mapsto \wedge\{f(x); x \in \overline{s}\}$
  - $\mathcal{I}(\twoheadrightarrow) = p \mapsto a \mapsto$ si $p = \bot$ alors $\top$ sinon a.
  - $\mathcal{I}_\mathcal{V}(\subset) = a \mapsto b \mapsto$ si $a \leq b$ alors $\top$ sinon $\bot$
- si $t = u(v)$, $\mathcal{I}_\mathcal{V}(u) \in \overline{s_1 \rightarrow s_2}$ et $\mathcal{I}_\mathcal{V}(v) \in \overline{s_2}$, $\mathcal{I}_\mathcal{V}(t) = \mathcal{I}_\mathcal{V}(u)(\mathcal{I}_\mathcal{V}(v))$.
- si $t = \lambda x : s_1.u$, si pour tout $a \in \overline{s_1}$, $\mathcal{I}_{\mathcal{V}[x:=a]}(u) \in s_2$, alors $\mathcal{I}_\mathcal{V}(t) = a \mapsto \mathcal{I}_{\mathcal{V}[x:=a]}(u)$.

On remarque que cette définition est par induction sur la longueur de t, et implique notamment que l'interprétation est une fonction partielle : $\lambda x : s.x(x)$ n'a pas d'interprétation.

Soit p un programme et $\mathcal{V}$ une valuation. On définit l'interprétation de p sous la valuation $\mathcal{V}$, notée $\mathcal{I}_\mathcal{V}(p)$ comme au chapitre 2, définition 20 page 31 :
- Si p est une variable de programme : $\mathcal{I}_\mathcal{V}(p) = \mathcal{V}(p)$
- Si $p =$ `(u v)` : $\mathcal{I}_\mathcal{V}(p) = \mathcal{I}_\mathcal{V}(u)@\mathcal{I}_\mathcal{V}(v)$
- Si $p =$ `fun x -> u` : $\mathcal{I}_\mathcal{V}(p) = \Lambda(a \mapsto \mathcal{I}_{\mathcal{V}[x:=a]}(u))$

3.6.2 Lemme d'adéquation

Définition 30 (Satisfaction) *Soit $\mathcal{V}$ une valuation.*
Soit Γ un contexte bien formé.
*$\mathcal{V}$ **satisfait** Γ, noté $\mathcal{V} \vDash \Gamma$ est défini par :*
- $\mathcal{V} \vDash []$
- $\mathcal{V} \vDash \Gamma, x : s$ *si et seulement si* $\mathcal{V}(x) \in \overline{s}$.
- $\mathcal{V} \vDash \Gamma, P$ *si et seulement si* $\mathcal{V} \vDash \Gamma$ *et* $\mathcal{I}_{\mathcal{V}}(P) = \top$
- $\mathcal{V} \vDash \Gamma, x : A$ *si et seulement si* $\mathcal{V} \vDash \Gamma$ *et* $\mathcal{V}(x) \leq \mathcal{I}_{\mathcal{V}}(A)$.

Le lemme d'adéquation s'énonce de la façon suivante :

Lemme 22 *Pour tout contexte Γ bien formé, pour tout terme t, pour toute sorte s, pour tout termes A et P, pour toute valuation $\mathcal{V}$, si $\mathcal{V} \vDash \Gamma$:*
- *Si $\Gamma \vdash t : s$, alors $\mathcal{I}_{\mathcal{V}}(t) \in \overline{s}$.*
- *Si $\Gamma \vdash P$, alors $\mathcal{I}_{\mathcal{V}}(P) = 1$.*
- *Si $\Gamma \vdash p : A$, alors $\mathcal{I}_{\mathcal{V}}(p) \leq \mathcal{I}_{\mathcal{V}}(A)$, et $\mathcal{I}_{\mathcal{V}}(p)$ est compacte.*

preuve: On prouve par induction sur la longueur de la dérivation la totalité de l'énoncé (on notera H.I. pour hypothèse d'induction).
- Formation des termes : trivial par la sémantique.
- Règles et axiomes purs
  - Règles logiques
    - Ax^o : trivial par définition de $\mathcal{V} \vDash \Gamma$
    - $\Rightarrow_I^o$: Si $\mathcal{I}_{\mathcal{V}}(P) = \top$, alors $\mathcal{V} \vDash \Gamma, P$ et donc par H.I. $\mathcal{I}_{\mathcal{V}}(Q) = \top$,et donc $\mathcal{I}_{\mathcal{V}}(P \Rightarrow Q) = \top$. Sinon, $\mathcal{I}_{\mathcal{V}}(P) = \bot$, et donc $\mathcal{I}_{\mathcal{V}}(P \Rightarrow Q) = \top$.
    - $\Rightarrow_E^o$: par H.I., $\mathcal{I}_{\mathcal{V}}(P \Rightarrow Q) = \mathcal{I}_{\mathcal{V}}(P) = \top$, et donc $\mathcal{I}_{\mathcal{V}}(Q) = \top$ par définition de la sémantique.
    - $\forall_I^o$: par H.I., $\mathcal{I}_{\mathcal{V}[x:=a]}(Q) = \top$ pour tout $a \in s$, et donc $\mathcal{I}_{\mathcal{V}}(\lambda x : s.Q) = a \mapsto \top$. Il s'ensuit que $\wedge\{\mathcal{I}_{\mathcal{V}}(\lambda x : s.Q)(a); a \in \overline{s}\} = \top$.
    - $\forall_E^o$: par H.I. $\mathcal{I}_{\mathcal{V}}(Q) = a \mapsto \top$, et donc $\mathcal{I}_{\mathcal{V}}(Q(t)) = \top$.
  - Règles de typage
    - La compacité de $\mathcal{I}_{\mathcal{V}}(p)$ est immédiate par définition d'un treillis applicatif λ-compact.
    - Ax^τ : trivial par définition de $\mathcal{V} \vDash \Gamma$.
    - $\Rightarrow_I^\tau$: On a par définition :

$$\begin{aligned} \mathcal{I}_{\mathcal{V}}(\texttt{fun } x \texttt{ -> t}) &= \wedge\{a \to \mathcal{I}_{\mathcal{V}[x:=a]}(t); a \in E\} \\ &\leq \wedge\{a \to \mathcal{I}_{\mathcal{V}[x:=a]}(t); a \text{ compact}, a \leq \mathcal{I}_{\mathcal{V}}(A)\} \end{aligned}$$

Par H.I, pour tout $a \leq \mathcal{I}_{\mathcal{V}}(A)$ compact, $\mathcal{I}_{\mathcal{V}[x:=a]}(t) \leq \mathcal{I}_{\mathcal{V}}(B)$, et donc :

$$\mathcal{I}_{\mathcal{V}}(\texttt{fun } x \texttt{ -> t}) \leq \wedge\{a \to \mathcal{I}_{\mathcal{V}}(B); a \text{ compact}, a \leq \mathcal{I}_{\mathcal{V}}(A)\}$$

Or, $\mathcal{I}_\mathcal{V}(A) = \vee\{a; a \text{ compact}, a \leq \mathcal{I}_\mathcal{V}(A)\}$, et donc (pour une meilleure lisibilité, on note A (resp. B) pour $\mathcal{I}_\mathcal{V}(A)$ (resp. $\mathcal{I}_\mathcal{V}(B)$)) :

$$\begin{array}{rcl}\mathcal{I}_\mathcal{V}(\texttt{fun } x \texttt{ -> t})@A & \leq & \wedge\{a \to B; a \text{ compact}, a \leq A\}@ \vee \{a; a \text{ compact}, a \leq A\} \\ & \leq & \vee\{\wedge\{a \to B; a \text{ compact}, a \leq A\}@a; a \text{ compact}, a \leq A\} \\ & \leq & \vee\{(a \to B)@a; a \text{ compact}, a \leq A\} \\ & \leq & \vee\{B; a \text{ compact}, a \leq A\} \\ & \leq & B\end{array}$$

On en déduit par le lemme 18 que $\mathcal{I}_\mathcal{V}(\texttt{fun } x \texttt{ -> t}) \leq A \to B$.

- $\Rightarrow_E^\tau$: Par H.I. $\mathcal{I}_\mathcal{V}(u) \leq \mathcal{I}_\mathcal{V}(A) \to \mathcal{I}_\mathcal{V}(B)$ et $\mathcal{I}_\mathcal{V}(u) \leq \mathcal{I}_\mathcal{V}A$. Donc par le lemme 18 :

$$\begin{array}{rcl}\mathcal{I}_\mathcal{V}((u\ v)) & = & \mathcal{I}_\mathcal{V}(u)@\mathcal{I}_\mathcal{V}(v) \\ & \leq & (\mathcal{I}_\mathcal{V}(A) \to \mathcal{I}_\mathcal{V}(B))@\mathcal{I}_\mathcal{V}(A) \\ & \leq & \mathcal{I}_\mathcal{V}(B)\end{array}$$

- $\forall_I^\tau$: par H.I., pour tout $a \in s$, $\mathcal{I}_\mathcal{V}(t) \leq \mathcal{I}_{\mathcal{V}[X:=a]}(A)$, et donc

$$\begin{array}{rcl}\mathcal{I}_\mathcal{V}(t) & \leq & \wedge\{\mathcal{I}_{\mathcal{V}[X:=a]}(A); a \in s\} \\ & = & \mathcal{I}_\mathcal{V}(\forall_s^\tau(\lambda X : s.A))\end{array}$$

- $\forall_E^\tau$: Par H.I. $\mathcal{I}_\mathcal{V}(t) \leq \wedge\{\mathcal{I}_\mathcal{V}(A)(a); a \in s\}$ et $\mathcal{I}_\mathcal{V}(v) \in s$, donc $\mathcal{I}_\mathcal{V}(t) \leq \mathcal{I}_\mathcal{V}(A(v))$.

- Règles mixtes :
  - $\subset$: Immédiat par H.I.
  - $\twoheadrightarrow_I$: Immédiat en distinguant les cas $\mathcal{I}_\mathcal{V}(P) = \top$ ou $\bot$, comme pour la règle logique correspondante.
  - $\twoheadrightarrow_E$: Immédiat par H.I. et sémantique.
  - *Renf* : On distingue deux cas :
    - Si $\mathcal{I}_\mathcal{V}(A) = \bot$, alors c'est bon.
    - Sinon, soit $a \leq A$ compact. $\mathcal{I}_{\mathcal{V}[x:=a]} \vDash \Gamma, x : A$ et donc par H.I. $\mathcal{I}_\mathcal{V}(A \subset B) = \mathcal{I}_{\mathcal{V}[x:=a]}(A \subset B) = \top$.
  - *Coup* : Si $\mathcal{V} \vDash \Gamma, x : A'$, alors par H.I., comme $\mathcal{V} \vDash \Gamma$, $\mathcal{I}_\mathcal{V}(A' \subset A) = \top$ et donc $\mathcal{V} \vDash \Gamma, x : A$, donc par H.I. $\mathcal{I}_\mathcal{V}(t) \leq \mathcal{I}_\mathcal{V}(B)$.
- Axiomes : ils se vérifient tous immédiatement, sauf celui de Mitchell. Soit $a \in s$, on a, en utilisant la sémantique et le lemme 18 page 31 :

$$\begin{array}{rc}& \mathcal{I}_\mathcal{V}(\forall_s^\tau x(A(x) \Rightarrow B(x)))@\mathcal{I}_\mathcal{V}(\forall_s^\tau xA(x)) \\ = & \wedge\{\mathcal{I}_\mathcal{V}(A)(y) \to \mathcal{I}_\mathcal{V}(B)(y); y \in s\}@ \wedge \{\mathcal{I}_\mathcal{V}(A)(y); y \in s\} \\ \leq & (\mathcal{I}_\mathcal{V}(A)(a) \to \mathcal{I}_\mathcal{V}(B)(a))@\mathcal{I}_\mathcal{V}(A)(a) \\ \leq & \mathcal{I}_\mathcal{V}(B)(a)\end{array}$$

On a ceci pour tout $a \in s$, et donc :

$$\begin{array}{rcl}\mathcal{I}(\forall_s^\tau x(A(x) \Rightarrow B(x)))@\mathcal{I}(\forall_s^\tau xA(x)) & \leq & \wedge\{\mathcal{I}(B)(y); y \in s\} \\ & = & \mathcal{I}(\forall_x^\tau B(x))\end{array}$$

Et enfin par le lemme 18, on en déduit que

$$\begin{aligned}\mathcal{I}(\forall^\tau_s x(A(x) \Rightarrow B(x))) &\leq \mathcal{I}(\forall^\tau_s x A(x)) \to \mathcal{I}(\forall^\tau_x B(x)) \\ &= \mathcal{I}(\forall^\tau_s x A(x) \Rightarrow \forall^\tau_x B(x))\end{aligned}$$

- Règles et axiomes avec opérateurs On a besoin ici du lemme 26 qui sera prouvé dans la section suivante.
  - Règles
    - *RC* : par H.I., $\mathcal{I}_\mathcal{V}((P \twoheadrightarrow \bot^\tau) \Rightarrow \bot^\tau) \neq \bot$ car $\mathcal{I}_\mathcal{V}(t)$ est compact (donc non vide). Il s'ensuit que $\mathcal{I}(P) \neq \bot$), donc $\mathcal{I}(P) = \top$.
    - *UG* : si $\mathcal{V} \vDash \Gamma, x : \cup y F(y)$, comme $\mathcal{V}(x)$ est compact, il s'ensuit qu'il existe $a_0 \in s$ tel que $\mathcal{V}(x) \subset F(a_0)$. Il est donc clair que $\mathcal{V}[y := a_0] \vDash \Gamma, y : s, x : F(y)$, et donc, avec la prémisse $\Gamma \vdash A : \tau$ et l'hypothèse d'induction :

$$\begin{aligned}\mathcal{I}_\mathcal{V}(t) &= \mathcal{I}_{\mathcal{V}[y:=a_0]}(t) \\ &\leq \mathcal{I}_{\mathcal{V}[y:=a_0]}(A) \\ &= \mathcal{I}_\mathcal{V}(A)\end{aligned}$$

  - Axiomes : Il se vérifient tous immédiatement par la sémantique, excepté l'extensionnalité. On suppose que B est un singleton et que $\forall X(A[X] \subset B[X])$. Il s'ensuit que $A \subset \forall X(X \Rightarrow B[X])$. Or $\mathcal{I}(\forall X(X \Rightarrow B[X])) = \mathcal{I}_{\mathcal{V}[b:=B]}(\texttt{fun } x \texttt{ -> } (b\ x))$ et, b étant compact, $\mathcal{I}_{\mathcal{V}[b:=B]}(\texttt{fun } x \texttt{ -> } (b\ x))$ est compact aussi et supérieur à b (voir par exemple la preuve de la proposition 17, page 33) , donc égal à b.

□

3.6.3 Lemmes techniques

Interprétations

Lemme 23 *Pour toute valuation $\mathcal{V}$, pour tout termes Q et t et toute variable x :*

$$\mathcal{I}_{\mathcal{V}[x:=\mathcal{I}_\mathcal{V}(t)]}(Q) = \mathcal{I}_\mathcal{V}(Q[x := t])$$

preuve: Par induction sur Q. □

Lemme 24 *Pour toute valuation $\mathcal{V}$, tout contexte Γ et tout terme T : Si $\Gamma \vdash T : s$, et si $T \succ_\beta T'$, alors $\Gamma \vdash T' : s$ et $\mathcal{I}_\mathcal{V}(T) = \mathcal{I}_\mathcal{V}(T')$*

preuve: La préservation du type par réduction découle du résultat pour le λ-calcul simplement typé. La préservation de l'interprétation est immédiate par induction. □

Lemme 25
- *Si $\Gamma \vdash P$, alors $\Gamma \vdash P : o$*
- *Si $\Gamma \vdash t : A$, alors $\Gamma \vdash A : \tau$.*

preuve: Preuve immédiate par induction mutuelle, en utilisant le lemme précédent pour les β-règles. □

Opérateurs et Prédicats

Lemme 26 (Interprétation des opérateurs de types) *Soit $\mathcal{V}$ une valuation, et $a, b \in E$, $p = \bot$ ou $\top$, s une sorte et $f : \overline{s} \to E$. L'interprétation des opérateurs est la suivante :*

1. $\mathcal{I}(\bot^\tau) = \bot$
2. $\mathcal{I}_{\mathcal{V}[A:=a,P:=p]}(A \restriction P) =$ *si* $p = \top$, *alors* a, *sinon* $\bot$
3. $\mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A \cup B) = \vee\{a; b\}$
4. $\mathcal{I}_{\mathcal{V}[A:=f]}(\cup_s x A(x)) = \vee\{f(y); y \in \overline{s}\}$
5. $\mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A \cap B) = \wedge\{a; b\}$
6. $\mathcal{I}_{\mathcal{V}[A:=a]}(A^c) = \vee\{k; k$ *compact et* $k \not\leq a\}$
7. $\mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A[B]) = a@b$

Remarque: On a $\mathcal{I}_{\mathcal{V}[P:=p]}(\forall X(P(X) \twoheadrightarrow X) = \wedge\{x; p(x) = \top\}$.
preuve:

1. Trivial.
2. – Si $p = \top$, alors $\mathcal{I}_{\mathcal{V}[X:=x]}(P \twoheadrightarrow X) = x$ pour tout $x \in \tau$, et donc $\mathcal{I}(A \restriction P) = a$ par la remarque.
   – Si $p = \top$, alors $\mathcal{I}_{\mathcal{V}[X:=x]}(P \twoheadrightarrow X) = \top$ et donc $\mathcal{I}(A \restriction P) = \bot$
3. $\mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A \cup B) = \wedge\{x; a \leq x$ et $b \leq x\} = \vee\{a; b\}$.
4. $\mathcal{I}_{\mathcal{V}[A:=f]}(\cup x A(x)) = \wedge\{x;$ Pour tout $y \in \overline{s}, f(y) \leq x\} = \vee\{f(y); y \in \overline{s}\}$
5. Par 1. et 4. :$\mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A \cap B) = \vee\{x; x \leq a$ et $x \leq b\} = \wedge\{a; b\}$
6. Par 1; et 4. : $\mathcal{I}_{\mathcal{V}[A:=a]}(A^c) = \vee\{x; \wedge\{x; a\} \leq \bot\}$. Soit k compact $k \leq \vee\{x; \wedge\{x; a\} \leq \bot\} = a^c$. Comme k est premier, il existe x_0 tel que $\wedge\{x_0; a\} \leq \bot$ et $k \leq x_0$; comme $k \neq \bot$, on a donc nécessairement $k \not\leq a$. Soit k compact, $k \not\leq a$. On immédiatement $\wedge\{k; a\} \leq \bot$ et donc $k \leq \vee\{x; \wedge\{x; a\} \leq \bot\}$. Comme le treillis est compact, $a^c = \vee\{k; k$ compact et $k \leq a^c\}$.
7. Notons $a[b] = \mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A[B])$. On montre la double inclusion :
   – $a[b] \leq a@b$ est immédiat car $a \leq (b \to a@b)$.
   – $a@b \leq a[b]$: si $a \leq b \to x$, alors $a@b \leq x$.

□

Lemme 27 (interprétation des prédicats de types)

1. $\mathcal{I}_{\mathcal{V}[A:=a,B:=b]}(A =_\subset B) = \top$ *si et seulement si* $a = b$.
2. $\mathcal{I}_{\mathcal{V}[A:=a]}(A \neq \emptyset) = \top$ *si et seulement si* $a \neq \bot$
3. $\mathcal{I}_{\mathcal{V}[A:=a]}(S(A)) = \top$ *si et seulement si* a *est compact.*

preuve:
– 1. et 2. sont immédiats.
– La traduction de la seconde partie de la conjonction est la définition d'élément premier.
□

3.7 Propriétés des opérateurs

Nous regroupons ici les faits qui seront utiles dans la suite et qui concernent les propriétés des opérateurs définis précédemment, ainsi que de quelques autres.

3.7.1 Opérateurs déjà définis

Application de types

Ce fait illustre les propriétés intuitives de l'application de types :

Fait 28 (Propriétés de l'application) *On a :*

1. *Équivalence de la flèche et de l'application :*
$$\vdash \forall A, B, C(A[B] \subset C \quad \Longleftrightarrow \quad A \subset (B \Rightarrow C))$$
est dérivable.
2. *Croissance de l'application :*
$$\vdash \forall A, A', B, B'(A \subset A' \quad \Rightarrow \quad B \subset B' \quad \Rightarrow \quad A[B] \subset A'[B'])$$
est dérivable.
3. *Typabilité de toute application : la règle suivante est dérivable :*
$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (uv) : A[B]}$$

preuve:

1. On prouve les deux sens :
   - $\Rightarrow$: On montre d'abord $A \subset B \Rightarrow A[B]$. On a effectivement $A \subset \forall X((A \subset (B \Rightarrow X)) \twoheadrightarrow (B \Rightarrow X))$ et en utilisant les axiomes de Permutation et de Mitchell, on en déduit le résultat. Il suffit donc de montrer que si $A[B] \subset C$, alors $B \Rightarrow A[B] \subset B \Rightarrow C$, ce qui est immédiat.
   - $\Leftarrow$: provient simplement de $A[B] \subset ((A \subset (B \Rightarrow C)) \twoheadrightarrow C$ et $((A \subset (B \Rightarrow C)) \twoheadrightarrow C \subset C$ par hypothèse.
2. Immédiat par les règles de sous-typage de $\Rightarrow$.
3. La dérivation est la suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, X : \tau, A \subset B \Rightarrow X \vdash u : A} \quad \dfrac{}{\Gamma, X : \tau, A \subset B \Rightarrow X \vdash A \subset B \Rightarrow X}\, Ax^o}{\Gamma, X : \tau, A \subset B \Rightarrow X \vdash u : B \Rightarrow X}\, \subset \quad \dfrac{\vdots}{\Gamma, X : \tau, A \subset B \Rightarrow X \vdash v : B}}{\Gamma, X : \tau, A \subset B \Rightarrow X \vdash (uv) : X}\, \Rightarrow^{\tau}_{E}}{\Gamma, X : \tau \vdash (uv) : (A \subset B \Rightarrow X) \twoheadrightarrow X}\, \twoheadrightarrow_I}{\Gamma \vdash (uv) : A[B]}\, \forall^{\tau}_{I}$$

□

Remarque: Le 3 de ce fait illustre la grande permissivité du système, qui autorise à appliquer n'importe quel programme à un autre, même si le programme qui joue le rôle de "fonction" n'a pas reçu un type fonctionnel (de la forme $A \Rightarrow^\tau B$).

Autour du faux

Nous regroupons dans le fait suivant des propriétés utiles liées à la règle RC.

Fait 29 *Les règles suivantes sont dérivables :*

1. $\dfrac{}{\Gamma \vdash \forall P((P \Rightarrow \bot^o) \Rightarrow \bot^o) \Rightarrow P)}$
2. $\dfrac{}{\Gamma \vdash \bot^o \Rightarrow \forall X X}$
3. $\dfrac{\Gamma, P \vdash Q \qquad \Gamma, P \Rightarrow \bot^o \vdash Q}{\Gamma \vdash Q}$
4. $\dfrac{\Gamma, x : A \vdash Q \qquad \Gamma, A \subset \bot^\tau \vdash Q}{\Gamma \vdash Q}$
5. $\dfrac{\Gamma \vdash t : \bot^\tau}{\Gamma \vdash \bot^o}$
6. $\dfrac{\Gamma \vdash t : A}{\Gamma \vdash A \neq \emptyset}$
7. $\dfrac{\Gamma, x : A \vdash B \subset B'}{\Gamma \vdash A \Rightarrow B \subset A \Rightarrow B'}$

preuve:

1. La dérivation est la suivante (on note $\neg P = P \Rightarrow \bot^o$) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \neg\neg P, x : P \twoheadrightarrow \bot^\tau, P \vdash x : \bot^\tau}\, Ax, \twoheadrightarrow_E}{\Gamma, \neg\neg P, x : P \twoheadrightarrow \bot^\tau, P \vdash \bot^o}\, \subset, RC}{\Gamma, \neg\neg P, x : P \twoheadrightarrow \bot^\tau \vdash P \Rightarrow \bot^o}\, \Rightarrow^o_I}{\Gamma, \neg\neg P, x : P \twoheadrightarrow \bot^\tau \vdash \bot^o}\, Ax, \Rightarrow^o_E}{\Gamma, \neg\neg P, x : P \twoheadrightarrow \bot^\tau \vdash x : \bot^\tau}\, \subset}{\Gamma, \neg\neg P \vdash \texttt{fun } x \texttt{ -> } x : (P \twoheadrightarrow \bot^\tau) \Rightarrow \bot^\tau}\, RC}{\Gamma, \neg\neg P \vdash P}$$

2. Immédiat par 1.
3. Immédiat par 1.
4. En utilisant 1, il suffit de montrer $\Gamma \vdash \neg\neg Q$. La dérivation est la suivante (il est facile de conclure) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \neg Q, x : A \vdash Q}}{\Gamma, \neg Q, x : A \vdash A \subset \bot^\tau}\Rightarrow_E^o, \forall_E^o}{\Gamma, \neg Q \vdash A \subset \bot^\tau}Renf \qquad \dfrac{\dfrac{\vdots}{\Gamma, \neg Q, A \subset \bot^\tau \vdash Q}}{\Gamma, \neg Q \vdash A \subset \bot^\tau \Rightarrow Q}\Rightarrow_I^o}{\Gamma, \neg Q \vdash Q}\Rightarrow_E^o$$

5. Immédiat par substitution par $(\bot^o \twoheadrightarrow \bot^\tau) \Rightarrow \bot^\tau$.
6. Il suffi de montrer $\Gamma, A \subset \bot^\tau \vdash \bot^o$, et donc par 5., $\Gamma, A \subset \bot^\tau \vdash t : \bot^\tau$, ce qui est immédiat.
7. Par 4., il suffit de montrer $\Gamma, A \subset \bot^\tau \vdash A \Rightarrow B \subset A \Rightarrow B'$, ce qui est immédiat par l'axiome Le faux est vide.

□

Remarque: Le 7. du fait précédent correspond à la propriété appelée "modified contraposition" dans le papier [50].

Opérateur restriction

Le fait suivant illustre le comportement de $A \upharpoonright P$: c'est A si P est vrai, et "vide" sinon.

Fait 30 *Les règles suivantes sont dérivables :*

1. $\vdash \forall A, B \forall P((P \Rightarrow A \subset B) \Rightarrow (A \upharpoonright P) \subset B)$
2. $\vdash \forall A \forall P((A \upharpoonright P) \subset A)$
3. $\vdash \forall A \forall P(P \quad \Rightarrow \quad A \subset (A \upharpoonright P))$
4. $\dfrac{\Gamma \vdash t : A \upharpoonright P}{\Gamma \vdash P}$
5. $\dfrac{\Gamma \vdash t : A \upharpoonright P}{\Gamma \vdash t : A}$
6. $\vdash \forall A, P((A \upharpoonright P) \neq \emptyset \quad \Rightarrow \quad P)$

preuve:

1. Immédiat par l'axiome droit de la flèche spéciale et l'axiome de la borne inférieure (a).
2. Immédiat par 1. et l'axiome de réflexivité.
3. Immédiat par l'axiome gauche de la flèche spéciale et l'axiome de la borne inférieure (b).
4. Immédiat en utilisant la règle RC (avec t) et l'axiome Le faux est vide.
5. Immédiat avec 2.
6. Avec le fait 29, il suffit de montrer $\neg\neg P$ sous l'hypothèse $(A \upharpoonright P) \neq \emptyset$, ce qui est immédiat en utilisant 1..

□

Réunion

Le fait suivant avait dans les versions originales le statut d'axiome, il remplaçait l'axiome de continuité de l'application.

Fait 31 *On montre la propriété suivante de la réunion :*

$$\vdash \forall F \forall A(\forall x(F(x) \Rightarrow A) \subset (\cup x F(x) \Rightarrow A))$$

preuve: Par le fait 28.1, il suffit de montrer que $\forall x(F(x) \Rightarrow A)[\cup x F(x)] \subset A$. Par l'axiome de continuité à droite de l'application, il suffit de montrer que $\cup y \forall x(F(x) \Rightarrow A)[F(y)] \subset A$. Cela découle immédiatement de $\forall y(\forall x(F(x) \Rightarrow A)[F(y)] \subset A)$ par le fait 28.1 encore. □

Singletons

On regroupe les propriétés qui utilisent la propriété d'être un singleton.

Fait 32 *1. Pour montrer une inclusion, il suffit de la montrer pour tous les singletons qui minorent le membre de gauche :*

$$\vdash \forall A, B(A \subset B \Longleftrightarrow \forall x(\mathcal{S}(x) \Rightarrow x \subset A \Rightarrow x \subset B))$$

2. Tout ensemble est égal à la réunion des singletons qu'il majore :

$$\vdash \forall A(A =_{\subset} \cup x(x \restriction (\mathcal{S}(x) \wedge x \subset A)))$$

3. Un type flèche peut être décrit par les types flèches portant sur les singletons qui le minorent :

$$\vdash \forall A, B(\forall X(\mathcal{S}(X) \twoheadrightarrow ((X \subset A) \twoheadrightarrow (X \Rightarrow B))) \quad \subset \quad A \Rightarrow B)$$

preuve:

1. On montre les deux sens de l'implication :
   - Le sens gauche-droite est trivial par transitivité de l'inclusion.
   - Le sens droite-gauche utilise l'axiome d'atomiticité : il suffit de montrer sous l'hypothèse $\forall x(\mathcal{S}(x) \Rightarrow x \subset A \Rightarrow x \subset B)$ que $\cup x(x \restriction (\mathcal{S}(x) \wedge x \subset A)) \subset B$. Il suffit de montrer $\forall x(x \restriction (\mathcal{S}(x) \wedge x \subset A)) \subset B)$, qui est évident par les fait 30 et 29.
2. On montre la double inclusion :
   - L'inclusion gauche-droite est l'axiome d'atomiticité.
   - L'inclusion droite-gauche est immédiate en montrant $\forall x(x \restriction (\mathcal{S}(x) \wedge x \subset A)) \subset A)$, comme au (1).
3. On montre d'abord que $\forall X(\mathcal{S}(X) \twoheadrightarrow (X \subset A) \twoheadrightarrow X \Rightarrow B) \subset \forall X(X \restriction (\mathcal{S}(X) \wedge (X \subset A)) \Rightarrow B)$. Plus généralement, cela vient de : $\vdash \forall A, P, B((P \twoheadrightarrow A \Rightarrow B) \subset (A \restriction P \Rightarrow B))$. En utilisant l'axiome de permutation, on veut montrer : $\vdash \forall A, P, B((A \Rightarrow P \twoheadrightarrow B) \subset (A \restriction P \Rightarrow B))$ Cela se montre en utilisant le fait 29 2. :

– Sous l'hypothèse P, c'est immédiat avec le fait 30.
– Sous l'hypothèse $\neg P$, on a par le fait 30, $(A \restriction P) \subset \bot^\tau$, et donc $\bot^\tau \Rightarrow B \subset (A \restriction P) \Rightarrow B$, et on conclut avec l'axiome "le faux est vide".

Corollaire immédiate du fait 31, en utilisant l'axiome d'atomiticité et la contravariance de $\Rightarrow$.

□

On donne la définition de singleton pour une sorte s quelconque :

Définition 31 *Le prédicat "être un **singleton** pour la sorte s" est défini par :*

$$S^s(A) = (A \neq \emptyset \wedge \forall^o_{s\to\tau} B((A \subset \cup x B(x)) \Rightarrow \exists x(A \subset B(x))))$$

Le fait suivant exprime que le prédicat être un singleton entraîne être un singleton pour une sorte s quelconque :

Fait 33 *On prouve, pour toute sorte s :*

$$\vdash \forall A(\mathcal{S}(A) \Rightarrow \mathcal{S}^s(A))$$

preuve: On montre d'abord que $\vdash \forall^o_{s\to\tau} \cup^s xB(x) \subset \cup^\tau Y(Y \restriction (\exists^s x(Y \subset B(x))))$: on met en hypothèses $\forall Y(Y \restriction (\exists^s x(Y \subset B(x)))) \subset X$ et $x : s$, et il faut en déduire $B(x) \subset X$, ce qui est immédiat en prenant $Y = B(x)$. On suppose $\mathcal{S}(A)$, $A \subset \cup x B(x)$, et il faut en déduire $\exists x(A \subset B(x))$. D'après ce qu'on vient de voir, on peut supposer que $A \subset \cup^\tau Y(Y \restriction (\exists^s x(Y \subset B(x))))$; par $\mathcal{S}(A)$, on a $\exists Y A \subset (Y \restriction (\exists^s x(Y \subset B(x))))$, et par le fait 30, et la propriété $A \neq \emptyset$, on a, en supposant $A \subset (Y \restriction (\exists^s x(Y \subset B(x))))$, que $A \subset Y$ et que $\exists^s x(Y \subset B(x))$, $\exists^s x(A \subset B(x))$. □

Intersection

Fait 34 *L'intersection se distribue sur la réunion de la façon suivante :*

$$\forall A \forall B(A \cap \cup x B(x) =_\subset \cup x(A \cap B(x)))$$

preuve:
– On montre les deux sens :
– $(A \cap \cup x B(x) \subset \cup x(A \cap B(x)))$: Posons $U = \cup x(A \cap B(x))$ et $V = \cup x B(x)$. En utilisant la définition de $\cap$, il suffit de montrer que $\forall y(y \restriction (y \subset A \wedge y \subset V)) \subset U$. Sous l'hypothèse $y \subset A \wedge y \subset V$ (a), et en appliquant le fait 32 (1) à y, il suffit de montrer sous les hypothèses supplémentaires $\mathcal{S}(z)$ (b) et $z \subset y$ (c) que $z \subset U$. Or par les hypothèses, il est clair que $\exists t(z \subset B(t))$, qui entraîne la conclusion.
– $\cup_x(A \cap B(x))) \subset (A \cap \cup_x B(x)$: il suffit de montrer $\forall x(A \cap B(x) \subset A \cap (\cup x B(x)))$, ce qui est immédiat par croissance de $\cap$ (exercice).

□

Complémentaire

Fait 35 *Un type inclus dans son complémentaire est vide :*

$$\vdash \forall A((A \subset A^c) \Rightarrow (A =_\subset \emptyset))$$

preuve: Il suffit de montrer sous l'hypothèse $A \subset A^c$ que $A \subset \perp^\tau$. On a donc $A \subset A \cap A^c$, et par le fait 34 et la définition du complémentaire, on a $A \subset \cup_X(A \cap (X \restriction (X \cap A \subset \emptyset)))$, et donc il suffit de montrer que $\forall X((A \cap (X \restriction (X \cap A \subset \perp^\tau))) \subset \perp^\tau)$. Le tiers exclu sur $(X \cap A \subset \perp^\tau)$, et les propriétés de l'opérateur restriction (fait 30) nous permettent de conclure aisément. □

3.7.2 Nouveaux opérateurs

Unions binaires

Définition 32 *On définit une autre union binaire par :*

$$A\widehat{\cup}B = \cup X(X \restriction (X =_\subset A \vee X =_\subset B))$$

Fait 36

1. *Les deux réunions binaires sont équivalentes :*

$$\vdash \forall A, B(A \cup B =_\subset A\widehat{\cup}B)$$

2. *L'intersection se distribue sur l'union binaire :*

$$\vdash \forall A, B, C((B \cup C) \cap A =_\subset (B \cap A) \cup (C \cap A))$$

preuve:

1. On montre le double sens de l'inclusion :
   - $A \cup B \subset A\widehat{\cup}B$: on met en hypothèse $\forall X((X \restriction (X =_\subset A \vee X =_\subset B)) \subset K$ et il faut en déduire $A \cup B \subset K$. Par le fait 30, on a immédiatement $X \subset X \restriction (X =_\subset A \vee X =_\subset B)$ pour $X =_\subset A$ ou B. On en déduit $A \subset K$ et $B \subset K$, d'où le résultat.
   - $A\widehat{\cup}B \subset A \cup B$: on met en hypothèses $A \subset K$ et $B \subset K$, et il faut en déduire $A\widehat{\cup}B \subset K$. Il suffit de montrer que $\forall X(X \restriction (X =_\subset A \vee X =_\subset B) \subset K)$. Par le fait 30, on met en hypothèse $(X =_\subset A \vee X =_\subset B)$ et il faut en déduire $X =_\subset K$, ce qui est immédiat.
2. On montre le double sens de l'inclusion :
   - $(B \cup C) \cap A \subset (B \cap A) \cup (C \cap A)$: il suffit de montrer d'après 1. que $(B\widehat{\cup}C) \cap A \subset (B \cap A) \cup (C \cap A)$. Or, par le fait 34 on a $(B\widehat{\cup}C) \cap A \subset \cup X((X \restriction (X =_\subset B \vee X =_\subset C)) \cap A)$, et il suffit donc de montrer que $\forall X(X \restriction (X =_\subset B \vee X =_\subset C)) \cap A) \subset (B \cap A) \cup (C \cap A)$. Par un tiers exclus (fait 29) sur $X =_\subset B \vee X =_\subset C$, ceci qui donne facilement le résultat.

– $(B \cap A) \cup (C \cap A) \subset (B \cup C) \cap A$: immédiat en prouvant d'abord que $X \subset (B \cup C) \wedge X \subset A$ pour $X = B \cap A$ et $X = C \cap A$.

□

Décideur

Définition 33 *On appelle* ***décideur*** *de P et Q :*

$$P \| Q = \forall X((P \twoheadrightarrow X) \Rightarrow (Q \twoheadrightarrow X) \Rightarrow X)$$

On utilise dans le fait suivant la notation $\Lambda X A(X) = \forall X(X \Rightarrow A(X))$, qui aura une grande importance dans le chapitre suivant, ainsi que le type des booléens du système F : $Bool = \forall X(X \Rightarrow X \Rightarrow X)$.

Fait 37 *On montre :*

1. $\vdash \forall P, Q(P \| Q \subset Bool)$
2. $\vdash \forall P, Q(P \Rightarrow \Lambda X \Lambda Y X \subset P \| Q)$
3. $\vdash \forall P, Q(Q \Rightarrow \Lambda X \Lambda Y Y \subset P \| Q)$
4. $\vdash \forall P(P \Rightarrow \Lambda X \Lambda Y X =_{\subset} P \| (\neg P))$
5. $\vdash \forall P(\neg P \Rightarrow \Lambda X \Lambda Y Y =_{\subset} P \| (\neg P))$

preuve:

1. Immédiat à l'aide des axiomes de Co-Contra variance et de la borne inférieure.
2. Idem.
3. Idem.
4. D'après 2., il suffit sous l'hypothèse P de montrer $(P \| \neg P \subset \Lambda X \Lambda Y X)$. Cela est immédiat en remarquant que $P \vdash Y \subset \neg P \twoheadrightarrow X$.
5. Idem, en utilisant de plus l'axiome de Mitchell.

□

Ces propriétés permettent d'expliquer le nom "décideur" que nous avons donné à cet opérateur : il correspond au type du booléen qui code la valeur de vérité d'un énoncé calculable.

Types inductifs

Nous donnons ici une nouvelle définition, différente de l'originale ([50]) pour le type inductif défini par un opérateur.

Définition 34 (Type inductif) *Le* ***type inductif*** *associé à un opérateur F est défini par :*

$$\mu X.F(X) = \forall X.(\forall A(A \subset X \Rightarrow F(A) \subset X) \twoheadrightarrow X)$$

On définit également la croissance d'un opérateur :

Définition 35 (Opérateur croissant) *Le prédicat "F est un* ***opérateur croissant****" est défini par :*

$$Cresc(F) = \forall X, Y(X \subset Y \Rightarrow F(X) \subset F(Y))$$

On saura que $\mu XF(X)$ est effectivement un plus petit point fixe dans le cas d'un opérateur F croissant, mais dans le cas général, on sait uniquement que $F(\mu X.F(X)) \subset \mu X.F(X)$. C'est ce que nous allons voir à l'aide des faits suivants.

Fait 38 *On montre :*

(1) $\vdash \forall F \forall X(X \subset \mu X.F(X) \Leftrightarrow \forall Y(\forall A(A \subset Y \Rightarrow F(A) \subset Y) \Rightarrow X \subset Y)$
(2) $\vdash \forall F \forall X(X \subset \mu X.F(X) \Rightarrow F(X) \subset \mu X.F(X))$
(3) $\vdash \forall F \forall X(\forall A(A \subset X \Rightarrow F(A) \subset X) \Rightarrow \mu X.F(X) \subset X)$

preuve:
- (1) : on montre plus généralement que $\vdash \forall P \forall X(X \subset \forall Y(P(Y) \twoheadrightarrow Y) \Leftrightarrow \forall Y(P(Y) \Rightarrow X \subset Y)$, qui est immédiat.
- (2) : s'obtient à partir de (1) en réinjectant $X \subset Y$ dans la prémisse.
- (3) : est une conséquence du fait $\forall P \forall X(P(X) \Rightarrow \forall X(P(X) \twoheadrightarrow X \subset X))$, qui est immédiat.

□

Le fait suivant donne sens à l'appellation "plus petit point fixe" :

Fait 39 *On montre :*

(1) $\vdash \forall F(F(\mu X.F(X)) \subset \mu x.F(X))$
(2) $\vdash \forall F(Cresc(F) \Rightarrow F(\mu X.F(X)) =_{\subset} \mu x.F(X))$
(3) $\vdash \forall F(Cresc(F) \Rightarrow \forall I(F(I) =_{\subset} I \Rightarrow \mu x.F(X) \subset I))$

preuve:
- (1) est une conséquence du fait précédent (2).
- (2) : il suffit de montrer que $\mu x.F(X) \subset F(\mu X.F(X))$, et d'après le fait précédent (3), il suffit de montrer que $\forall A(A \subset F(\mu X.F(X)) \Rightarrow F(A) \subset F(\mu X.F(X))$. En supposant que $A \subset F(\mu X.F(X))$, on a par (1) $A \subset \mu X.F(X)$, et donc par $Cresc(F)$ on obtient la conclusion.
- (3) : par le fait précédent (3), il suffit de montrer sous les hypothèses $Cresc(F)$ et $F(I) =_{\subset} I$ que $\forall A(A \subset I \Rightarrow F(A) \subset I)$, ce qui est immédiat.

□

3.8 Conclusion

Avec le travail effectué dans ce chapitre et le précédent, nous avons une idée un peu plus claire de ce que peut modéliser le système ST, et de ce qu'il peut exprimer. Nous allons, dans le chapitre suivant, démontrer le théorème de préservation du type par réduction, en utilisant des techniques qui seront utiles dans les sections qui concerneront l'élaboration du langage de programmation proprement dit bâti au-dessus de ST.
Remarque: La totalité des énoncés de ce chapitre a été prouvée dans l'assistant de preuve PhoX ([47]), et peuvent être téléchargés à l'URL suivante : http ://www.lama.univ-savoie.fr/ ruyer/these

Chapitre 4

Préservation du type

Nous allons montrer dans ce chapitre le résultat énoncé et partiellement prouvé dans [49] : la préservation du type par réduction. Celle-ci n'est pas triviale, car le mélange du sous-typage et de l'omission de contenu algorithmique rend le système difficile à appréhender.

Nous mettrons donc en évidence les 3 mécanismes qui rentrent en jeu dans cette preuve :

- Premièrement (partie 1), et c'est un fait assez étonnant, le typage est en quelque sorte (modulo une hypothèse sur la forme de la dérivation, que nous appellerons *sans coupure essentielle*) équivalent au sous-typage dans le système ST : ceci repose sur un codage des termes dans les types. Nous montrons aussi que ce codage permet de transformer les relations de β et η réduction en relations de sous-typage.
- Deuxièmement (partie 2), nous montrons que toute dérivation peut être transformée en une dérivation dans laquelle le contexte a une certaine forme que nous appelons *c*ontexte précisé, et qui est sans coupure essentielle.
- Troisièmement (partie 3), on arrive à prouver de façon interne que, en supposant que les variables libres d'un terme sont "codées" par des singletons, alors ce terme est un singleton.
- La dernière partie (partie 4) combine les trois premières pour obtenir le résultat de préservation du type.

Notre preuve apporte cela de nouveau par rapport au papier original, d'une part que nous explicitons les mécanismes qui sous-tendent la preuve, et d'autre part que nous la complétons. Dans l'article [49] en effet, l'auteur s'appuie sur le Théorème 14 qui, s'il est vrai en l'absence des règles gauches *Coup* et *UG*, n'est plus aussi évident en leur présence ...Nous le remplaçons par deux versions :

- Nous montrons que le résultat reste vrai si les règles gauches restent cantonnées au-dessus de jugements logiques -c'est le sens de la notion de dérivation sans coupures essentielles- (proposition 43 page 64).
- Le théorème du type principal donne une version plus faible, en l'absence de contexte, du résultat (théorème 44 page 65).

4.1 Typage et sous-typage

L'objectif de cette section est de montrer comment typage et sous-typage interagissent dans le système ST. Nous commençons par donner la traduction sous forme de type des programmes. Puis, nous montrons comment d'un jugement de sous-typage on peut déduire un jugement de typage.
Nous introduisons ensuite une nouvelle notion, la notion de dérivation sans coupure essentielle, et montrons la réciproque (que d'un jugement de typage on peut déduire un jugement de sous-typage, qui est plus surprenante), sous l'hypothèse qu'on a une dérivation sans coupure essentielle.

4.1.1 λ-abstraction dans les types, et interprétation

Pour qui arrive à ce point de la thèse en ayant lu le chapitre de sémantique (2), la définition suivante ne sera pas étonnante :

Définition 36 (λ-abstraction de types) *On définit un nouvel opérateur de types :*

$$\Lambda X A(X) = \forall X (X \Rightarrow A(X))$$

Ceci nous permet de définir, avec l'abstraction de types, l'interprétation d'un terme :

Définition 37 (Interprétation de programme) *Soit t un terme de variables libres $x_1, \ldots, x_n$ et ϕ une application qui à chaque variable du λ-calcul associe un terme de type τ. L'**interprétation de** t **sous** ϕ, notée $|t|^\phi$ est définie par :*
- $|x|^\phi = \phi(x)$
- $|(u\ v)|^\phi = |u|^\phi[|v|^\phi]$
- $|\mathbf{fun}\ x\ \texttt{->}\ t|^\phi = \Lambda X.|t|^{\phi[x:=X]}$

β-expansion et η-réduction

Nous illustrons ici un intérêt de cette traduction : les relations de réduction se traduisent en relations de sous-typage. Plus que d'un intérêt anecdotique, ce fait relève d'un intérêt pratique certain puisqu'il joue un rôle clé dans la preuve de préservation du type. Avec les simples définitions données ci-dessus, et moyennant quelques lemmes techniques faciles, nous avons donc le résultat suivant (de [50]) :

Théorème 40 *Soit ϕ une interprétation, et t et t' deux programmes.*
- *Si $t \succ_\beta t'$, alors $|t|^\phi \subset |t'|^\phi$.*
- *Si $t \succ_\eta t'$, alors $|t'|^\phi \subset |t|^\phi$.*

preuve: On le montre d'abord pour les redex :
- $\Lambda X.F(X)[V] \subset F(V)$ car $\Lambda X.F(X) \subset V \Rightarrow F(V)$ et par le fait 28.1.

– $A \subset \Lambda X.A[X]$ est un peu plus délicat : on a $A \subset \forall X, Y((A \subset (X \Rightarrow Y)) \twoheadrightarrow (X \Rightarrow Y))$ de manière immédiate. En utilisant les axiomes de transitivité, de permutation de la flèche spéciale et l'axiome de Mitchell, on en déduit que $A \subset \forall X(X \Rightarrow \forall Y((A \subset (X \Rightarrow Y)) \twoheadrightarrow Y))$.

Pour généraliser à tous les cas on montre :

1. Si $A \subset A'$ et $B \subset B'$ alors $A[B] \subset A'[B']$, immédiat par le fait 28.2.
2. Si F minore G c'est à dire $\forall X(F(X) \subset G(X))$, alors $\Lambda X.F(X) \subset \Lambda X.G(X)$: immédiat par la co-variance de $\Rightarrow$.

On suppose $t \succ_\alpha t'$ ($\alpha = \beta$ ou η) :
– Si t est un redex, le résultat est vrai d'après ce qu'on vient de voir.
– Si $t = (u\ v)$ et $t' = (u'\ v)$ (resp. $t' = (u\ v')$) avec $u \succ_\alpha u'$ (resp. $v \succ_\alpha v'$), on conclut par induction et par 1..
– Si $t =$ `fun` x `->` u et $t' =$ `fun` x `->` u avec $u \succ_\alpha u'$, on a pour β par hypothèse d'induction pour tout X $|u|^{\phi[x:=X]} \subset |u'|^{\phi[x:=X]}$, et donc par 2. $|t|^\phi \subset |t'|^\phi$.

□

4.1.2 Équivalence entre typage et sous-typage

Dans cette sous-partie, sans précision supplémentaire, on considérera toujours l'interprétation ϕ associée à un contexte Γ définie par : si $x : A \in \Gamma$, alors $\phi(x) = A$. On montre d'abord qu'un jugement de typage peut se déduire d'un jugement de sous-typage, puis on montre l'inverse, dans le cas où la dérivation a une forme particulière (si elle n'utilise les règles gauches *Coup* et *UG* que d'une façon inessentielle pour le typage).

Du sous-typage vers le typage

L'objectif est de montrer la proposition suivante :

Proposition 41 *Pour tout contexte Γ, tout programme t et tout type A :*

$$\textit{Si } \Gamma \vdash |t|^\phi \subset A \textit{ , alors } \Gamma \vdash t : A$$

preuve: La preuve découle immédiatement du lemme 42 ci-dessous : on a par le lemme que $\Gamma \vdash t : |t|^\phi$. Si $\Gamma \vdash |t|^\phi \subset A$, alors par la règle $\subset$, on a le résultat. □

On montre donc le lemme suivant :

Lemme 42 *Pour tout contexte Γ et tout programme t dont les variables libres sont dans Γ :*

$$\Gamma \vdash t : |t|^\phi$$

preuve: Par induction sur la longueur du programme t :
– Si c'est une variable , c'est clair.
– Si c'est une application, on a le résultat par induction et par le fait 28 3.
– Si $t =$ `fun` x `->` u. On a, par hypothèse d'induction : $\Gamma, X : \tau, x : X \vdash u : |u|^{\phi[x:=X]}$ et donc par les règles d'introduction de $\Rightarrow$ et $\forall$, on a le résultat.

□

Du typage vers le sous-typage

Nous allons montrer la réciproque de la proposition 41, dans le cas où la dérivation a une forme particulière : les règles gauches n'interviennent qu'éventuellement au-dessus de déductions logiques. Précisons à l'aide d'une définition :

Définition 38 (Coupures essentielles) *On dit qu'une dérivation de typage est sans* ***coupures essentielles*** *si la dernière règle est telle que :*
- *C'est une instance de Ax^τ.*
- *C'est une instance d'une règle de typage, qui n'est pas pas UG, dont les prémisses sont sans coupures essentielles.*
- *C'est une instance d'une règle mixte, qui n'est pas Coup, et dont la prémisse de conclusion un jugement de typage est sans coupure essentielle, i.e. :*
  - *soit c'est une instance de $\subset$ dont la prémisse droite est sans coupure essentielle*
  - *soit c'est une instance de $\twoheadrightarrow_I$ dont la prémisse est sans coupure essentielle*
  - *soit c'est une instance de $\twoheadrightarrow_E$ dont la prémisse gauche est sans coupure essentielle*

On souhaite montrer la proposition suivante :

Proposition 43 *Pour tout contexte Γ, tout programme t et tout type A :*

$$\textit{Si } \Gamma \vdash t : A \textit{ sans coupures essentielles, alors } \Gamma \vdash |t|^\phi \subset A$$

preuve: Par induction sur la la longueur de la dérivation :
- Ax^τ : immédiat.
- $\forall_I$, $\forall_E$: immédiats par induction et sous-typage.
- $\Rightarrow_E^\tau$: Immédiat par 28 page 52.
- $\Rightarrow_I^\tau$: Par H.I :

$$\frac{\Gamma, x : A \vdash |u|^{[x:=A]} \subset B}{\Gamma, x : A \vdash A \Rightarrow |u|^{[x:=A]} \subset A \Rightarrow B}$$

  Et, de plus, par l'axiome "Le faux est vide" et par la contravariance de $\Rightarrow$:

$$\Gamma, A \subset \bot^\tau \vdash (A \Rightarrow |u|^{[x:=A]}) \subset (A \Rightarrow B)$$

  Donc, par le 4 du fait 29 page 53 :

$$\Gamma \vdash (A \Rightarrow |u|^{[x:=A]}) \subset (A \Rightarrow B)$$

  Et on conclut avec une instance de l'axiome de la borne inférieure (Minorant).
- $\subset$: remplacé par une instance de la transitivité.
- $\twoheadrightarrow_I$: remplacé par une instance de l'axiome droit de la flèche spéciale.
- $\twoheadrightarrow_E$: remplacé par une instance de l'axiome gauche de la flèche spéciale.

□

Les deux propositions 43 et 41 sont des quasi-réciproques. Outre leur intérêt futur dans la preuve de préservation du type, nous souhaitons également montrer le résultat suivant :

Théorème 44 (Théorème du type principal) *Pour tout programme t et tout type A :*

$$\vdash t : A \text{ si et seulement si } \vdash |t| \subset A$$

Nous nommons ce résultat "Théorème du type principal" car son sens est que dans le système ST, tout programme admet un plus petit type (au sens de l'inclusion).
Comme nous l'avons vu, cela repose d'abord sur le fait de pouvoir transformer une dérivation en une dérivation "équivalente" sans coupures essentielles. La dérivation ne sera pas vraiment équivalente, car elle nécessitera une transformation de contextes, que nous allons étudier maintenant (le théorème sera prouvé à la section 4.2.1).

4.2 Transformations de contextes

Les contextes modifiés sont élaborés en remplaçant chaque déclaration de typage par une déclaration "plus précise" qui établit qu'une variable de programme est non seulement dans un type, mais aussi dans un singleton qui est dans ce type. Cette notion était déjà présente dans [49], Fait 32, mais apparaît ici comme doublement essentielle pour montrer la préservation du type.

Définition 39 (Contextes précisés) *Soit Γ un contexte. On appelle* ***contexte précisé*** *issu de Γ, noté $Prec(\Gamma)$ le contexte obtenu en remplaçant toutes les déclarations de variables de programme de la forme $x : A$ par la suite de définitions : $X : \tau, x : X, \mathcal{S}(X), X \subset A$.*

Le premier lemme concernant les contextes précisés établit la possibilité d'éliminer une déclaration précisée :

Lemme 45 *La règle suivante est admissible :*

$$\frac{\Gamma, X : \tau, x : X, \mathcal{S}(X), X \subset A, \Delta \vdash t : B \qquad \Gamma, \Delta \vdash B : \tau}{\Gamma, x : A, \Delta \vdash t : B}$$

preuve: On montre tout d'abord que les deux règles suivantes sont admissibles :

1. $$\frac{\Gamma, P, Q, \Delta \vdash t : B}{\Gamma, P \wedge Q, \Delta \vdash t : B}$$

2. $$\frac{\Gamma, x : A, P, \Delta \vdash t : B}{\Gamma, x : A \restriction P, \Delta \vdash t : B}$$

Pour 1., c'est évident. Pour 2., on en déduit d'abord que $\Gamma, x : A, \Delta, P \vdash t : B$, et donc $\Gamma, x : A, \Delta \vdash t : P \twoheadrightarrow B$.
On construit ensuite la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : A, \Delta \vdash t : P \twoheadrightarrow B} \quad \dfrac{}{\Gamma, x : A, \Delta \vdash A \restriction P \subset A}\,(a)}{\Gamma, x : A \restriction P, \Delta \vdash t : P \twoheadrightarrow B}\,Coup \quad \dfrac{\dfrac{}{\Gamma, x : A \restriction P, \Delta \vdash x : A \restriction P}\,Ax^{\tau}}{\Gamma, x : A \restriction P, \Delta \vdash P}\,(b)}{\Gamma, x : A \restriction P, \Delta \vdash t : B}\,{\twoheadrightarrow_E}$$

(a) et (b) utilisant le fait 30.
On en déduit que la règle suivante est admissible :

$$\dfrac{\Gamma, X : \tau, x : X, \mathcal{S}(X), X \subset A, \Delta \vdash t : B}{\Gamma, x : X \restriction (\mathcal{S}(X) \wedge X \subset A), \Delta \vdash t : B}$$

On conclut ainsi (en omettant Δ pour des raisons de place) :

$$\dfrac{\dfrac{\Gamma, X : \tau, x : X \restriction (\mathcal{S}(X) \wedge X \subset A) \vdash t : B \quad \Gamma \vdash B : \tau}{\Gamma, x : \cup X(X \restriction (\mathcal{S}(X) \wedge X \subset A)) \vdash t : B}\,UG \quad \dfrac{}{\Gamma \vdash A \subset \cup Y(Y \restriction (\mathcal{S}(Y) \wedge Y \subset A))}\,Atom.}{\Gamma, x : A \vdash t : B}\,Coup$$

□

Nous souhaitons maintenant prouver que la règle ci-dessus est inversible, mais de plus qu'il y a un moyen d'obtenir une dérivation sans coupures essentielles.
On désire donc prouver le résultat suivant :

Lemme 46 *Si $\Gamma \vdash t : B$, alors il existe une dérivation de $Prec(\Gamma) \vdash t : B$ qui est sans coupure essentielle.*

preuve: On montre plus précisément : Si $\Gamma \vdash t : B$, alors il existe une dérivation de $Prec(\Gamma) \vdash t : B$ qui est sans coupure essentielle, et si $\Gamma \vdash P$, alors il existe une dérivation de $Prec(\Gamma) \vdash P$. Comme d'habitude, on fait une induction sur la preuve :
– Règles logiques : inchangées.
– Ax^{τ} : Est remplacé par une instance de Ax^{τ} suivie d'une instance de la Transitivité.
– $\Rightarrow_I^{\tau}$: Par H.I, on a :

$$\dfrac{Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset A \vdash u : B}{Prec(\Gamma) \vdash \texttt{fun}\ \ x\ \ \texttt{->}\ \ u : \forall X(X \twoheadrightarrow \mathcal{S}(X) \twoheadrightarrow (X \subset A) \Rightarrow B)}\,{\Rightarrow_I^{\tau}, \forall_I^{\tau}}$$

Et, par le fait 32 3. :

$$Prec(\Gamma) \vdash \forall X(X \twoheadrightarrow \mathcal{S}(X) \twoheadrightarrow (X \subset A) \Rightarrow B) \subset (A \Rightarrow B)$$

D'où, par $\subset$:

$$Prec(\Gamma) \vdash \texttt{fun}\ x\ \texttt{->}\ u : A \Rightarrow B$$

- $\Rightarrow_E^\tau$, $\forall_I^\tau$, $\forall_E^\tau$: inchangées.
- β-règles : inchangées.
- $\subset$, $\twoheadrightarrow_I$, $\twoheadrightarrow_E$: inchangées.
- $Renf$:

  On a par H.I :

  $$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), (X \subset A) \vdash A \subset B$$

  On construit la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), (X \subset A) \vdash X \subset B}\,Ax^o, \subset}{Prec(\Gamma), X : \tau, x : X \vdash X \subset (\mathcal{S}(X) \twoheadrightarrow (X \subset A) \twoheadrightarrow B)}\,(a)}{Prec(\Gamma), X : \tau \vdash X \subset (\mathcal{S}(X) \twoheadrightarrow (X \subset A) \twoheadrightarrow B)}\,Renf}{Prec(\Gamma), X : \tau \vdash (X \restriction (\mathcal{S}(X) \wedge X \subset A)) \subset B}\,(b)}{Prec(\Gamma) \vdash \forall X (X \restriction (\mathcal{S}(X) \wedge X \subset A)) \subset B}\,\forall_I^o}{Prec(\Gamma) \vdash \cup X (X \restriction (\mathcal{S}(X) \wedge X \subset A)) \subset B}\,(c)}{Prec(\Gamma) \vdash A \subset B}\,(d)$$

  Avec :
  - (a) se dérivant avec les axiomes de la flèche spéciale.
  - (b) se dérivant avec les axiomes de la flèche spéciale et le fait 30.
  - (c) est immédiat par la définition de $\cup$.
  - (d) utilisant l'axiome d'atomiticité.
- $Coup$: Par H.I. :

  $$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset A, Prec(\Delta) \vdash t : B$$

  On en déduit, par permutation et affaiblissement :

  $$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset A', Prec(\Delta), X \subset A \vdash t : B$$

  est dérivable sans coupure essentielle, et donc

  $$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset A', Prec(\Delta) \vdash t : (X \subset A) \twoheadrightarrow B$$

  De plus, par H.I., on a aussi

  $$Prec(\Gamma), Prec(\Delta) \vdash A' \subset A$$

  On en déduit donc :

  $$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset A', Prec(\Delta) \vdash X \subset A$$

et finalement par une instance de $\twoheadrightarrow_E$:

$$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset A', Prec(\Delta) \vdash t : B$$

– UG : Par H.I :

$$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), y : s, X \subset A(y), Prec(\Delta) \vdash t : B$$

On en déduit, comme y n'est pas libre dans $Prec(\Delta)$:

$$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), Prec(\Delta), y : s, X \subset A(y) \vdash t : B$$

et donc :

$$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), Prec(\Delta) \vdash t : \forall y(X \subset A(y) \twoheadrightarrow B)$$

Par affaiblissement :

$$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset \cup y A(y), Prec(\Delta) \vdash t : \forall y(X \subset A(y) \twoheadrightarrow B)$$

Et, par définition de singleton, on a

$$Prec(\Gamma), X : \tau, x : X, \mathcal{S}(X), X \subset \cup y A(y), Prec(\Delta) \vdash \exists y(X \subset A(y))$$

La conclusion est ensuite (quasi) immédiate.

□

Remarque: Le problème avec RG serait par exemple une dérivation du type suivant :

$$\dfrac{\dfrac{\Gamma, X : s, x : F(X) \vdash t : B}{\Gamma, x : \cup X F(X) \vdash t : B}\, RG \qquad \Gamma \vdash A' \subset \cup x F(x)}{\Gamma, x : A' \vdash t : B}\, Coup$$

4.2.1 Preuve du théorème du type principal

Nous allons maintenant prouver le théorème 44 :
preuve:
– "si" : est une conséquence directe de la proposition 41.
– "seulement si" : On suppose que $\vdash t : A$. Par le lemme 46, comme $\Gamma = []$, il en existe une preuve sans coupure essentielle. Et donc, par la proposition 43, $\vdash |t| \subset A$.

□

A titre d'illustration, montrons comment cela "fonctionne" sur l'exemple traditionnel des entiers de Church. On pose :
– $Nat = \forall X(X \Rightarrow (X \Rightarrow X) \Rightarrow X)$
– $Deux =$ `fun` x `->` `fun` f `->` $(f\ (f\ x))$
On veut montrer $\vdash |Deux| \subset Nat$, avec $|Deux| = \forall X(X \Rightarrow \forall F(F \Rightarrow F[F[X]]))$.
En posant $F = X \Rightarrow X$, on a :
$\vdash F[X] \subset X$, d'où $\vdash F[F[X]] \subset X$, et on conclut facilement.

4.3 Singletonicité des λ-termes

Par ce néologisme, il est vrai assez inélégant, nous entendons le fait suivant :

Théorème 47 (Singletonicité) *Soit Γ un contexte, t un terme, et ϕ une interprétation telle que pour toute variable libre x de t, si $\phi(x) = A$, $\Gamma \vdash \mathcal{S}(A)$. Alors :*

$$\Gamma \vdash \mathcal{S}(|t|^{\phi})$$

Comme le fait d'être un singleton est préservé par application (axiome 4.*b*), il faut vérifier que l'abstraction *réduite aux cas de typage de termes* préserve la singletonicité : c'est le sens de la proposition 52, qui prélude à la preuve du théorème donnée à la section 4.3.2. Nous aurons besoin pour ce travail de quelques propriétés sur les singletons, que nous allons maintenant présenter.

4.3.1 Autres définitions de singleton

On définit deux nouvelles notions de singletons, la première pouvant être vue comme un cas particulier de la définition générale, et la seconde étant liée à la symétrie de la relation qui quotiente Λ :

Définition 40 *On définit le prédicat "être un singleton pour la réunion" par :*

$$\mathcal{S}_{\cup}(A) = A \neq \emptyset \wedge \forall X, Y (A \subset X \cup Y \Rightarrow (A \subset X \vee A \subset Y))$$

On définit le prédicat "être un singleton pour l'inclusion" (on pourrait aussi dire "être un atome") par :

$$\mathcal{S}_{\subset}(A) = A \neq \emptyset \wedge \forall X (X \subset A \Rightarrow X \neq \emptyset \Rightarrow A =_{\subset} X)$$

En fait ces trois définitions sont équivalentes :

Proposition 48 *On prouve l'équivalence suivante :*

$$\forall A (\mathcal{S}(A) \iff \mathcal{S}_{\cup}(A) \iff \mathcal{S}_{\subset}(A))$$

La preuve utilise le lemme suivant :

Lemme 49 *Un type non vide contient un singleton :*

$$\vdash \forall A (A \neq \emptyset \Rightarrow \exists x (\mathcal{S}(x) \wedge x \subset A))$$

preuve: On montre la contraposée : $\forall x (\mathcal{S}(x) \Rightarrow x \not\subset A) \Rightarrow A = \emptyset$: sous l'hypothèse $\forall x (\mathcal{S}(x) \Rightarrow x \not\subset A)$, et en utilisant le fait 32 (1), il suffit de montrer $\forall x (\mathcal{S}(x) \Rightarrow x \subset A \Rightarrow x \subset \bot^{\tau})$, ce qui est trivial. □

On est maintenant en mesure de prouver la proposition 48 :
preuve: On montre l'équivalence par circularité (on ne montrera pas à chaque fois $A \neq \emptyset$ qui est évidente) :

- $\mathcal{S}(A) \Rightarrow \mathcal{S}_\cup(A)$: on suppose $\mathcal{S}(A)$ et $A \subset X \cup Y$, et il faut en déduire $A \subset X \vee A \subset Y$. Par le fait 36, on a $A \subset X \widehat{\cup} Y = \cup Z(Z \restriction (Z =_\subset X \vee Z =_\subset Y)$. Par définition de $\mathcal{S}$, on a donc $\exists Z(A \subset (Z \restriction (Z =_\subset X \vee Z =_\subset Y)$. Sous l'hypothèse $A \subset Z \restriction (Z =_\subset X \vee Z =_\subset Y)$, et avec le fait 29 on en déduit $Z =_\subset X \vee Z =_\subset Y$. Si $Z = X \vee Z = Y$, alors $A \subset Z$ par le fait 30.
- $\mathcal{S}_\cup(A) \Rightarrow \mathcal{S}_\subset(A)$: on suppose $\mathcal{S}_\cup(A)$, $X \neq \emptyset$ et $X \subset A$, et il faut en déduire $X = A$; il suffit donc d'en déduire $A \subset X$. L'axiome du complément nous donne $A \subset X \cup X^c$, et donc par $\mathcal{S}_\cup(A)$, $A \subset X \vee A \subset X^c$. On raisonne par cas :
  - Si $A \subset X$, c'est fini.
  - Si $A \subset X^c$, on a par transitivité de $\subset$: $X \subset X^c$, et donc par le fait 35 on a $X = \emptyset$: contradiction
- $\mathcal{S}_\subset(A) \Rightarrow \mathcal{S}(A)$: par le lemme 49, et sous l'hypothèse $\mathcal{S}_\subset(A)$, on a $\exists x(\mathcal{S}(x) \wedge x \subset A)$. En supposant $\mathcal{S}(x) \wedge x \subset A$ (h), et en utilisant $\mathcal{S}_\subset(A)$, on en déduit que $A \subset x$ (c), et donc que $\mathcal{S}(A)$: effectivement, si $A \subset \cup_X(B(X))$ on a par (h) et la définition de $\mathcal{S}$, $\exists y(x \subset B(y))$, et donc par (c), $\exists y(A \subset B(y))$.

□

On prouve de plus deux lemmes qui apportent, nous l'espérons, une meilleure compréhension de la preuve de 52.

Lemme 50 (Une autre caractérisation de singleton) *On prouve :*

$$\vdash \forall A(A \neq \emptyset \Rightarrow \forall x(\mathcal{S}(x) \Rightarrow x \subset A \Rightarrow A \subset x) \Rightarrow \mathcal{S}(A))$$

preuve: On met en hypothèse $A \neq \emptyset$ (a), $\forall x(\mathcal{S}(x) \Rightarrow x \subset A \Rightarrow A \subset x)$ (b), et $A \subset \cup y F(y)$ (c). Il suffit de montrer $\exists y(A \subset F(y))$.
Par le lemme 49, et l'hypothèse (a), il suffit de le montrer sous l'hypothèse $\mathcal{S}(x) \wedge x \subset A$ (d). En utilisant (d) et (b), on obtient $\exists y(A \subset F(y))$. □

Lemme 51 (η-réduction pour les singletons) *On prouve :*

$$\vdash \forall A(\mathcal{S}(A) \Rightarrow \Lambda X A[X] \subset A)$$

preuve: Par l'axiome d'extensionnalité, il suffit de prouver, sous l'hypothèse $\mathcal{S}(A)$, que $\forall x((\Lambda X A[X])[x] \subset A[x])$, ce qui est immédiat par le théorème 40. □

4.3.2 Preuve du théorème de singletonicité

On montre tout d'abord la proposition suivante (de [49]) :

Proposition 52 *L'abstraction de type préserve le fait d'être un singleton :*

$$\vdash \forall A(\Lambda X A(X) \neq \emptyset \Rightarrow \forall X(\mathcal{S}(X) \Rightarrow \mathcal{S}(A(X))) \Rightarrow \mathcal{S}(\Lambda X A(X)))$$

preuve: On suppose $\Lambda X A(X) \neq \emptyset$ (a) et $\forall X(\mathcal{S}(X) \Rightarrow \mathcal{S}(A(X))$ (b), et il faut prouver $\mathcal{S}(\Lambda X A(X))$. A l'aide du lemme 50 il suffit de montrer que $\forall x(\mathcal{S}(x) \Rightarrow x \subset \Lambda X A(X) \Rightarrow \Lambda X A(X) \subset x)$. On met en hypothèses $\mathcal{S}(x)$ (c) et $x \subset \Lambda X A(X)$ (d). Ajoutons l'hypothèse $\mathcal{S}(z)$ (e). Par les propriétés de l'application (fait 28), et le théorème 40 : $x[z] \subset \Lambda X A(X)[z] \subset A(z)$. Par (b) et (e), on a $\mathcal{S}(A(z)$, et par l'axiome de conservation des singletons, $x[z] \neq \emptyset$. Par le lemme 48, il vient $\mathcal{S}_{\subset}(A(z))$, donc $A(z) \subset x[z]$. On en déduit $\Lambda X A(X) \subset \forall Z(\mathcal{S}(Z) \twoheadrightarrow Z \Rightarrow x[Z])$.
On a de plus en utilisant le fait 29 (tiers exclu sur $\mathcal{S}(Z)$), et les propriétés de l'opérateur restriction (fait 30), $\forall Z(\mathcal{S}(Z) \twoheadrightarrow Z \Rightarrow x[Z]) \subset \forall Z((Z \restriction \mathcal{S}(Z)) \Rightarrow x[Z])$, et donc $\Lambda X A(X) \subset \Lambda Z x[Z]$, en utilisant le fait 32. On utilise maintenant le lemme 51, qui donne $\Lambda X A(X) \subset x$. □

On montre maintenant le théorème 47 qui exprime qu'on peut prouver que l'interprétation d'un terme, sous l'hypothèse que ses variables libres sont déclarées comme ayant le type d'un singleton, est un singleton :
preuve: Par induction sur la longueur de t :
- Si t est une variable, c'est immédiat par hypothèse.
- Si t est une application c'est immédiat par l'axiome de conservation des singletons.
- Si $t = \texttt{fun}\ x\ \texttt{->}\ u$. On a par le lemme 42, $\Gamma \vdash t : |t|$, et donc $|t| \neq \emptyset$ par le fait 29. On a $|t| = \Lambda X |u|^{[x:=X]}$. Pour utiliser le lemme 52, il suffit donc de montrer : $\Gamma \vdash \forall X(\mathcal{S}(X) \Rightarrow \mathcal{S}(|u|^{[x:=X]})$, ce qui est vrai par hypothèse d'induction.

□

Ce théorème, outre son intérêt dans la preuve de préservation du type, présente un intérêt propre car il montre que le système ST permet effectivement de cerner avec précision le "comportement" des types qui représentent des termes, au niveau de leurs propriétés géométriques.

4.4 Préservation du type

Voici venu le temps d'énoncer et prouver le résultat qui forme le titre de ce chapitre :

Théorème 53 (Préservation du type) *Le système ST possède la préservation du type par $\beta\eta$-équivalence :*

$$Si\ \Gamma \vdash t : A\ et\ t' =_{\beta\eta} t,\ alors\ \Gamma \vdash t' : A$$

preuve: Disposant déjà du théorème 40, il suffit de le montrer pour la β-réduction et la η-expansion.
- Si $\Gamma \vdash t : A$, alors d'après le lemme 46, il existe une dérivation de $Prec(\Gamma)$ sans coupure essentielle, et donc par la proposition 43, $Prec(\Gamma) \vdash |t| \subset A$. De plus, par le théorème 47 de singletonicité, $Prec(\Gamma) \vdash \mathcal{S}(|t|)$, et $Prec(\Gamma) \vdash \mathcal{S}(|t'|)$. Par le théorème 40, $Prec(\Gamma) \vdash |t| \subset |t'|$, et par la proposition 48, on en déduit $Prec(\Gamma) \vdash |t'| \subset |t|$, et donc $Prec(\Gamma) \vdash |t'| \subset A$. Par le lemme 45, on en déduit $\Gamma \vdash t' : A$.

– La preuve est analogue pour η.

□

Nous avons maintenant terminé d'exposer les propriétés de base du système ST. Dans les deux prochains chapitres, nous exposerons encore quelques résultats personnels sur ce système.

Chapitre 5

Expressivité de ST

Après avoir vu et prouvé dans le chapitre précédent les résultats sur le système ST énoncés par son concepteur, nous présentons ici nos propres résultats théoriques. Ils illustrent bien l'originalité de ce système, car nous n'en connaissons pas d'autres qui permettent d'exprimer simultanément ce genre de propriétés. Nous nous sommes intéressés en particulier à quatre thèmes usuels du domaine :

- Le tiers exclu, ou possibilité de calculer une preuve de A à partir d'une preuve de $\neg\neg A$, pour tout énoncé A.
- L'indécidabilité, ou la possibilité d'exprimer dans un système formel un énoncé "vrai" mais non "calculable".
- La normalisabilité, ou propriété d'arrêt du calcul.
- La résolubilité, ou possibilité de résoudre une équation de la forme :

$$(a \quad x_1 \ldots \quad x_n) = b$$

  a et b étant fixés.

Nous montrons ici que :

- Il n'y a pas d'espoir de calculer le tiers exclu dans ST (i.e. on a une preuve formelle dans ST du fait que $\forall X(\neg\neg X \Rightarrow X) \subset \emptyset$ sous réserve d'avoir un modèle non trivial).
- On exprime à l'intérieur du système - ce qui montre sa puissance-, par un argument diagonal, l'existence d'énoncés indécidables.
- Il existe un "type des normalisables" qui permet d'exprimer que les types de données usuels sont normalisables, et aussi de parler des termes normalisables.
- Il existe un "type des résolubles" qui permet d'exprimer qu'un type ne contient que des termes résolubles, et qu'un terme est résoluble.

Tous ces résultats nécessitent cependant un axiome de plus, par une remarque d'ordre sémantique qui a naturellement sa contrepartie syntaxique : d'après le chapitre 2, le treillis quasi-trivial $\{\bot; \top\}$ est un modèle de ST au sens du lemme d'adéquation 22 page 48 du chapitre 3, et donc si on veut dire des choses plus intéressantes, on doit travailler dans un système plus riche, $ST_{\neg TRIV}$que nous présentons dans la section suivante.

5.1 Modèles non triviaux

5.1.1 Système $ST_{\neg TRIV}$

Le système $ST_{\neg TRIV}$ est le système ST auquel on a rajouté un axiome qui exprime l'existence de deux singletons distincts :

Axiome 3 (Axiome de non-trivialité) *L'axiome suivant exprime l'existence d'au moins deux singletons :*

$$(\neg TRIV) \qquad \exists X, Y((X \neq Y) \wedge \mathcal{S}(X) \wedge \mathcal{S}(Y))$$

Le fait suivant exprime que les résultats principaux de ST restent valides :

Théorème 54

1. *Le lemme d'adéquation (lemme 22 page 48 du chapitre 3), reste vrai si $\overline{\tau}$ est non trivial.*
2. *$ST_{\neg TRIV}$ vérifie la préservation du type (théorème 53 page 71 du chapitre 4).*

preuve:

1. La seule chose à vérifier est que l'interprétation de l'axiome $\neg TRIV$ est $\top$, ce qui est immédiat par définition de l'interprétation.
2. La preuve du théorème de préservation du type ne fait pas intervenir de propriétés logiques (on peut d'ailleurs en déduire que l'on peut ajouter autant d'axiomes logiques sémantiquement vrais que l'on veut sans le modifier).

□

5.1.2 Distinguabilité des booléens

Nous mettons ici le premier résultat "pratique" qui interviendra par la suite, et qui est d'importance : les interprétations des booléens (ou "brancheurs") vrais et faux du λ-calcul sont distinctes dans $ST_{\neg TRIV}$.
On commence par un lemme qui est en fait équivalent à l'axiome $\neg TRIV$:

Lemme 55 *On montre :*

$$\vdash \neg\forall X, Y(\mathcal{S}(X) \Rightarrow \mathcal{S}(Y) \Rightarrow (X \subset Y))$$

preuve: Il est immédiat que $\forall X, Y(\mathcal{S}(X) \Rightarrow \mathcal{S}(Y) \Rightarrow (X \subset Y))$ entraîne la négation de l'axiome de non trivialité. □

Lemme 56 *Les booléens sont distincts :*

$$\vdash (\Lambda X.\Lambda Y.X \subset \Lambda X.\Lambda Y.Y) \Rightarrow \perp^o$$

preuve: On met en hypothèse $\Lambda X.\Lambda Y.X \subset \Lambda X.\Lambda Y.Y$ (a), $\mathcal{S}(x)$ (b), et $\mathcal{S}(y)$ (c). Par la croissance de l'application (fait 28 page 52 ch. 3), on a : $\Lambda X.\Lambda Y.X[x][y] \subset \Lambda X.\Lambda Y.Y[x][y]$. Par le théorème 47 page 69 du chapitre 4, (a) et (b), on a $\mathcal{S}(\Lambda X.\Lambda Y.X[x][y])$ et $\mathcal{S}(\Lambda X.\Lambda Y.Y[x][y])$. Par le théorème 40 page 62 du chapitre 4, on a $\Lambda X.\Lambda Y.X[x][y] \subset x$ et $\Lambda X.\Lambda Y.Y[x][y] \subset y$. Et, par la définition de $\mathcal{S}_\subset$, et la proposition 48 page 69, il vient donc, avec (a) et (b) : $x \subset y$. Et donc $\forall x, y(\mathcal{S}(x) \Rightarrow \mathcal{S}(y) \Rightarrow x \subset y)$, ce qui par le lemme précédent entraîne $\perp^o$. □

5.2 Existence d'énoncés indécidables

On montre ici qu'en interne dans le système $ST_{\neg TRIV}$, qu'on ne peut pas "tout calculer" dans le sens suivant : il n'existe pas de modèle de ST muni "d'oracles" (programmes qui calculent la valeur de vérité d'un énoncé) pour tout énoncé.
On utilise pour cela le décideur $\|$ présenté à la définition 33 page 58 : c'est effectivement cet opérateur qui permet de calculer la valeur de vérité d'un énoncé P, qui est $P\|\neg P$, d'après le fait 37 page 58.
On définit un type dont les habitants éventuels sont les programmes qui calculent la valeur de vérité d'un prédicat P sur les singletons :

$$\forall X(\mathcal{S}(X) \twoheadrightarrow X \Rightarrow (P(X)\|\neg P(X)))$$

Pour montrer qu'il n'existe pas de tel programme pour tous les prédicats, il suffit de montrer qu'il est faux que ce type soit non vide pour tous les prédicats ; c'est ce qu'exprime la proposition suivante :

Proposition 57 *On montre :*

$$\vdash \forall^{\tau\to o} P(\quad \forall X(\mathcal{S}(X) \twoheadrightarrow X \Rightarrow (P(X)\|\neg P(X))) \neq \emptyset \quad) \Rightarrow \perp$$

preuve: On suppose $\forall P(\forall X(\mathcal{S}(X) \twoheadrightarrow X \Rightarrow (P(X)\|\neg P(X))) \neq \emptyset)$, et on prend $P(X) = X[X] \subset \Lambda X\Lambda Y.Y$. On en déduit par le lemme 49 page 69 du ch. 4 $\exists U(\mathcal{S}(U) \wedge U \subset \forall X(\mathcal{S}(X) \Rightarrow X \Rightarrow (P(X)\|\neg P(X))))$.
Sous l'hypothèse $\mathcal{S}(U) \wedge U \subset \forall X(\mathcal{S}(X) \Rightarrow X \Rightarrow (P(X)\|\neg P(X)))$, on en déduit par les propriétés de l'application $U[U] \subset (P(U)\|\neg P(U))$ et par l'axiome d'atomiticité $\mathcal{S}(U[U])$.
On effectue ensuite un tiers exclu sur le prédicat $P(U)$:

- Sous l'hypothèse $P(U)$, on en déduit par le fait 37 page 58 que $U[U] =_\subset \Lambda X\Lambda Y.X$, donc que $U[U] \subset \Lambda X\Lambda Y.X$, ce qui entraîne par $P(U)$ $\Lambda X\Lambda Y.X \subset \Lambda X\Lambda Y.Y$, qui entraîne $\perp$ par le lemme 56 page 74.
- Sous l'hypothèse $\neg P(U)$, on en déduit par le fait 37 page 58 que $U[U] \subset \Lambda X\Lambda Y.Y$, donc $P(U)$: contradiction.

□

Un autre intérêt de cette proposition réside dans sa preuve qui est "typique" dans le sens où il s'agit de ce cher argument diagonal, s'exprimant ici sous la forme "je suis une preuve

qu'il est vrai que si je m'applique à moi même je suis dans le faux". On pourrait constater qu'un simple argument de cardinalité suffirait à montrer sémantiquement l'impossibilité d'existence d'oracles mais ici il s'agit de montrer l'expressivité syntaxique du système. Nous allons par la suite traiter des problèmes plus "pratiques"...

5.3 Logique classique

Nous montrons dans cette section que le système $ST_{\neg TRIV}$ est inadapté à la logique classique : on peut en effet inférer dans ce système que $\forall X(\neg\neg X \Rightarrow X) \subset \bot^\tau$.
La preuve, relativement élémentaire, d'impossibilité de réaliser $\forall X(\neg\neg X \Rightarrow X)$ est aisément transposable en réalisabilité usuelle, et en voici une version informelle :
Soit $x \neq \emptyset$. On a donc $(x \Rightarrow \emptyset) = \emptyset$, d'où il vient $\neg\neg x = Tout$. Par conséquent, tout élément de $\forall X(\neg\neg X \Rightarrow X)$, est élément de $Tout \Rightarrow x$ pour tout x non vide. Un tel élément appliqué à l'identité (par exemple) se réduit donc à la fois à 0 et à 1, ce qui, modulo le fait que 0 et 1 sont distincts est impossible.
Il reste à vérifier que ce raisonnement est transposable dans le système ST, ce qui se fait sans trop de difficultés.
Un premier lemme permet d'utiliser l'application pour montrer la "vacuité" :

Lemme 58 *L'application est intègre, dans le sens où le séquent suivant est dérivable :*

$$\vdash \forall X, Y(X[Y] \subset \bot \iff (X \subset \bot \vee Y \subset \bot))$$

preuve:
- On suppose $X[Y] \subset \bot$. En utilisant le tiers exclu avec $X \subset \bot$, puis avec $Y \subset \bot$, il reste à montrer le résultat sous les hypothèses $(X \subset \bot) \Rightarrow \bot_o$ et $(Y \subset \bot) \Rightarrow \bot_o$.
  En utilisant le lemme 49 page 69 du ch. 4, et l'axiome de préservation des singletons par application, et la croissance de l'application, on en déduit $\bot_o$.
- Pour le sens contraire, il suffit de montrer d'après le fait 28 page 52 que $X \subset (Y \Rightarrow \bot^\tau)$. On raisonne par cas :
  - Si $X \subset \bot^\tau$, on a $X \subset (Y \Rightarrow \bot^\tau)$.
  - Si $Y \subset \bot^\tau$, l'axiome du vide donne $X \subset (\bot^\tau \Rightarrow \bot^\tau)$ et on a $(\bot^\tau \Rightarrow \bot^\tau) \subset (Y \Rightarrow \bot^\tau)$, et donc $X \subset (Y \Rightarrow \bot^\tau)$.

□

Le second lemme montre qu'il n'y a pas de programme qui envoie un type non vide dans le vide :

Lemme 59 *On montre :*

$$\vdash \forall X(X \neq \emptyset \Rightarrow (X \Rightarrow \bot^\tau) \subset \bot^\tau)$$

preuve: On met en hypothèse $X \neq \emptyset$, et en appliquant le lemme précédent on a :

$$\vdash (X \Rightarrow \bot^\tau)[X] \subset \bot^\tau \Leftrightarrow (X \subset \bot^\tau \vee ((X \Rightarrow \bot^\tau) \subset \bot^\tau))$$

On a de plus par le fait 28 page 52 $X \Rightarrow \bot^\tau[X] \subset \bot^\tau$, on en déduit donc
$X \subset \bot^\tau \vee ((X \Rightarrow \bot^\tau) \subset \bot^\tau)$. On raisonne ensuite par cas :
– Si $X \subset \bot^\tau$, alors par l'hypothèse $X \neq \emptyset$, on en déduit $\bot^o$, et donc $(X \Rightarrow \bot^\tau) \subset \bot^\tau$.
– Sinon, on a $(X \Rightarrow \bot^\tau) \subset \bot^\tau$, et c'est fini.

□

Le troisième lemme montre que tout programme habite la double négation d'un type non vide. On pose : $\top^\tau = \cup xx$.

Lemme 60 *On montre :*

$$\vdash \forall X(X \neq \emptyset \Rightarrow (\top^\tau \subset ((X \Rightarrow \bot^\tau) \Rightarrow \bot^\tau))$$

preuve: Par le lemme précédent, on a sous l'hypothèse $X \neq \emptyset$, que $(X \Rightarrow \bot^\tau) \subset \bot^\tau$, et donc $(\bot^\tau \Rightarrow \bot^\tau) \subset ((X \Rightarrow \bot^\tau) \Rightarrow \bot^\tau)$. Par l'axiome le faux est vide, on a $\forall Y(Y \subset \bot^\tau \Rightarrow \bot^\tau)$, ce qui nous donne donc le résultat. □

On a maintenant tout ce qu'il faut pour prouver le théorème :

Théorème 61 (Vacuité classique) *On montre :*

$$\vdash (\forall X(((X \Rightarrow \bot^\tau) \Rightarrow \bot^\tau) \Rightarrow X)) \subset \bot^\tau$$

preuve: On raisonne par l'absurde : on suppose $\forall X((X \Rightarrow \bot^\tau) \Rightarrow \bot^\tau) \neq \emptyset$. Par le lemme 49 page 69, il suffit de déduire $\bot^o$ sous les hypothèses $C \subset \forall X((X \Rightarrow \bot^\tau) \Rightarrow \bot^\tau)$ (h) et $\mathcal{S}(C)$. En utilisant l'axiome $\neg TRIV$, on met en hypothèse $\mathcal{S}(a)$, $\mathcal{S}(b)$ et $a \neq b$.
On a par (h) $C \subset ((a \Rightarrow \bot^\tau) \Rightarrow \bot^\tau) \Rightarrow a$ et $C \subset ((b \Rightarrow \bot^\tau) \Rightarrow \bot^\tau) \Rightarrow b$. De plus, par le lemme précédent, on a $\top^\tau \subset (a \Rightarrow \bot^\tau) \Rightarrow \bot^\tau$ et $\top^\tau \subset (b \Rightarrow \bot^\tau) \Rightarrow \bot^\tau$. Donc, $C \subset \top^\tau \Rightarrow a$ et $C \subset \top^\tau \Rightarrow b$. Par conséquent, en utilisant le fait 28 page 52, $C[a] \subset a$ et $C[a] \subset b$. Par l'axiome de conservation des singletons, on a $\mathcal{S}(C[a])$, et donc par la proposition 48 page 69, $a \subset C[a]$, et par suite $a \subset b$, ce qui entraîne $\bot^o$ par hypothèse. □
Nous n'avons pas identifié précisément où cela "coince" dans la définition du système. Cela constitue une piste de recherche. Nous pouvons envisager de voir les axiomes et règles qui sont vrais dans la sémantique de Jean-Louis Krivine ([32]) ou de Michel Parigot ([40]), travail que nous avons déjà effectué partiellement [51].
Après ces deux résultats que nous pouvons qualifier de "négatifs" (impossibilité de ...), nous voyons maintenant deux résultats "positifs", le premier étant d'importance puisqu'exprimant la terminaison des programmes.

5.4 Normalisabilité

L'objectif de cette section est de montrer le résultat suivant :

Théorème 62 *Il existe un type $\mathcal{N}$ tel que :*

1. *Pour tout terme t clos :*

$$t \text{ est normalisable si et seulement si } \vdash |t| \subset \mathcal{N}$$

2. *Pour tout type A :*

$$\text{Si } \vdash A \subset \mathcal{N}, \text{ alors pour tout terme } t, \text{ si} \vdash t : A, \text{ alors } t \text{ est normalisable.}$$

Les techniques que nous utilisons ne sont pas nouvelles. Elles s'inspirent directement de la preuve de forte normalisation du système F de J-Y Girard par candidats de réductibilité. Ce qui est nouveau, c'est de les utiliser au niveau formel.

5.4.1 Rappels sur les parties adéquates

Définition 41 (Couples adéquats, ensembles adaptés) *On dit qu'un couple de parties (N_0, N) de Λ est **adéquat** si :*

$$N_0 \subset (N \Rightarrow N_0) \subset (N_0 \Rightarrow N) \subset N$$

*On dit qu'un ensemble est bien **adapté** s'il est clos par intersection et par $| \Rightarrow |$.*

Notation 9 *Si N_0 et N sont deux parties de Λ, on note :*

$P(N/N_0) = \{X; X \ \beta\eta \text{ saturé et } N_0 \subset X \subset N\}$.

Le lemme suivant est bien connu :

Lemme 63 *Si (N_0, N) est adéquat, alors $P(N/N_0)$ est bien adapté.*

preuve: Il est clair que $P(N/N_0)$ est clos par intersection. Les propriétés de covariance et contravariance de $| \Rightarrow |$ assure la clôture par $| \Rightarrow |$. □

Posons :
– $N = \{t; t \text{ normalisable }\}$
– $N_0 = \{(x)t_1...t_n; x \text{ est une variable et } t_i \in N\}_{sat}$
(on remarque aisément que N est $\beta\eta$ saturé).

Fait 64 *Le couple (N_0, N) est adéquat.*

preuve: Il est clair que $N_0 \subset N$, et que $N_0 \subset N \Rightarrow N_0$. $N \Rightarrow N_0 \subset N_0 \Rightarrow N$ est immédiat par le comportement de la flèche (co- et contravariance).
$(N_0 \Rightarrow N) \subset N$ peut se montrer de la façon suivante : soit $f \in N_0 \Rightarrow N$, et $x \notin FV(f)$. On a $(f \ x) \in N$, et donc $f \in N$. □

5.4.2 Preuve du théorème

On pose :
– Le prédicat $Adequ(X, X_0)$, traduction directe de la définition 41 page 78, défini par :

$$Adequ(X, X_0) = X_0 \subset (X \Rightarrow X_0) \wedge (X \Rightarrow X_0) \subset (X_0 \Rightarrow X) \wedge (X_0 \Rightarrow X) \subset X$$

– Le prédicat $N(A)$, défini par :

$$N(A) = \forall X, X_0 (Adequ(X, X_0) \Rightarrow A \subset X)$$

– Le type $\mathcal{N}$, défini par :

$$\mathcal{N} = \cup X (X \restriction (\mathcal{S}(X) \wedge N(X)))$$

On commence par un lemme qui donne l'utilité du prédicat $N(A)$:

Lemme 65 *Pour tout type A, si $\vdash N(A)$, alors $\mathcal{I}(A) \subset N$.*

preuve: Par le fait 64 page 78 et les propriétés de l'interprétation, on a $\mathcal{I}(Adequ)(N, N_0) = \top$. Par le lemme d'adéquation 22 page 48, si on suppose $\vdash N(A)$, alors $\mathcal{I}(A) \subset X$ pour tout couple adéquat (X, X_0), et donc $\mathcal{I}(A) \subset N$. □
On peut maintenant montrer le théorème 62 page 78 :
preuve:
Montrons d'abord le point 2. On suppose $\vdash A \subset \mathcal{N}$ (a) et $\vdash t : A$. On en déduit par le lemme d'adéquation (lemme 22 page 48) que $t \in \mathcal{I}(A)$. Il suffit donc de montrer que $\mathcal{I}(A) \subset N$. D'après le lemme précédent, il suffit de montrer $\vdash N(A)$. On met en hypothèse $Adequ(X, X_0)$, et par le fait 32 page 55, il suffit de montrer sous les hypothèses supplémentaires $\mathcal{S}(Y)$ et $Y \subset A$ que $Y \subset X$. Par (a) on a $Y \subset \mathcal{N}$, et par définition de singleton, $\exists Z(Y \subset (Z \restriction (\mathcal{S}(Z) \wedge N(Z))))$. Par le fait 30 page 54 (propriétés de l'opérateur restriction) et la proposition 48 page 69, on en déduit $N(Y)$, et par l'axiome d'atomicité, $N(A)$.
Pour le point 1., le sens droite-gauche est une conséquence du 2. et du théorème 44 page 65. Le sens gauche-droite se montre par induction sur le longueur de la forme normale du terme.
Soit t normalisable, et t' sa forme normale. D'après le théorème 40 page 62, $\vdash |t| \subset |t'|$, et il suffit de montrer $\vdash N(|t'|)$. Montrons-le par induction sur la longueur de t'. Plus précisément, nous allons montrer le résultat suivant :

"Si θ est un terme normalisable, et si on suppose $\vdash Adequ(N, N_0)$, alors en notant $|\theta|$ l'interprétation de θ dans laquelle chaque variable libre est remplacée par N_0, on a $\vdash |\theta| \subset N$ ".

Posons $\theta = \lambda x_1 ... x_n.(x) s_1 ... s_m$, on a : $\vdash |\theta| \subset N_0 \Rightarrow ... N_0 \Rightarrow N_0[|s_1|]...[|s_m|]$. Par hypothèse d'induction, on a $\vdash |s_j| \subset N$ pour $i = 1, .., m$, et par croissance on a donc $\vdash |\theta| \subset N_0 \Rightarrow \ldots N_0 \Rightarrow N_0[N]...[N]$. Comme $\vdash N_0 \subset N \Rightarrow N_0$, on en déduit que $\vdash N_0[N] \subset N_0$, et donc par une récurrence immédiate que $\vdash N_0[N]...[N] \subset N$, d'où

$\vdash |\theta| \subset N_0 \Rightarrow ...N_0 \Rightarrow N$. Par application répétée de $\vdash N_0 \Rightarrow N \subset N$, il vient $\vdash |\theta| \subset N$.

On déduit du raisonnement précédent que $Adequ(N, N_0) \vdash |t'| \subset N$, et donc que $\vdash N(|\theta|)$. □

5.4.3 Un résultat sur les types à ∀ positifs.

On va montrer ici que, bien qu'apparemment l'on ne puisse rien dire en général des types du système F, (par exemple de $(\forall XX \Rightarrow \forall XX)$, qui est "tout"), on conserve cependant un résultat de normalisabilité pour les types qui ont une certaine forme appelée "à ∀-positifs".

Définition 42 *L'ensemble des types à* ∀***-positifs****, noté* F^+*, et l'ensemble des types à* ∀***-négatifs****, noté , sont définis par induction mutuelle par :*

$$\begin{cases} F^+ & = & X \mid F^- \Rightarrow F^+ \mid \forall X F^+ \\ F^- & = & X \mid F^+ \Rightarrow F^- \end{cases}$$

Ces ensembles sont des sous-ensembles des types du système F (construits uniquement avec ∀ et ⇒. Soit N_0 et N deux constantes de types d'interprétation respectives N_0 et N. On définit sur tous les types du système F une transformation $.^*$ définie par :

$$\begin{cases} X^* & = & X \\ (A \Rightarrow B)^* & = & (A^* \Rightarrow B^*) \\ (\forall X A)^* & = & \forall X((N_0 \subset X \subset N) \twoheadrightarrow A^*) \end{cases}$$

en utilisant la notation $(N_0 \subset X \subset N)$ pour $((N_0 \subset X) \wedge (X \subset N))$.

Nous allons montrer deux lemmes concernant la transformation $.^*$: d'une part, elle transforme tout type du système F en un type compris entre N_0 et N, d'autre part elle transforme tout type à ∀ positifs en un type plus grand.

Lemme 66 *Pour tout type* A *du système* F *de variables libres* $X_1, \ldots, X_n$

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash (N_0 \subset A^* \subset N)$$

preuve: Par induction sur la structure de A :
– Si A est une variable, c'est une instance de Ax^o (c'est à dire une hypothèse).
– Si $A = B \Rightarrow C$: par hypothèse d'induction, on a

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash (N_0 \subset B^* \subset N)$$

et

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash (N_0 \subset C^* \subset N)$$

donc par la co- et contra-variance de ⇒ et l'hypothèse $Adequ(N, N_0)$:

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash (N_0 \subset (B^* \Rightarrow C^*) \subset N)$$

– Si $A = \forall X B$, par hypothèse d'induction, on a :

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N), (N_0 \subset X \subset N) \vdash (N_0 \subset B^* \subset N)$$

Il est immédiat de dériver

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash N_0 \subset \forall X((N_0 \subset X \subset N) \twoheadrightarrow B^*)$$

avec une succession de règles élémentaires.
Pour montrer

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash \forall X((N_0 \subset X \subset N) \twoheadrightarrow B^*) \subset N$$

on montre d'abord

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash (N_0 \subset N_0 \subset N) \twoheadrightarrow B^*[X := N_0] \subset N$$

qui est facile par hypothèse d'induction et par $Adequ(N, N_0)$, la conclusion étant ensuite immédiate.

□

Lemme 67 *Pour tout type A du système F de variables libres $X_1, \ldots, X_n$, si A est à $\forall$ positifs, alors :*

$$(N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N), \vdash A \subset A^*$$

preuve: On montre également que si A est à $\forall$ négatifs, alors :

$$Adequ(N, N_0), (N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash A^* \subset A$$

par induction mutuelle sur la structure de A :
– Si A est une variable, c'est immédiat.
– Si $A = B \Rightarrow C$: on a trivialement le résultat selon que $A \in F^+$ ou que $A \in F^-$.
– SI $A = \forall X B$, on a par hypothèse d'induction :

$$(N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N), (N_0 \subset X \subset N), \vdash B \subset B^*$$

donc

$$(N_0 \subset X_1 \subset N), \ldots, (N_0 \subset X_n \subset N) \vdash B \subset (N_0 \subset X \subset N) \twoheadrightarrow B^*$$

et on conclut facilement.

□

Avec ces deux lemmes, on en déduit le théorème suivant :

Théorème 68 *Pour tout type A à $\forall$ positifs clos du système F, on a :*

$$\vdash A \subset \mathcal{N}$$

preuve: A l'aide des deux lemmes précédents, on en déduit que si A est à $\forall$ positifs, alors :

$$Adequ(N, N_0) \vdash A^* \subset N \quad \text{et} \quad \vdash A \subset A^*$$

On en déduit facilement $N(A)$, et donc par définition de $\mathcal{N}$, et en utilisant l'axiome d'atomicité, $A \subset \mathcal{N}$.

□

L'exemple suivant montre comment "fonctionne" l'algorithme sous-jacent à la preuve que nous avons faite.

Exemple 2 *On prend le type $Nat = \forall X(X \Rightarrow (X \Rightarrow X) \Rightarrow X)$ des entiers de Church.*

$$\begin{aligned} Nat &\subset N_0 \Rightarrow (N_0 \Rightarrow N_0) \Rightarrow N_0 \\ &\subset N_0 \Rightarrow (N \Rightarrow N_0) \Rightarrow N \\ &\subset N_0 \Rightarrow N_0 \Rightarrow N \\ &\subset N_0 \Rightarrow N \\ &\subset N \end{aligned}$$

Tous les types de données usuels (entiers de Church, listes, arbres...) du système F entrent dans cette catégorie de types $\forall$-positifs. Dans la pratique, on les utilisera en fait peu, mais on prendra à leur place leurs définitions récursives "à la ML", et on montrera la préservation de la normalisabilité par les différentes opérations (produit cartésien, somme, types inductifs positifs...).
Nous venons donc de gagner, par le résultat précédent, ce que nous semblions avoir perdu dans le système ST, c'est à dire la possibilité de décrire la normalisabilité. Nous voyons maintenant que nous gagnons également la possibilité de décrire la résolubilité.

5.5 Résolubilité

La notion de résolubilité est une des notions centrales du λ-calcul ; elle permet de cerner deux concepts :

- Le premier, qui peut sembler artificiel à première vue, et d'où provient, à notre sens, le terme assez abscons de "résolubilité" : on dit qu'un terme t est résoluble s'il existe une solution $\overrightarrow{a}$ à l'équation $t\ \overrightarrow{a} \succ_\beta \texttt{fun}\ x\ \texttt{->}\ x$, autrement dit si cette équation est résoluble (la notation $\overrightarrow{a}$ représentant une suite de terme $a_1, \ldots, a_n$ et $t\ \overrightarrow{a}$ représentant la succession de n applications $\texttt{(}\ \ldots \texttt{(}\ t\ a_1\texttt{)}\ \ldots a_n\texttt{)}$).
- Le second, qui est équivalent au premier (cf. par exemple [3], [32], [7] ...) est moins abstrait : un terme t est résoluble s'il est normalisable de tête, autrement dit si sa réduction de tête $\succ_h$ termine :

  $$t \succ_h \quad \ldots \quad \succ_h \texttt{fun}\ x_1\ \texttt{->}\ \ldots \texttt{fun}\ x_n\ \texttt{->}\ \texttt{(}\ \ldots \texttt{(}x_i\ t_1\texttt{)}\ \ldots t_m\texttt{)}$$

  $(1 \leq i \leq n)$.

L'importance de cette notion tient d'une part au fait qu'on est certain qu'une forme de calcul (d'évaluation) s'arrêtera sur un terme résoluble, et d'autre part au fait qu'un terme résoluble est un terme dont on pourra "tirer de l'information" : si on l'applique à une suite de termes, on est certain qu'un de ces termes sera mis en action car il sera substitué à la variable de tête x_i.
L'exemple typique de terme non résoluble est le (fameux) terme $\delta\ \delta$, avec $\delta = \texttt{fun}\ x\ \texttt{->}\ (x\ x)$, qui se reproduit à l'identique par réduction sans jamais s'autoriser à avaler un autre terme. Le lecteur intéressé par les questions autour de la résolubilité pourra consulter la thèse de Y. Bertini ([8]).
Loin de nous la prétention de nous attaquer à des notions fines autour de ce concept. Nous montrons ici simplement que, comme sa grande soeur la normalisabilité, la résolubilité est exprimable dans le système ST. Nous commençons par définir la notion d'η-résolubilité, plus proche de notre sémantique, et montrons que cette notion est captée par la syntaxe, à l'aide d'un type inductif. Puis nous montrons que cette notion est équivalente avec la résolubilité usuelle.

5.5.1 η-résolubilité

Nous introduisons la notion suivante, qui correspond à la notion usuelle avec la $\beta\eta$ équivalence au lieu de la β-réduction :

Définition 43 *On dit qu'un terme t est η- **résoluble** s'il existe une suite (éventuellement vide) de λ-termes $a_1, \ldots, a_n$ telle que :*

$$(\ t\ a_1, \ldots, a_n\ ) =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$$

$(\ t\ a_1, \ldots, a_n\ )$ représente $(\ \ldots (t\ a_1) \ldots a_n)$.
On note $\mathcal{R}$ l'ensemble des termes η-résolubles.

Il est immédiat que l'ensemble que $\mathcal{R}$ est saturé par $\beta\eta$, et est donc un élément de $\overline{\tau}$.

On peut le définir d'une autre façon. Soit la F fonction de $\overline{\tau}$ dans $\overline{\tau}$ qui à tout type A associe l'ensemble des termes t tels qu'il existe un terme u tel que $(t\ u) \in A$ (cet ensemble est clairement aussi un élément de $\overline{\tau}$). Nous constatons aisément que $F(\mathcal{R}) \subset \mathcal{R}$: effectivement, si $t \in F(\mathcal{R})$, alors il existe u tel que $(t\ u) \in \mathcal{R}$. Par suite, il existe une suite (éventuellement vide) de λ-termes $a_1, \ldots, a_n$ telle que : $(\ (t\ u)\ a_1, \ldots, a_n\ ) =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$, et donc t est η-résoluble (on prend la suite $u, a_1, \ldots, a_n$). Prenons maintenant F' définie par $F'(A) = \{\texttt{fun}\ x\ \texttt{->}\ x\ \}^{\beta\eta} \cup F(A)$. On a encore $F'(\mathcal{R}) \subset \mathcal{R}$, et on a de plus $\mathcal{R} \subset F'(\mathcal{R})$. Effectivement, si $t \in \mathcal{R}$:
- Soit $t =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$, et donc $t \in F'(\mathcal{R})$.
- Sinon, il existe une suite non vide $a_1, \ldots, a_n$ telle que $(\ t\ a_1, \ldots, a_n\ ) =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$, et donc $(\ t\ a_1) \in \mathcal{R}$, donc $t \in F'(\mathcal{R})$.

$\mathcal{R}$ est donc un point fixe de F'.
Il s'agit même du plus petit point fixe de F' : on montre que pour tout point fixe I de F', pour tout $t \in \mathcal{R}$, $t \in I$ par récurrence sur la plus petite longueur de suite $a_1, \ldots, a_n$ telle que $(\ t\ a_1, \ldots, a_n\ ) =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$:
- Si elle vaut 0, $t =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$, et donc $t \in I$.
- Si elle vaut $m+1$, il existe une suite $a_1, \ldots, a_m, a_{m+1}$ telle que $(\ t\ a_1, \ldots, a_m, a_{m+1}\ ) =_{\beta\eta} \texttt{fun}\ x\ \texttt{->}\ x$. Par minimalité, la plus petite longueur de suite correspondant au terme $(t\ a_1)$ est m et donc $(t\ a_1) \in I$, et donc $t \in F'(I)$.

Nous avons donc montré la proposition suivante :

Proposition 69 *L'ensemble des termes η-résolubles est le plus petit point fixe de la fonction :*

$$F' : A \mapsto \{\texttt{fun}\ x\ \texttt{->}\ x\}^{\beta\eta} \cup F(A)$$

La fonction F étant définie par :

$$F : A \mapsto \{t; \exists u : (t\ u) \in A\}$$

Par une heureuse coïncidence, la fonction F' est définissable dans le système ST, et le plus petit point fixe également (cf. le chapitre 3).

Fait 70 *On a dans le modèle standard :*

$$\mathcal{I}(\lambda A(\Lambda X X \quad \cup \quad (\cup X((X \Rightarrow A) \restriction (X \neq \emptyset))))) = F'$$

preuve: Il suffit de vérifier que $\mathcal{I}(\lambda A(\cup X(X \Rightarrow A) \restriction (X \neq \emptyset)) = F$, ce qui est immédiat par la sémantique. □
Munis de ces résultats, nous sommes en mesure de prouver le théorème suivant :

Théorème 71 *Il existe un type Res tel que pour tout terme clos t, pour tout type A :*

$$Si \vdash A \subset Res \text{ et } \vdash t : A, \text{ alors } t \text{ est } \eta\text{-résoluble.}$$

preuve: D'après ce qu'on a vu dans le chapitre 3, le plus petit point fixe d'un opérateur croissant est définissable dans ST à l'aide de l'opérateur μ. On a donc, à l'aide des faits précédents, si on pose

$$Res = \mu A(\Lambda X X \cup (\cup X(X \Rightarrow A) \restriction (X \neq \emptyset)))$$

alors $\mathcal{I}(Res) = \mathcal{R}$, ce qui donne avec le lemme d'adéquation le résultat. □

5.5.2 Normalisabilité de tête

Nous allons montrer ici un autre théorème, qui est une conséquence assez immédiate du précédent : il est possible de caractériser dans le système ST la normalisabilité de tête :

Théorème 72 *Il existe un type Res tel que, pour tout terme t :*

$$\vdash t : Res \text{ si et seulement si } t \text{ est normalisable de tête}$$

La preuve de ce théorème repose sur la proposition suivante :

Proposition 73 *Pour tout terme t :*

$$t \text{ est normalisable de tête si et seulement si } t \text{ est } \eta\text{-résoluble}$$

preuve: On a vu dans le début de la section qu'un terme était résoluble si et seulement si il était normalisable de tête. Il suffit donc de montrer qu'un terme est résoluble si et seulement si il est η-résoluble, ce qui est prouvé par exemple dans [3], proposition 15.1.7. □

On donne maintenant la preuve du théorème :

preuve: On reprend le type Res défini dans la preuve du théorème 71 page 84.

- Seulement si : est clair par le théorème 71, et par la proposition 73, et le lemme d'adéquation.
- Si : Soit t' en forme normale de tête tel que $t \succ_\beta t'$. Par le théorème 40 page 62 et le théorème du type principal 44 page 65, il suffit de montrer que $\vdash |t'| \subset Res$.
  Posons

$$t' = \texttt{fun } x_1 \texttt{ -> } \ldots x_n \texttt{ (} x_i \ u_1(\overline{x}) \ \ldots u_m(\overline{x}) \texttt{)} \text{ et}$$

$$\pi = \texttt{fun } y_1 \texttt{ -> } \ldots y_m \texttt{ fun } x \texttt{ -> x}.$$

  En prenant $U_i = |u_i(\pi)|$, et $\Pi = |\pi|$, on a clairement : $\vdash |t'| \subset (\Pi \Rightarrow ... \Rightarrow \Pi \Rightarrow \Pi[U_1]...[U_m]) \subset (\Pi \Rightarrow ... \Rightarrow \Pi \Rightarrow I)$. On a de plus que $\vdash \Pi \neq \emptyset$. Maintenant, par le fait 39 page 59 :

  1. $\vdash \Lambda X X \subset Res$
  2. $\vdash \forall X (X \neq \emptyset \Rightarrow (X \Rightarrow Res \subset Res)$

  En itérant n fois le point 2. avec $X = \Pi$, on obtient : $\vdash |t'| \subset Res$.

□

5.6 Conclusion

On a vu dans ce chapitre comment exprimer dans ST de nombreuses propriétés d'une part théoriques (décidabilité, logique classique) et d'autre part pratiques (différentes formes de terminaison, par normalisation et par normalisation de tête). Nous allons maintenant

présenter une extension de ST à vision plus pragmatique, et regarder comment on peut envisager un codage des types de données usuels, dans le but d'obtenir un langage de programmation et un système de typage les plus proches possibles de ce qu'on utilise dans ML.

Chapitre 6

Types de données dans le système ST

Nous voyons ici comment "encoder" dans le λ-calcul les types de données de base, et comment leur donner un type adéquat. En premier lieu nous nous intéressons à l'archétype des types de données qui est celui des entiers. Nous constaterons (théorème 80 page 93) à cette occasion que le système de typage "pur" est suffisant pour exprimer les propriétés de l'arithmétique du second ordre (et même au delà). Puis nous voyons comment encoder les types usuels (produit cartésien, sommes, enregistrements et types abstraits), ainsi qu'une forme de schéma de compréhension dans le système ST[1], étendu par deux axiomes pour ce qui concerne les types abstraits et le schéma de compréhension.

6.1 Types de données stricts et entiers (de Church)

Depuis les travaux de Church sur les liens entre fonctions récursives et λ-calcul, on dispose d'une façon simple d'encoder les entiers à qui l'on a même donné le patronyme de leur inventeur. L'idée est de représenter un entier n comme un "itérateur" $\tilde{n}$ prenant en argument un terme z jouant le rôle de zéro, une fonction s jouant le rôle du successeur, et fabriquant le résultat de la fonction s itérée n fois sur le terme z :

$$\tilde{n} = \texttt{fun } z \texttt{ -> fun } s \texttt{ -> (}\underbrace{s \ldots s}_{n}\ z\texttt{)}$$

Cette classe de programme est étonnamment bien cernée par divers systèmes de types, dans lesquels on a le fait que si un terme t a un certain type, alors il se β-réduit à un entier de Church.

[1] La totalité des résultats énoncés dans cette section sont prouvés dans le logiciel PhoX

En particulier :
– Dans le système des types simples : si

$$\vdash t : o \to (o \to o) \to o$$

alors t se β-réduit à un entier de Church.
– Dans le système F de Girard [24] : si

$$\vdash t : \forall X(X \to (X \to X) \to X)$$

alors t se β-réduit à un entier de Church.
– Dans le système AF_2 de Krivine [32] : si

$$\vdash t : \forall X((X0) \to \forall y((Xy) \to (X(s(y)))) \to (X(s^n(0))))$$

alors t se β-réduit à l'entier de Church $\tilde{n}$.

Les deux premiers systèmes sont assez pauvres au niveau expressivité logique. Le troisième système fait apparaître le "sens" du type, qui est celui du principe de récurrence ; dans ce système, on se donne un ensemble de termes du premier ordre qui contient la constante 0, la fonction S, et dans lequel on peut ajouter d'autres fonctions, avec un ensemble d'équations logiques qui les caractérisent, une preuve de totalité d'une fonction devenant un terme qui calcule cette fonction.
Nous allons voir que le système $ST_{\neg TRIV}$ est en fait suffisant en lui-même pour obtenir bien plus que les autres systèmes : il permet naturellement comme ses cousins de cerner les entiers de Church, et de plus avec une définition de type de donnée "interne", mais il permet surtout de prouver les axiomes de Peano.

6.1.1 Types de données stricts

La définition d'un type de données strict correspond à ce qu'on attend d'un (prédicat de) type suffisamment précis pour décrire ses éléments :

Définition 44 *On dit qu'un terme $P : \tau \to \tau$ est un **type de données strict** si :*

$$\vdash \forall x(P(x) \neq \emptyset \quad \Rightarrow \quad \mathcal{S}(x) \wedge (x =_{\subset} P(x)))$$

Le sens de cette définition est que si $P(t)$ est non vide, alors t est un singleton et t est égal à $P(t)$; ceci qui correspond à la définition usuelle de AF_2 pour, par exemple, les entiers : si $t : N(s^n(0))$, alors $t =_\beta \overline{n}$, $\overline{n}$ étant l'entier de Church associé à n.
Le fait suivant établit le lien entre typage d'un programme d'un type de données strict et forme de ce programme :

Fait 74 *Pour tout programme t, si P est un type de donnée strict et X est un type, la règle suivante est dérivable :*

$$\frac{\vdash t : P(X)}{\vdash t : X}$$

preuve: Immédiat avec la définition de type de données strict et le fait 29.6. □
Nous allons maintenant voir comment cette notion abstraite se concrétise à travers le prototype des types de données stricts que sont les entiers de Church.

6.1.2 Zéro, successeur et prédicat "être entier"

Nous définissons les types correspondants aux programmes "zéro" et "successeur" usuels, puis les types et formules caractérisant le fait d'être entier :

Définition 45 – *Le type "Zéro" (0) est défini par :*

$$0 = \Lambda X \Lambda F X$$

– *Le type "Successeur" (S) est défini par :*

$$S = \Lambda N \Lambda X \Lambda F F[N[X][F]]$$

– *Le prédicat "être entier" (N^o) est défini par :*

$$N^o(n) = \forall^o_{\tau\to o} X(X(0) \Rightarrow \forall y(X(y) \Rightarrow X(S[y])) \Rightarrow X(n))$$

– *Le type paramétré "être de type entier" (N^τ) est défini par :*

$$N^\tau(n) = \forall^\tau_{\tau\to\tau} X(X(0) \Rightarrow \forall y(X(y) \Rightarrow X(S[y])) \Rightarrow X(n))$$

Le premier résultat que nous montrons est que le type paramétré N^τ est un type de données strict :

Théorème 75 *N^τ est un type de données strict.*

D'après le lemme précédent, cela entraînera que si $\vdash t : N^\tau(S^n[0])$, alors $\vdash t : S^n[0]$, soit que l'on obtient bien un entier de Church.
La preuve repose sur quelques lemmes techniques.
Le premier exprime le lien entre typage et possibilité de faire un raisonnement par récurrence purement logique :

Lemme 76 *Être de type entier entraîne être entier :*

$$\vdash \forall n(N^\tau(n) \neq \emptyset \Rightarrow N^o(n))$$

preuve: Avec le fait 29 page 53, il suffit de montrer :

$$x : N^\tau(n) \vdash N^o(n)$$

Et, avec la règle RC, il suffit donc de trouver un programme p tel que :

$$x : N^\tau(n) \vdash p : (N^o(n) \twoheadrightarrow \perp^\tau) \Rightarrow \perp^\tau$$

En posant $A = \lambda t.(N^o(t) \twoheadrightarrow \perp^\tau) \Rightarrow \perp^\tau$, on a :

$$x : N^\tau(n) \vdash x : A(0) \Rightarrow \forall y(A(y) \Rightarrow A(S[y])) \Rightarrow A(n)$$

Il reste donc à trouver un habitant pour $A(0)$, et un autre pour $\forall y(A(y) \Rightarrow A(S[y])$:
– On a évidemment $\vdash N^o(0)$, ce qui entraîne $\vdash$ `fun` x `->` $x : A(0)$.
– On a évidemment également $\vdash \forall y(N^o(y) \Rightarrow N^o(S[y]))$, ce qui entraîne :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{u : A(y), z : N^o(S[y]) \twoheadrightarrow \bot^\tau \vdash N^o(y)}\, Ax, RC}{u : A(y), z : N^o(S[y]) \twoheadrightarrow \bot^\tau \vdash N^o(S[y])}\, \Rightarrow^o_E}{u : A(y), z : N^o(S[y]) \twoheadrightarrow \bot^\tau \vdash \text{z: } \bot^\tau}\, Ax, \twoheadrightarrow_E}{u : A(y) \vdash \texttt{fun } z \texttt{ -> } z : A(S[y]))}\, \Rightarrow^\tau_I}{\vdash \texttt{fun } u \texttt{ -> fun } z \texttt{ -> } z : \forall y(A(y) \Rightarrow A(S[y]))}\, \Rightarrow^\tau_I, \forall^\tau_I$$

On obtient donc bien le séquent souhaité, avec

$$p = \texttt{((}x \texttt{ fun } x \texttt{ -> } x\texttt{) fun } u \texttt{ -> fun } z \texttt{ -> } z\texttt{)}$$

□

Le deuxième lemme sert à montrer la singletonicité :

Lemme 77 *Tout entier est un singleton :*

$$\vdash \forall n(N^o(n) \Rightarrow \mathcal{S}(n))$$

preuve: C'est une simple preuve par récurrence :
– $\mathcal{S}(0)$ est évident par le théorème de singletonicité (théorème 47 page 69).
– $\forall y(\mathcal{S}(y) \Rightarrow \mathcal{S}(S[y]))$ est immédiat par l'axiome de conservation des singletons.

□

Le troisième lemme est un lemme sur le "comportement calculatoire" des types entiers :

Lemme 78 *On montre :*

$$\vdash \forall n(N^\tau(n) \subset \Lambda X \Lambda F[[nX]F])$$

preuve: On passe par deux types intermédiaires :
– On définit le type "mixte" :

$$N^{\tau,o}(n) = \forall X(X(0) \Rightarrow \forall y(N^o(y) \twoheadrightarrow X(y) \Rightarrow X(S[y])) \Rightarrow X(n))$$

et on montre :

$$\vdash \quad N^\tau(n) \subset N^{\tau,o}(n)$$

en montrant que :

$$\vdash \quad N^\tau(n) \quad \subset \quad A(0) \Rightarrow \forall y(A(y) \Rightarrow A(S[y])) \Rightarrow A(n)$$

avec $A(t) = X(t) \restriction N^o(t)$.

– On définit le type lambda-terme agrémenté de singletons :

$$T(n) = \forall X, F((\mathcal{S}(X) \wedge \mathcal{S}(F)) \twoheadrightarrow X \Rightarrow F \Rightarrow n[X][F])$$

et on montre :

$$\vdash \forall n(T(n) \subset \Lambda X \Lambda F n[X][F])$$

en utilisant l'axiome d'atomicité et les propriétés de l'application de types : il suffit en effet de montrer $\vdash T(n)[X][F] \subset n[X][F]$. On met en hypothèses $x \subset X, \mathcal{S}(x), f \subset F, \mathcal{S}(f)$, et on montre facilement sous ces hypothèses que $T(n)[x][f] \subset n[x][f]$, et donc $T(n)[x][f] \subset n[X][F]$. La conclusion vient facilement avec l'axiome d'atomicité.

Il reste donc à montrer que :

$$\vdash N^{\tau,o}(n) \subset T(n)$$

Cela se fait en prouvant :

$$\mathcal{S}(X), \mathcal{S}(F) \vdash \begin{array}{c}(0[X][F] \Rightarrow \forall y(N^o(y) \twoheadrightarrow y[X][F] \Rightarrow S[y][X][F]) \Rightarrow n[X][F]) \\ \subset \\ (X \Rightarrow F \Rightarrow n[X][F])\end{array}$$

Avec les règles de co et contra-variance de $\Rightarrow$, il suffit de montrer deux sous-typages (le troisième étant trivial) :

$$\mathcal{S}(X), \mathcal{S}(F) \vdash X \subset 0[X][F]$$
et
$$\mathcal{S}(X), \mathcal{S}(F) \vdash F \subset \forall y(N^o(y) \twoheadrightarrow y[X][F] \Rightarrow S[y][X][F])$$

Pour le premier séquent, on obtient facilement l'inclusion inverse, avec le théorème 40 page 62, et en utilisant l'équivalence entre $\mathcal{S}$ et $\mathcal{S}_\subset$ (proposition 48 page 69), on obtient le séquent voulu.

Pour le second, toujours avec le théorème 40 page 62, on obtient : $S[y][X][F] \subset F[y[X][F]]$, en utilisant l'équivalence entre $\mathcal{S}$ et $\mathcal{S}_\subset$ et en mettent en hypothèse $\mathcal{S}(y)$, on obtient l'inclusion inverse et on conclut en utilisant le lemme précédent et les propriétés de l'application. □

Le quatrième lemme est un lemme de sous-typage pur qui exprime que le type paramétré "être entier" appliqué à n'importe quel type est un sous-type de ce dernier[2].

Lemme 79 *On montre :*

$$\vdash \forall n((N^\tau(n) \subset n))$$

preuve: On fait deux étapes :

– On établit d'abord le comportement calculatoire du successeur :

$$\vdash \forall y(\mathcal{S}(y) \Rightarrow S[y] \subset \Lambda X \Lambda F S[y][X][F]$$

Comme il s'agit d'une β-équivalence, celle-ci se traite avec les outils du chapitre 4.

[2]il peut paraître paradoxal, mais il suffit de remarquer que si A n'est pas un entier, alors le type "A est un entier" est vide...

– On utilise le prédicat renforcé usuel sur les entiers :

$$N^{o+}(n) = \forall X(X(0) \Rightarrow \forall y(N^o(y) \Rightarrow X(y) \Rightarrow X(S[y])) \Rightarrow X(n))$$

Et on montre le fait usuel :

$$\vdash \forall n(N^o(n) \Rightarrow N^{o+}(n))$$

– On montre par récurrence le sous-typage :

$$\forall n(N^o(n) \Rightarrow (N^\tau(n) \subset n))$$

Cela se fait en utilisant la récurrence renforcée, la relation de β-équivalence ci-dessus, et les deux lemmes précédents (comportement calculatoire et singletonicité).

Enfin, en utilisant ce qu'on vient de montrer, le résultat est assez aisé à prouver en faisant une disjonction sur la vacuité de $N^\tau(n)$ et en utilisant le lemme qui fait le lien entre typage et récurrence. □

Ces lemmes étant établis, on peut prouver le théorème 75 :

preuve: On suppose $N^\tau(n) \neq \emptyset$. Avec les lemmes 76 et 77, on en déduit que $\mathcal{S}(n)$. Avec l'équivalence entre $\mathcal{S}$ et $\mathcal{S}_\subset$ (proposition 48 page 69), le lemme 79, et l'hypothèse on en déduit que $n =_\subset N^\tau(n)$. □

6.1.3 Arithmétique de Peano

On montre ici qu'on peut formuler et prouver les axiomes de Peano d'une façon très simple à l'intérieur du système de types.

Définition 46 *On appelle axiomes de Peano les formules suivantes :*

1. *Zéro est distinct de successeur :*

$$\forall n \neg(0 =_\subset S[n])$$

2. *Le successeur est injectif sur les entiers :*

$$\forall n, m(N(n) \Rightarrow N(m) \Rightarrow (S[n] =_\subset S[m]) \Rightarrow n =_\subset m)$$

3. *Tout entier non nul est le successeur d'un entier :*

$$\forall n(N(n) \Rightarrow \neg(n =_\subset 0) \Rightarrow \exists m(N(m) \wedge n =_\subset S[m])$$

4. *Propriétés de l'addition :*

   (a)

   $$\forall n((n + 0) =_\subset n)$$

   (b)

   $$\forall n, m((n + S[m]) =_\subset S[n + m])$$

5. *Propriétés de la multiplication :*

   (a)
   $$\forall n((n \times 0) =_{\subset} 0)$$

   (b)
   $$\forall n, m((n \times S[m]) =_{\subset} n + (n \times m))$$

en prenant par exemple : $m + n = m[n][S]$ et $m \times n = m[0][\Lambda x.(x + n)]$.

Théorème 80 *Les axiomes de Peano sont prouvables dans ST.*

Ce qui est surprenant dans ce résultat, c'est la capacité du système à prouver l'injectivité du successeur et le fait que zéro ne soit pas un successeur.
Nous donnons maintenant la preuve :
preuve:

1. Il suffit de remarquer que 0 et $S[n]$ étant résolubles et non β-équivalents, l'axiome de non trivialité (3 page 74) permet de conclure aisément.
2. Pour l'injectivité du successeur, on utilise simplement un prédécesseur, par exemple :
   $$Predn = n[(0,0)][\Lambda C.(Pi2[c], S[Pi2[c]])].$$
   le couple (a, b) étant défini de manière usuelle par $(a, b) = \Lambda C.C[a][b]$, et la seconde projection par $Pi2 = \Lambda C.C[\Lambda X \Lambda Y X]$.
   On montre ensuite par récurrence que :
   $$\forall n(N^o(n) \Rightarrow Pred[S[n]] \subset n)$$
   munis de cela, et des faits évidents que l'on ne manipule que des singletons, on en déduit le résultat.
3. Immédiat par récurrence.
4. Immédiat par récurrence.
5. Immédiat par récurrence.

□

6.1.4 Égalité sur les entiers

On montre ici un fait qui sera utile par la suite pour le typage des programmes, à savoir le comportement (typage) du test d'égalité.
On définit le terme `eq_nat` de la façon suivante :

```
true    =  fun x -> fun y -> x
false   =  fun x -> fun y -> y
(a,b)   =  fun c -> ( (c a) b)
pi1     =  fun c -> (c true)
pi2     =  fun c -> (c false)
eq_zero =   fun m -> ((m true ) (fun r -> false))
eq_succ =  fun r -> fun m -> (pi1
           ( (m (false,zero))
           ( fun rec -> ((r (pi2 rec)), (succ (pi2 rec))))).
eq_nat  =  fun n -> ( (n eq_zero) eq_succ )
```

Et on considère les traduits en type `Eq_nat`, `True` et `False` des programmes nommés ci-dessus.
On montre le fait suivant :

Fait 81 *Les règles suivantes sont dérivables dans ST*

$$(1) \quad \vdash \mathcal{S}(\mathit{Eq\_nat}) \wedge \mathit{Eq\_nat} \subset \forall x, y(N^{\tau}(x) \Rightarrow N^{\tau}(y) \Rightarrow ((x = y) \| (x \neq y)))$$

$$(2) \quad \vdash \forall x, y(N^{o}(x) \Rightarrow N^{o}(y) \Rightarrow \begin{pmatrix} (x = y) \Rightarrow \mathit{Eq\_nat}[x][y] \subset \mathit{True} \\ \wedge \\ (x \neq y) \Rightarrow \mathit{Eq\_nat}[x][y] \subset \mathit{False} \end{pmatrix})$$

preuve:
- La première partie est évidente car il s'agit du traduit d'un terme. La seconde partie se prouve par récurrence.
- En utilisant les propriétés de l'application (28 page 52) on déduit de (1) que

$$\texttt{Eq\_nat}[x][y] \subset ((x = y) \| (x \neq y)))$$

  On conclut facilement en utilisant les propriétés du décideur (37 page 58).

□

Notation 10 *Nous conserverons dans la suite la convention d'écriture suivante : le traduit en type de tout programme défini par un nom (`true`,`pi1`,...) sera écrite avec le même nom commençant par une majuscule (`True`,`Pi1`,...), à l'exception des paires qui seront écrites à l'identique.*

6.2 Produits et Paires

Les paires constituent la deuxième brique de base pour construire nos types de données. Nous définissons le type produit d'une façon légèrement inhabituelle : effectivement, le type habituel $A\overline{\times}B = \forall X((A \Rightarrow B \Rightarrow X) \Rightarrow X)$ est victime d'une pathologie illustrée par l'exemple ci-dessous. Posons :
– $a_x =$ `fun i ->(((i false)`x`)i)`, on a $a_x : I \Rightarrow I$ (avec $I = \Lambda XX$).
– $p =$ `fun` c `->((`c a_c`)`a_c `)`, on a $p : (I \Rightarrow I)\overline{\times}(I \Rightarrow I)$.
Mais on a par contre $p \neq_{\beta\eta}$ `((pi1` p`) ,(pi2` p`))`.
Ce qui fait que, dans le cas général :

$$A\overline{\times}B \not\subset \cup_{\alpha\in A,\beta\in B}(\Lambda C.C[\alpha][\beta])$$

le type de droite étant donc dans le cas général strictement plus petit que celui de gauche. C'est donc à cause de cette pathologie qu'on choisit le codage suivant :

Définition 47 (Produit) *On pose :*

$$A \times B = \cup_{\alpha\in A,\beta\in B}\texttt{(}\alpha\texttt{ , }\beta\texttt{)}$$

On a introduit les notations suivantes, que nous précisons :

Notation 11

$$\begin{array}{rcl} x \in A & = & (x \subset A) \wedge \mathcal{S}(x) \\ \cup_{\alpha_1\in A_1,\ldots,\alpha_n\in A_n} T(\alpha_1,\ldots,\alpha_n) & = & \cup\alpha_1 \ldots \cup \alpha_n (T(\alpha_1,\ldots,\alpha_n) \restriction (\wedge_{1\leq i\leq n}\alpha_i \in A_i) \end{array}$$

Comme nous allons beaucoup utiliser les singletons par la suite, nous introduisons également les notation suivantes :

Notation 12

$$\forall x : A.P(x) = \forall x(x \in A \Rightarrow P(x))$$
$$\exists x : A.P(x) = \exists x(x \in A \wedge P(x))$$

6.2.1 Typage

Le fait suivant donne les principales propriétés du produit, et servira à établir la preuve de correction :

Fait 82 (Propriétés du produit) *On prouve :*

$$\begin{array}{ll} (1) & \vdash \forall A, B \forall \alpha : A.\forall \beta : B((\alpha,\beta) \in A \times B) \\ (2) & \vdash \forall A, B \forall X : A \times B.(\texttt{Pi1}[X] \in A) \\ (3) & \vdash \forall A, B \forall X : A \times B.(\texttt{Pi2}[X] \in B) \\ (4) & \vdash \forall A, B \forall X : A \times B(X = (\texttt{Pi1}[X], \texttt{Pi2}[X])) \end{array}$$

avec `Pi1`, `Pi2` *et* `(X,Y)` *définis à l'aide de la convention 10.*

preuve:
- (1) : On met en hypothèses $\mathcal{S}(\alpha)$, $\alpha \subset A$, $\mathcal{S}(\beta)$, et $\beta \subset B$. La première composante de la conjonction est évidente par définition de la réunion. La singletonicité est immédiate avec les outils du chapitre 4.
- (2) : On met en hypothèse $X \subset A \times B$ et $\mathcal{S}(X)$. Par définition de $\mathcal{S}$, on en déduit $\exists \alpha \in A, \beta \in B(X \subset (\alpha, \beta))$. Sous les hypothèses $\mathcal{S}(\alpha)$, $\alpha \subset A$, $\mathcal{S}(\beta)$, $\beta \subset B$ et $X \subset (\alpha, \beta)$, on déduit $\texttt{Pi1}[X] \subset \texttt{Pi1}[(\alpha,\beta)]$, et par le fait 40, par la croissance de la paire et par hypothèse on en déduit que $\texttt{Pi1}[X] \subset A$. La preuve que $\mathcal{S}(\texttt{Pi1}[X])$ est immédiate.
- (3) : analogue à (2).
- (4) : comme dans la preuve de (2), on a sous les hypothèses $X \subset A \times B$ et $\mathcal{S}(X)$, $\exists \alpha \in A, \beta \in B(X \subset (\alpha,\beta))$. Sous les hypothèses $\mathcal{S}(\alpha)$, $\alpha \subset A$, $\mathcal{S}(\beta)$, $\beta \subset B$ et $X \subset (\alpha, \beta)$, on déduit $\mathcal{S}((\alpha, \beta))$ par (1). Par la propriété 48 page 69, on a $\mathcal{S}_{\subset}$, et on en déduit que $X = (\alpha, \beta)$. Il suffit donc de montrer $(\alpha, \beta) = (\texttt{Pi1}[(\alpha, \beta)], \texttt{Pi2}[(\alpha, \beta)])$. En utilisant les mêmes techniques que pour la preuve de (1), on montre que $\texttt{Pi1}(\alpha, \beta)] = \alpha$ et que $\texttt{Pi2}[(\alpha, \beta)] = \beta$, ce qui permet de conclure.

□

Remarque: Le "surjective pairing" n'est pas une règle de réduction, mais une règle de typage conditionnée par une prémisse qui est suffisamment forte pour assurer que les habitants du type ont la bonne forme.

6.2.2 Sous-typage

Le sous-typage du produit est un sous-typage par composante :

Proposition 83 *La règle de sous-typage suivante est dérivable :*

$$\vdash \forall A, A', B, B'(A \subset A' \Rightarrow B \subset B' \Rightarrow (A \times B) \subset (A' \times B'))$$

preuve: Sous les hypothèses $A \subset A'$ et $B \subset B'$, il suffit de montrer, en utilisant l'atomicité :

$$\forall \alpha : A \forall \beta : B(\alpha \in A' \wedge \beta \in B')$$

ce qui est immédiat. □

Ceci termine l'étude des propriétés du type produit, qui sera la base des autres types que l'on va considérer maintenant.

6.3 Sommes et cases

Les types sommes seront simplement des réunions de produits de la forme $C \times A$ avec C un entier, et donc leurs habitants, des paires : les règles d'introduction sont immédiates, on ne s'intéressera donc qu'aux règles d'élimination.

Définition 48 (Motifs et Analyse par cas) *On définit les motifs de base par :*
– `C,x -> t(x) [k,u]` = `(` $P_1(u)$ $P_2(u)$ `)` k `)` *avec :*

$$P_1(u) = \texttt{((eq\_nat (pi1 C))}\ u\texttt{)}$$
$$P_2(u) = \texttt{(((fun y -> (t (y))) (pi2 u))}\ u\texttt{)}$$

– `C -> t [k,u]` = `C,x -> t[k,u]`
– `_ -> t[k,u]` = `t`
On définit les motifs par :
– *Un motif de base est un motif*
– *Si* $M[k,u]$ *est un motif, alors*

$$\texttt{C,x -> t(x) |}\ M[k,u] = \texttt{C,x -> t(x)}[M[k,u],u]$$

est un motif.
– *Si* M *est un motif, et* u *un programme, on définit l'analyse par cas de* M *par :*

$$\texttt{cases}\ u\ \texttt{of}\ M = M[\texttt{fun x -> x}, u].$$

Le rôle d'un motif est simplement de tester si la première projection de u est égale à C. Si c'est le cas, il renvoie l'application de la fonction t à la seconde composante de D, sinon, il renvoie k.
Par exemple, si on considère deux motifs M_1 et M_2 définissant chacun deux programmes P_1^1, P_2^1, et P_2^1, P_2^2, on a :

$$\begin{aligned}
\texttt{cases}\ u\ \texttt{of}\ M_1\ \texttt{|}\ M_2 &= M_1|M_2\ \ [\texttt{fun x -> x}, u].\\
&= M_1[M_2[\texttt{fun x -> x}, u], u]\\
&= M_1[\texttt{(}\ \texttt{(}\ P_1^2(u)\ P_2^2(u)\ \texttt{)}\ \texttt{fun x -> x}\ \texttt{)}, u]\\
&= \texttt{(}\ P_1^1(u)\ P_2^1(u)\ \texttt{)}\ \texttt{(}\ \texttt{(}\ P_1^2(u)\ P_2^2(u)\ \texttt{)}\ \texttt{fun x -> x}\ \texttt{)}
\end{aligned}$$

Notre objectif étant de ne travailler que sur les types, on utilisera dans la suite les programmes comme leurs traduits en type, en suivant la convention de la notation 10 : c'est à dire qu'on notera `Cases ... Of ...` pour le traduit de `cases ... of ...` .

Proposition 84 *Les règles suivantes sont dérivables.*

1. $$\frac{\Gamma \vdash u \in A \qquad \Gamma \vdash \forall x \in A.t(x) \in B(x)}{\Gamma \vdash \texttt{Cases}\ u\ \texttt{Of}\ \_\ \texttt{->}\ t(u) \in B(u)}$$

2. $$\frac{\begin{array}{l}\Gamma \vdash u \in C \times A \\ \Gamma \vdash N^o(C) \\ \Gamma \vdash \forall y : A.t(y) \in B((C, y)) \\ \Gamma \vdash \mathcal{S}(\texttt{Fun}\ x\ \texttt{->}\ t(x))\end{array}}{\Gamma \vdash \texttt{Cases}\ u\ \texttt{Of}\ C\texttt{,}\ x\ \texttt{->}\ t(x) \in B(u)}$$

3. $$\frac{\Gamma \vdash u \in C \times \Lambda XX \qquad \Gamma \vdash N^o(C) \qquad \Gamma \vdash t \in B((C, \Lambda XX))}{\Gamma \vdash \texttt{Cases}\ u\ \texttt{Of}\ C\ \texttt{->}\ t \in B(u)}$$

preuve:

1. C'est immédiat car par définition :

   $$\texttt{Cases}\ u\ \texttt{Of}\ \_\ \texttt{->}\ t(u) = t(u)$$

   On remarque qu'ici il n'y a aucune condition sur le type A ; ceci dit, dès que l'on composera les motifs, on aura des conditions similaires aux autres cas.

2. Il s'agit de montrer $\texttt{((}\ P_1(u)\ P_2(u)\ \texttt{)}\ \texttt{Fun x -> x)} \in B(u)$. La première chose à vérifier est $\mathcal{S}(\texttt{((}\ P_1(u)\ P_2(u)\ \texttt{)}\ \texttt{Fun x -> x)})$, ce qui est immédiat par le théorème de singletonicité (47 page 69), avec les première et dernière prémisses.

   Ensuite, il faut vérifier $\texttt{((}\ P_1(u)\ P_2(u)\ \texttt{)}\ \texttt{Fun x -> x)} \subset B(u)$. Les propriétés du produit (fait 82) avec la première prémisse assurent que $\texttt{Pi1}\ u \in C$ et que $\texttt{Pi2}\ u \in A$. On en déduit que $\texttt{Pi1}\ u = C$ par les propriétés d'équivalence des singletons (48 page 69). Avec les propriétés de l'égalité (fait 81), et par β-réduction (théorème 40 page 62), on en déduit $\texttt{((}\ P_1(u)\ P_2(u)\ \texttt{)}\ \texttt{Fun x -> x)} \subset \texttt{((True}\ P_2(u)\ \texttt{)}\ \texttt{Fun x -> x)} \subset P_2(u) \subset B(u)$.

3. Immédiat par (2).

□

Proposition 85 *Les règles suivante sont dérivables :*

1.
$$\frac{\begin{array}{l} P_1 : \ \Gamma \vdash u \in (C \times A) \cup T \\ P_2 : \ \Gamma \vdash N^o(C) \\ P_3 : \ \Gamma \vdash \forall X : T.(\texttt{Pi1}[X] \neq C \wedge N^o(\texttt{Pi1}[X])) \\ P_4 : \ \Gamma \vdash \forall y : A.t(y) \in B((C, y)) \\ P_5 : \ \Gamma \vdash \Lambda X(tX) \neq \emptyset \\ P_6 : \ \Gamma \vdash \forall y : T.\texttt{Cases } y \texttt{ Of } M(y) \in B(y) \end{array}}{\Gamma \vdash \texttt{Cases } u \texttt{ Of } C\texttt{, x -> } t(\texttt{x}) \texttt{ | } M(u) \in B(u)}$$

2.
$$\frac{\begin{array}{l} P_1 : \ \Gamma \vdash u \in (C \times \Lambda XX) \cup T \\ P_2 : \ \Gamma \vdash N^o(C) \\ P_3 : \ \Gamma \vdash \forall X : T.(\texttt{Pi1}[X] \neq C \wedge N^o(\texttt{Pi1}[X])) \\ P_4 : \ \Gamma \vdash t \in B((C, \Lambda XX)) \\ P_5 : \ \Gamma \vdash \forall y : T.\texttt{Cases } y \texttt{ Of } M(y) \in B(y) \end{array}}{\Gamma \vdash \texttt{Cases } u \texttt{ Of } C \texttt{ -> } t \texttt{ | } M(u) \in B(u)}$$

preuve:

1. On sait que $\Gamma \vdash \mathcal{S}(u)$. Donc, d'après la propriété 48 page 69, on a $\Gamma \vdash \mathcal{S}_\cup(u)$. Par définition, on en déduit avec P_1 que $(u \subset (C \times A)) \vee (u \subset T)$. On raisonne par cas en utilisant les propriétés 82, 81, et 40 page 62 :
   - Sous l'hypothèse $u \subset (C \times A)$, on en déduit que
   $$\texttt{Cases } u \texttt{ Of } C\texttt{, x -> } t(\texttt{x}) \texttt{ | } \texttt{M}(u) = t(\texttt{(Pi1 u)})$$
   ce qui donne le résultat, avec P_4.
   - Sous l'hypothèse $u \subset T$, on en déduit que $(\texttt{Pi1}[u]) \neq C \wedge N^o(\texttt{Pi1}[u]))$. Et donc, $\texttt{Cases } u \texttt{ Of } C\texttt{, x -> } t(\texttt{x}) \texttt{ | } \texttt{M}(u) = \texttt{Cases } u \texttt{ Of M}(u)$, d'où la conclusion grâce à P_6.
2. Immédiat d'après 1.

□

6.4 Enregistrements

6.4.1 Typage

Définition 49 (Type enregistrement) *On pose, pour tous types $A_1, \ldots A_n$ et $C_1, \ldots C_n$:*

$$\{C_1 : A_1; \ldots; C_n : A_n\}$$
$$=$$
$$\forall X(((C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup (C_n \times (A_n \Rightarrow X))) \Rightarrow X)$$

Définition 50 (Enregistrements)
On pose, pour tous programmes $a_1, \ldots a_n$ et $C_1, \ldots, C_n$:

$$< C_1 = a_1; \ldots; C_n = a_n >$$
$$=$$
$$\texttt{fun x -> (cases x of } C_1\texttt{,f -> (f } a_1\texttt{) | }\ldots C_n\texttt{,f -> (f } a_n\texttt{))}$$

Et on définit l'ajout externe d'un champs pour tous programmes C, a et p :

$$[C = a] + p \;=\; \texttt{fun x -> cases x of } C\texttt{,f -> (f a) | _ -> (p x)}$$

On prouve le fait suivant sur le typage des traduits en type des enregistrements :

Proposition 86 *Les règles suivantes sont dérivables :*

(1)
$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash N^o(C)}{\Gamma \vdash < C = a > \in \{C : A\}}$$

(2)
$$\frac{\begin{array}{l}\Gamma \vdash a \in A \\ \Gamma \vdash < u > \in \{C_1 : A_1; \ldots; C_n : A_n\} \\ \Gamma \vdash N^o(C) \\ \Gamma \vdash \forall X(\forall y : ((C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup (C_n \times (A_n \Rightarrow X))).\left(\begin{array}{c}(\texttt{Pi1}[y]) \neq C \\ \wedge \\ N^o(\texttt{Pi1}[y])\end{array}\right))\end{array}}{\Gamma \vdash < C = a; u > \in \{C : A; C_1 : A_1; \ldots; C_n : A_n\}}$$

(3)
$$\frac{\begin{array}{l}\Gamma \vdash a \in A \\ \Gamma \vdash p \in \{C_1 : A_1; \ldots; C_n : A_n\} \\ \Gamma \vdash N^o(C) \\ \Gamma \vdash \forall X(\forall y : ((C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup (C_n \times (A_n \Rightarrow X))).\left(\begin{array}{c}(\texttt{Pi1}[y]) \neq C \\ \wedge \\ N^o(\texttt{Pi1}[y])\end{array}\right))\end{array}}{\Gamma \vdash [C = a] + p \in \{C : A; C_1 : A_1; \ldots; C_n : A_n\}}$$

preuve:

1. Il faut vérifier :

$$\Gamma \vdash \texttt{Fun } x \texttt{ -> Cases } x \texttt{ Of } C, f \texttt{ -> } (f\ a) \in \forall X((C \times (A \Rightarrow X)) \Rightarrow X)$$

   Il suffit donc de montrer :

$$\Gamma, X : \tau, x \in C \times (A \Rightarrow X) \vdash \texttt{Cases } x \texttt{ Of } C, f \texttt{ -> } (f\ a) \in X$$

   On utilise la règle 2. de la propriété 84. Avec notre hypothèse et les prémisses qui assurent $\mathcal{S}(a)$, $a \subset A$ et $N^o(C)$:
   – Les deux premières prémisses sont immédiates.
   – $\forall y : A \Rightarrow X).(y\ a) \in X$ est facile.
   – $\mathcal{S}(\texttt{Fun } x \texttt{ -> } (x\ a)$ également.

2. On met en hypothèse $x \in (((C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup (C_n \times (A_n \Rightarrow X)))$, et il faut en déduire

$$(\texttt{cases } x \texttt{ of } C\ ,\ f \texttt{ -> } (f\ a)\ \texttt{|}\ u(x)) \in \{C : A; C_1 : A_1; \ldots; C_n : A_n\}$$

   On utilise la règle 1. de la propriété 85, dont il faut vérifier les 6 prémisses :
   – P_1 et P_2 sont données par nos prémisses.
   – P_3 vient de notre dernière prémisse.
   – P_4 et P_5 viennent de $t = \texttt{fun } f \texttt{ -> } (f\ a)$.
   – P_6 vient de $(\texttt{fun } x \texttt{ -> cases } x \texttt{ of } u(x)) \in \{C_1 : A_1; \ldots; C_n : A_n\}$.

3. Il suffit d'utiliser les règle 1. de la propriété 85 et 3. de la propriété 84.

□

Afin de pouvoir accéder aux composantes d'un enregistrement, on définit la projection :

Définition 51 (Projection) *La projection $u.C$ est définie par :*

$$u.C = \texttt{(} u\ \texttt{ (} C\ \texttt{ , fun x -> x))}$$

On donne simplement la règle suivante, qui est suffisante grâce aux propriétés de sous-typage que nous verrons ensuite :

Proposition 87 *La règle suivante est dérivable :*

$$\frac{\Gamma \vdash u \in \{C : A\} \quad \Gamma \vdash N^o(C)}{\Gamma \vdash u.C \in A}$$

preuve: Il suffit de remarquer que $(C\ ,\ \Lambda X X) \subset C \times (A \Rightarrow A)$. □

6.4.2 Sous-typage

On désire deux propriétés de sous-typage : le sous-typage par composante et le sous-typage par omission de composante.

Proposition 88 *Les règles de sous-typage suivantes sont dérivables :*

$$(1) \quad \vdash \forall A_1, \ldots, A_n, A'_i((A_i \subset A'_i) \Rightarrow \left(\begin{array}{c} \{C_1 : A_1; \ldots; C_i : A_i; \ldots; C_n : A_n\} \\ \subset \\ \{C_1 : A_1; \ldots; C_i : A'_i; \ldots; C_n : A_n\} \end{array} \right)$$

$$(2) \quad \vdash \forall A_1, \ldots, A_n, A'_i \left(\begin{array}{c} \{C_1 : A_1; \ldots; C_{i-1} : A_{i-1}; C_i : A_i; C_{i+1} : A_{i+1}; \ldots; C_n : A_n\} \\ \subset \\ \{C_1 : A_1; \ldots; C_{i-1} : A_{i-1}; C_{i+1} : A_{i+1}; \ldots; C_n : A_n\} \end{array} \right)$$

preuve:

1. Sous l'hypothèse $A_i \subset A'_i$, on a en utilisant le sous-typage du produit et la contravariance de la flèche que :

   $$C_i \times (A'_i \Rightarrow X) \subset C_i \times (A_i \Rightarrow X)$$

   En utilisant le sous-typage de la réunion :

   $$\vdash \begin{array}{c} (C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup C_i \times (A'_i \Rightarrow X) \cup \ldots \cup C_n \times (A_n \Rightarrow X) \\ \subset \\ (C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup C_i \times (A_i \Rightarrow X) \cup \ldots \cup C_n \times (A_n \Rightarrow X) \end{array}$$

   La conclusion est immédiate.

2. On a simplement en utilisant le sous-typage de la réunion :

   $$\vdash C_i \times (A_i \Rightarrow X) \subset (C_1 \times (A_1 \Rightarrow X)) \cup \ldots \cup C_n \times (A_n \Rightarrow X)$$

   La conclusion est immédiate.

□

6.5 Types abstraits et description définie

6.5.1 ST_Δ

Tout comme le système $ST_{\neg TRIV}$ présenté au chapitre 5, le système ST_Δ est une extension de ST, à ceci près qu'il ajoute de nouveaux opérateurs.

6.5.2 Syntaxe et règles

Nous ajoutons au langage de termes du système ST une nouvelle famille de constantes : pour chaque suite de sortes $s = s_1, \ldots, s_n$, on ajoute à l'ensemble des constantes $Const\ n$ opérateurs Δ_i^s $(i = 1, \ldots, n)$ de sortes respectives $(s_1, \ldots, s_n \to o) \to s_i$.
Une écriture agréable et compacte des règles correspondant à ces opérateurs nécessite une définition pour les propositions existentielles portants sur plusieurs variables.

Définition 52 (Existentielles et Unions multiples)
A chaque suite de sortes $s = s_1, \ldots, s_n$, on associe un opérateur $\exists^s$ défini par :

$$\exists^s = \lambda P \forall K(\forall x_1 \ldots x_n(P(x_1) \ldots (x_n) \Rightarrow K) \Rightarrow K)$$

A chaque suite de sortes $s = s_1, \ldots, s_n$, on associe un opérateur $\cup^s$ défini par :

$$\cup^s = \lambda A \forall X(\forall x_1 \ldots x_n(A(x_1) \ldots (x_n) \subset X) \twoheadrightarrow X)$$

Notation 13 *On utilisera les notations $\exists x1 \ldots x_n P$ pour $\exists P$, et $\cup x_1 \ldots x_n A$ pour $\cup A$.*

On donne une famille d'axiomes :

Axiome 4 (Axiomes de description définie)
Pour chaque liste de sortes $s = s_1, \ldots, s_n$, on pose l'axiome :

$$(\Delta_s) \qquad \forall P(\exists^s P(x_1) \ldots (x_n) \Rightarrow P(\Delta_1^s(P)) \ldots (\Delta_n^s(P)))$$

Nous donnons ici quelques règles dérivables sur les existentielles qui seront utiles dans la suite de la thèse.

Fait 89 *On prouve :*

1. $\vdash \forall P \forall x_1 \ldots x_n(P(x_1) \ldots (x_n) \Rightarrow \exists x_1 \ldots x_n P)$
2. $\vdash \forall P \forall K(\exists x_1 \ldots x_n P \Rightarrow \forall x_1 \ldots x_n(P(x_1) \ldots (x_n) \Rightarrow K) \Rightarrow K)$
3. $\vdash \forall A \forall x_1 \ldots x_n(A(x_1) \ldots (x_n) \subset \cup x_1 \ldots x_n A)$
4. $\vdash \forall A \forall X(\mathcal{S}(X) \Rightarrow X \subset \cup x_1 \ldots x_n A \Rightarrow \exists x_1 \ldots x_n(X \subset A(x_1) \ldots (x_n)))$
5. *En posant $X.A.i = \Delta_i^s(\lambda x_1 \ldots x_n(X \subset A(x_1) \ldots (x_n)))$, on a :*

$$\vdash \forall X \forall A(\mathcal{S}(X) \Rightarrow X \subset \cup x_1 \ldots x_n A \Rightarrow X \subset A(X.A.1) \ldots (X.A.n))$$

preuve:
– 1,2,3 sont immédiates.
– 4 : On montre d'abord $\cup x_1 \ldots x_n A(x_1) \ldots (x_n) \subset \cup x_1 \cup x_2 \ldots \cup x_n A(x_1) \ldots (x_n)$ par récurrence sur la longueur l de la suite $s = s_1, \ldots, s_n$:
  – Si $l = 1$, c'est vrai.

- On suppose $l > 1$. Il suffit de montrer $A(x_1)\dots(x_n) \subset \cup x_1 \cup x_2 \dots \cup x_n A(x_1)\dots(x_n)$. On met en hypothèse $\forall x_1(\cup x_2 \dots \cup x_n A(x_1)\dots(x_n) \subset X)$ et il faut en déduire que $A(x_1)\dots(x_n) \subset X$. Il suffit de montrer que $A(x_1)\dots(x_n) \subset \cup x_2 \dots \cup x_n A(x_1)\dots(x_n)$, ce qui est vrai car $A(x_1)\dots(x_n) \subset \cup x_2 \dots x_n A(x_1)\dots(x_n)$ et par hypothèse de récurrence $\cup x_2 \dots x_n A(x_1)\dots(x_n) \subset \cup x_2 \dots \cup x_n A(x_1)\dots(x_n)$.

  En appliquant un nombre de fois égal à la longueur de la suite la définition de singleton de sorte s telle que vue au fait 33, on obtient $\exists x_1 \dots \exists x_n(X \subset A(x_1)\dots(x_n)$. Par suite, il suffit de montrer $\forall P(\exists x_1 \dots \exists x_n P \Rightarrow \exists x_1 \dots x_n P)$ ce qui est immédiat par 1..
- 5 : Immédiat par 4. et Δ_s.

□

On remarque que la notation introduite dans le point 5. du fait précédent s'approche d'assez près de la notation habituelle de la projection.

6.5.3 Adéquation

Il est tout de même nécessaire de montrer que l'extension que nous proposons est consistante. Il suffit pour cela de donner une interprétation aux opérateurs Δ_i^s.
Soit $s = s_1, \dots, s_n$ une suite de sortes. On applique la proposition 6 page 22 du chapitre 2 avec :
- $A_i = \overline{s_i}$
- $m = 1$
- $E = \overline{o} = \{\bot; \top\}$ (on a donc $P_E = \top$)

Soit $\mathcal{I}$ une interprétation. On l'étend en posant :

$$\mathcal{I}(\Delta_i^s) = \cdot_{(i,1)}$$

Vérifions la validité des nouveaux axiomes :
Si $\mathcal{I}(\exists x_1 \dots x_n P(x_1)\dots(x_n)) = \top$, alors :

$$\top \leq \vee\{\mathcal{I}(P)(a_1)\dots(a_n); \bigwedge_{1\leq i\leq n} a_i \in A_i\}$$

et donc

$$\top \leq \mathcal{I}(P)(\cdot_{(1,1)}(\mathcal{I}(P), \top))\dots(\cdot_{(n,1)}(\mathcal{I}(P), \top))$$

On en déduit que $\mathcal{I}(P)(\cdot_{(1,1)}(\mathcal{I}(P), \top))\dots(\cdot_{(n,1)}(\mathcal{I}(P), \top)) = \top$, qui est le résultat souhaité.
On a donc prouvé la proposition suivante :

Proposition 90 *Le système $ST_{\Delta\neg TRIV}$ est consistant.*

6.6 Antisymétrie et schéma de compréhension

On voit ici que l'on peut également définir une forme de schéma de compréhension en tant que type de données, modulo l'ajout d'un axiome :

Axiome 5 (Axiome d'antisymétrie) *L'axiome suivant exprime l'antisymétrie du sous-typage :*

$$(AS) \vdash \forall A, B((A =_{\subset} B) \Rightarrow (A =_L B))$$

Cet axiome est évidemment vrai dans notre sémantique, car son interprétation est l'axiome d'antisymétrie !
On définit le schéma de compréhension d'un prédicat P, noté $SC(P)$ par :

$$SC(P) = \cup X(X \restriction (\mathcal{S}(X) \wedge P(X)))$$

Le fait suivant justifie l'appellation schéma de compréhension :

Fait 91 *Le séquent suivant est dérivable :*

$$\vdash \forall P \forall X((\mathcal{S}(X) \wedge P(X)) \Longleftrightarrow (\mathcal{S}(X) \wedge X \subset SC(P)))$$

preuve:
- $\Rightarrow$: On met en hypothèse $\mathcal{S}(X) \wedge P(X)$, et il suffit d'en déduire $X \subset \cup X(X \restriction (\mathcal{S}(X) \wedge P(X)))$, ce qui est immédiat.
- $\Leftarrow$: On met en hypothèse $\mathcal{S}(X) \wedge X \subset SC(P)$. Par définition de singleton, on en déduit $\exists Y(X \subset (Y \restriction (\mathcal{S}(Y) \wedge P(Y))))$. On met en hypothèse $Y : \tau, X \subset (Y \restriction (\mathcal{S}(Y) \wedge P(Y))))$. Par le point 6 du fait 30 page 54, et par définition de singleton, on a $\mathcal{S}(Y) \wedge P(Y)$. Par le point 2 du même fait, on a $X \subset Y$. Par l'équivalence des singletons (48 page 69), on a $\mathcal{S}_{\subset}(Y)$, et on en déduit donc que $X =_{\subset} Y$. Par l'axiome (AS), on en déduit $X =_L Y$, et donc $P(X)$. Y n'étant pas libre dans la conclusion, on a le résultat.

□

6.7 Conclusion

On a vu dans ce chapitre comment coder les types de données usuels dans le système ST. L'étude des entiers n'aura plus d'autre importance dans la suite que celle de représenter les constructeurs du langage. On ne s'intéressera plus en fait qu'au niveau "supérieur", celui des types de données (paires, sommes, enregistrements,...) que nous avons définies. Nous allons maintenant définir un petit langage de programmation basé sur ces types de données.

Chapitre 7

Langage de programmation

Nous présentons dans ce chapitre un langage de programmation fonctionnel typé avec spécifications[1].
Notre langage manipule quatre sortes d'objets :

1. Les *programmes*, qui seront écrits en police droite ($\mathsf{fun\ n} \rightarrow \mathsf{n + 1} \ldots$)
2. Les *formules*, qui servent à exprimer les propriétés de programmes. Par exemple, si on se donne le type "*nat*" des programmes qui représentent les entiers, la formule :

   $$\Pi^o n : nat.(n\ \mathtt{+}\ \mathtt{1} = \mathtt{1}\ \mathtt{+}\ n)$$

   doit se lire :

   "Pour tout entier n, $n\ \mathtt{+}\ \mathtt{1}$ égale $\mathtt{1}\ \mathtt{+}\ n$"
3. Les *types*, qui servent à décrire des ensembles de programmes. Par exemple, si on se donne en plus du types "*nat*", le type "*bool*" des booléens :

   $$\Pi^\tau n : nat.bool$$

   doit se lire :

   "L'ensemble des programmes qui prennent en entrée un entier, et renvoient comme résultat un booléen"
4. Les *jugements de sous-typage*, qui correspondent à la notion de sous-ensemble. Par exemple, si on se donne le type "*pair*" des entiers pairs :

   $$pair \subset nat$$

   doit se lire

   "tout entier pair est un entier"

[1] Bien que le langage repose in fine sur le système ST, via sa traduction établissant sa consistance, ce chapitre peut être lu de manière indépendante.

Ces quatre sortes d'objets sont définis par règles d'inférences et des grammaires, et les règles logiques, de typage et de sous-typage permettent d'exprimer leurs interactions.

Dans une première section, nous définirons les règles qui permettent de construire les formules, les types et les relations de sous-typage, et plus généralement nous définirons également les types et prédicats paramétrés, tous ces objets étant regroupés sous le vocable *terme*. Les termes seront construits grâce à des sortes simples restreintes à la position positive d'une sorte particulière , la sorte π, qui sera la sorte des programmes. Une fois les termes définis, nous donnerons dans les sections suivantes les règles du langage à proprement parler[2]. Dans la section 2, on présentera le noyau fonctionnel, qui concerne le langage de programmation restreint aux seules constructions d'abstraction ($\mathsf{fun}\ x \to t$) et d'application ($(u\ v)$). Cette partie est essentielle car elle contient les règles de typage du produit (Π^r) correspondant à l'abstraction, ainsi que les règles logiques de base. Les sections suivantes sont dédiées à la présentation des règles qui concernent les types de données plus élaborés :

- La section 3 concerne les paires. Les règles de typage sont les règles habituelles, la seule règle logique spécifique étant celle de la paire surjective.
- La section 4 concerne les types sommes, qui correspondent à la réunion de types discriminés par des constructeurs. Les règles grammaticales définissant les types sommes sont un peu fastidieuses, mais nécessaires pour obtenir la cohérence du système : elles pourront être sautées en première lecture.
- La section 5 présente les types inductifs, correspondant à la notion vue dans le chapitre 2. Les règles d'élimination sont restreintes à la récurrence à un pas et à la récurrence bien fondée.
- La section 6 présente les types coinductifs, correspondant à la notion vue dans le chapitre 2. Les section 5 et 6 peuvent être vues par le lecteur connaissant les types algébriques du type ML comme une façon de préciser à l'intérieur de ces types ceux qui terminent et ceux qui ne terminent (éventuellement) pas.
- La section 7 présente les enregistrements : une nouvelle notion y est présentée, celle d'enregistrement fort, qui permet de préciser les habitants d'un tel type, car en présence de sous-typage, un type enregistrement ne contient pas assez d'information pour cerner le comportement de ses habitants relativement à la terminaison.
- La section 8 présente deux notions d'abstractions :
  - L'abstraction correspondant aux types abstraits de ML, qui consiste à dissimuler un type (ou un type paramétré, plus généralement un terme dans notre formalisme) qui apparaît dans un autre type.
  - L'abstraction présentée sous la forme du lieur Self, permet d'exprimer le programme lui-même dans son type.
- La section 9 présente un prédicat de normalisation, vu comme une contrainte de sous-typage.

[2]On prendra la convention d'encadrer les règles qui concernent la construction de termes, et de ne pas encadrer les règles du langage

7.1 Sortes et termes

7.1.1 Sortes et termes de base

Nous présentons tout d'abord les règles qui permettent de construire ce que nous appelons les *termes* : formules, types et relations de sous-typage. Pour faciliter la traduction ultérieure, nous présentons les règles de grammaire sous la forme de règles d'inférence :

$$\frac{\text{Prémisse(s)}}{\text{Conclusion}}$$

On distingue deux sortes d'objets de base :
– La sorte des propositions, qui sera notée o (on notera les propositions (P, Q, ...).
– La sorte des types, qui sera notée τ (on notera les types A, B, ...).
Une autre sorte d'objets sera les jugements de sous-typage, de la forme $A \subset B$, A et B étant des types.
On se donne trois ensembles de variables :
– L'ensemble des variables de propositions V_o.
– L'ensemble des variables de types V_τ.
– L'ensemble des variables de programmes $\mathcal{V}$.
A l'aide de ces ensembles de variables, on construit des contextes, que l'on notera Γ, qui sont des listes qui comportent :
– des paires de la forme *variable :sorte* ou *variable :type*.
– des *propositions*
– des *jugement de sous-typage avec une variable à gauche*.
C'est ce que reflètent les règles de bonne formation de contexte (on note $WF(\Gamma)$ le jugement "Γ est bien formé").

Ces contextes servent à déclarer les variables libres dans un terme, et à construire les termes. Les termes peuvent être :
– Des variables déclarées dans le contexte.
– Des propositions, construites avec :
  – l'implication $\Rightarrow$ à partir de deux propositions
  – un quantificateur universel $\forall_s^o$ qui est un lieur portant sur les objets de sorte s.
  – un produit Π^o, qui est un lieur portant sur les programmes.
– Des types, construits avec :
  – un quantificateur universel $\forall_s^\tau$ qui est un lieur portant sur les objets de sorte s.
  – un produit Π^τ, qui est un lieur portant sur les programmes.
  – un schéma de compréhension $\{x : A; P\}$, qui est également un lieur.
C'est ce que reflètent les règles de formation des termes.
Le sens du schéma de compréhension est le suivant :

$\{x : A; P\}$ doit se lire : "l'ensemble des programmes de type A qui vérifient la propriété P"

Règles de bonne formation de contextes :

$$\frac{}{WF([])}$$

$$\frac{WF(\Gamma) \quad s = o \text{ ou } s = \tau \quad x \notin V(\Gamma) \quad x \in V_s}{WF(\Gamma, x : s)}$$

$$\frac{WF(\Gamma) \quad \Gamma \vdash A : \tau \quad x \notin V(\Gamma) \quad x \in \mathcal{V}}{WF(\Gamma, x : A)}$$

$$\frac{WF(\Gamma) \quad \Gamma \vdash P : o}{WF(\Gamma, P)}$$

$$\frac{WF(\Gamma) \quad x : \tau \in \Gamma \quad y : \tau \in \Gamma}{WF(\Gamma, x \subset y)}$$

Règles de formation des termes :

$$\frac{WF(\Gamma) \quad x : s \in \Gamma}{\Gamma \vdash x : s}$$

Propositions :

$$\frac{\Gamma \vdash P : o \quad \Gamma \vdash Q : o}{\Gamma \vdash P \Rightarrow Q : o} \qquad \frac{\Gamma, x : s \vdash P : o \quad s \neq \pi}{\Gamma \vdash \forall^o_s x.P : o}$$

$$\frac{\Gamma, x : A \vdash P : o}{\Gamma \vdash \Pi^o x : A.P : o}$$

Types :

$$\frac{\Gamma, x : s \vdash A : \tau \quad s \neq \pi}{\Gamma \vdash \forall^\tau_s x.A : \tau}$$

$$\frac{\Gamma, x : A \vdash B : \tau}{\Gamma \vdash \Pi^\tau x : A.B : \tau} \qquad \frac{\Gamma, x : A \vdash P : o}{\Gamma \vdash \{x : A; P\} : \tau}$$

7.1.2 Termes paramétrés

On ajoute aux termes définis précédemment des termes paramétrés qui peuvent être des prédicats ou des types paramétrés.
Pour les construire, on se donne une sorte supplémentaire : π, la sorte des programmes, et on définit les sortes dont la sorte la plus à droite est o ou τ par les règles suivantes :

$$\frac{}{\vdash \tau : *_r} \qquad \frac{}{\vdash o : *_r} \qquad \frac{}{\vdash \pi : *}$$

$$\frac{\vdash s : * \qquad \vdash r : *_r}{\vdash s \to r : *_r} \qquad \frac{\vdash r : *_r}{\vdash r : *}$$

Le jugement $s : *_r$ signifie "s est une sorte qui a le droit de se trouver à droite d'une flèche.

A chaque sorte $s : *$, on associe un ensemble de variables de sortes V_s, l'ensemble des variables de sorte π étant égal à l'ensemble des variables de programmes ($V_\pi = \mathcal{V}$), et on étend les règles de formation pour construire des termes simplement sortés :

Bonne formation de contextes :

$$\frac{WF(\Gamma) \qquad \vdash s : * \qquad x \notin V(\Gamma) \qquad x \in V_s}{WF(\Gamma, x : s)}$$

Termes sortés :

$$\frac{\Gamma, x : s \vdash t : s' \qquad \vdash s' : *_r}{\Gamma \vdash \lambda x.t : s \to s'} \qquad \frac{\Gamma \vdash u : s \to s' \qquad \Gamma \vdash v : s}{\Gamma \vdash u(v) : s'}$$

Propositions atomiques :

$$\frac{\Gamma \vdash u : \pi \qquad \Gamma \vdash v : \pi}{\Gamma \vdash u = v : o}$$

On identifiera les termes égaux par β-réduction : par exemple

$$\lambda X \lambda Y.(X \Rightarrow Y) \quad (P)(Q)$$

sera considéré comme égal à :

$$P \Rightarrow Q.$$

7.2 Le noyau fonctionnel

La partie la plus simple de notre langage concerne le typage des programmes du λ-calcul. On y distingue (et cela sera aussi vrai par la suite) quatre genres de dérivations :

- Les dérivations *logiques*, qui servent à raisonner sur les programmes.
- Les dérivations *de typage*, qui servent à donner un type aux programmes.
- Les dérivations *de sous-typage*, qui permettent de donner des relations entre les types.
- Les dérivations *de réduction*, qui permettent de réduire (c'est à dire d'évaluer) des programmes.

Nous présentons leurs règles dans la section suivante qui concerne les dérivations "pures". Ces quatre genres de dérivations interagissent par le biais de dérivations "mixtes", que nous étudions par la suite.

7.2.1 Dérivations pures

Dérivations de typage

Nous présentons ici le typage des programmes du noyau fonctionnel sont donnés par la syntaxe habituelle du λ-calcul :

$$\mathcal{P} = \quad \mathcal{V} \quad | \quad \mathsf{fun}\ \mathcal{V} \to \mathcal{P} \quad | \quad (\ \mathcal{P}\ \ \mathcal{P}\ )$$

Les occurrences libres de la variable x dans p sont liées dans $\mathsf{fun}\ x \to p$. Deux programmes α-équivalents sont considérés comme égaux.
Pour typer un programme, on impose la condition que le contexte Γ soit bien formé, et ne contienne pas de déclarations de variables de la forme $x : \pi$, ce que l'on note $WF_\pi(\Gamma)$.
Les règles de typage sont les cinq règles usuelles du λ-calcul, avec en plus la possibilité de lier la variable abstraite par fun dans le type :

$$\frac{WF_\pi(\Gamma) \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\, Ax^\tau$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \mathsf{fun}\ x \to t : \Pi^\tau x : A.B}\, \Pi_I^\tau \qquad \frac{\Gamma \vdash u : \Pi^\tau x : A.B \qquad \Gamma \vdash v : A}{\Gamma \vdash (u\ \ v) : B[x := v]}\, \Pi_E^\tau$$

$$\frac{\Gamma, x : s \vdash t : A}{\Gamma \vdash t : \forall_s^\tau x.A}\, \forall_I^\tau \qquad \frac{\Gamma \vdash t : \forall_s^\tau x.A \qquad \Gamma \vdash v : s}{\Gamma \vdash t : A[x := v]}\, \forall_E^\tau$$

Pour faciliter la programmation et la lisibilité des types, nous introduisons deux notations :

Notation 14

- *`let` x`=` p `in` q est une notation pour (`fun` x `->` q p), (les occurrences libres de x dans q sont donc liées dans `let` x `=` q `in` p).*
- *Si x n'est pas libre dans B, on notera $A \Rightarrow^\tau B$ au lieu de $\Pi^\tau x : A.B$.*

On a donc les règles correspondantes :

$$\frac{\Gamma \vdash p : A \qquad \Gamma, x : A \vdash q : B}{\Gamma \vdash \mathsf{let}\ x = p\ \mathsf{in}\ q : B[x := p]}\ \mathsf{let}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad x \text{ non libre dans } B}{\Gamma \vdash \mathsf{fun}\ x \to t : A \Rightarrow^\tau B} \Rightarrow^\tau_I$$

$$\frac{\Gamma \vdash u : A \Rightarrow^\tau B \qquad \Gamma \vdash v : A}{\Gamma \vdash (u\ v) : B} \Rightarrow^\tau_E$$

Dérivations logiques

Dans la construction des termes, on a introduit comme propositions l'égalité entre objets de la sorte π. Comme on manipule ici des contextes dénués de déclaration de variables de sorte π, les termes de cette sorte qui seront introduits seront les programmes[3] :

$$\boxed{\frac{\Gamma \vdash u : A}{\Gamma \vdash u : \pi}\ prog2\pi}$$

On déduit des jugements logiques de la manière usuelle, en ayant de plus, comme pour les règles de typage, la possibilité de lier une variable de programme dans les propositions à l'aide du produit :

[3] On écrira les règles grammaticales dans des encadrés, et les règles du système non encadrées

$$\frac{P \in \Gamma}{\Gamma \vdash P}\, Ax^o$$

$$\frac{\Gamma, x : A \vdash P}{\Gamma \vdash \Pi^o x : A.P}\, \Pi^o_I \qquad \frac{\Gamma \vdash \Pi^o x : A.P(x) \qquad \Gamma \vdash t : A}{\Gamma \vdash P[x := t]}\, \Pi^o_E$$

$$\frac{\Gamma, x : s \vdash P}{\Gamma \vdash \forall^o_s x.P}\, \forall^o_I \qquad \frac{\Gamma \vdash \forall^o_s x.P \qquad \Gamma \vdash t : s}{\Gamma \vdash P[x := t]}\, \forall^o_E$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow^o Q}\, \Rightarrow^o_I \qquad \frac{\Gamma \vdash P \Rightarrow^o Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\, \Rightarrow^o_E$$

On définit de plus les connecteurs logiques habituels :

Définition 53 (Opérateurs) *Les connecteurs logiques sont définis par leur codage usuel au second ordre :*
- $\bot_o = \forall^o_o X.X$
- $A \wedge B = \forall^o_o X((A \Rightarrow B \Rightarrow X) \Rightarrow X)$
- $A \vee B = \forall^o_o X((A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X)$
- $A \Longleftrightarrow B = (A \Rightarrow B) \wedge (B \Rightarrow A)$
- $\neg = \lambda A.(A \Rightarrow \bot_o)$

Les règles logiques associées (introduction et élimination) sont les règles usuelles. Cependant, pour ce qui concerne l'égalité et l'absurde, nous ajoutons des règles particulières :
- La règle de l'égalité :

$$\frac{}{\vdash \forall^o_\tau A, B.\Pi^o a : A.\Pi^o b : B.(a = b \Longleftrightarrow \forall^o_{\pi \to o} X.(X(a) \Rightarrow X(b)))}\, =_E$$

  Cette règle exprime que l'égalité est l'égalité de Leibnitz, permettant de subsituer dans tout prédicat un programme par son égal.
- La règle du raisonnement classique :

$$\frac{}{\vdash \forall^o_o A(\neg\neg A \Rightarrow A)}\, RC$$

- La règle de non trivialité[4], qui affirme qu'il existe des programmes distincts :

$$\frac{}{\vdash \neg\forall X \Pi^o x : X.\forall Y \Pi^o y : Y.(x = y)}\, \neg\forall =$$

Nous verrons dans un exemple donné en fin de section comment utiliser cette règle pour distinguer deux programmes.

[4]appellée ainsi en référence à la règle correspondante de $ST_{\neg TRIV}$

Dérivations de sous-typage

Les règles qui permettent d'inférer les jugements de sous-typage sont restreintes aux règles propres au sous-typage, que nous détaillons ci-après.

– Axiome :

$$\frac{(X \subset Y) \in \Gamma}{\Gamma \vdash X \subset Y} \subset_{Ax}$$

– Réflexivité :

$$\frac{\Gamma \vdash A : \tau}{\Gamma \vdash A \subset A} \subset_{Refl}$$

– Transitivité :

$$\frac{\Gamma \vdash A \subset B \qquad \Gamma \vdash B \subset C}{\Gamma \vdash A \subset C} \subset_{Trans}$$

– Produit :

$$\frac{\Gamma \vdash A' \subset A \qquad \Gamma, x : A' \vdash B \subset B'}{\Gamma \vdash \Pi^{\tau} x : A.B \subset \Pi^{\tau} x : A'.B'} \subset_{\Pi^{\tau}}$$

– Croissance du schéma de compréhension pour $\subset$:

$$\frac{\Gamma \vdash A \subset B}{\Gamma \vdash \{x : A; P(x)\} \subset \{x : B; P(x)\}} \subset_{SC-CR\subset}$$

– Croissance du schéma de compréhension pour $\Rightarrow^o$:

$$\frac{\Gamma \vdash \Pi^o x : A(P(x) \Rightarrow^o Q(x))}{\Gamma \vdash \{x : A; P(x)\} \subset \{x : A; Q(x)\}} \subset_{SC-CR\Rightarrow^o}$$

La règle de sous-typage du produit se justifie intuitivement facilement en reprenant l'intuition présentée dans l'introduction, et les règles relatives au schéma de compréhension également, en reprenant la présentation intuitive faite à la fin de la section 5.1.

Dérivations de Réduction

Les jugements de réduction sont de la forme $\vdash u \succ v$, avec u et v deux programmes. Les règles pour construire des jugements de réduction dépendent de la définition de la réduction des programmes, qui sont données en annexe B.

$$\frac{u \succ v}{\vdash u \succ v} \succ$$

7.2.2 Interactions entre les dérivations

Interactions entre Typage et Logique

Les règles mixtes Typage- Formules sont les suivantes :
– Introduction du schéma de compréhension :

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash P(t)}{\Gamma \vdash t : \{x : A; P(x)\}} SC_I$$

– Élimination droite du schéma de compréhension :

$$\frac{\Gamma \vdash t : \{x : A; P(x)\}}{\Gamma \vdash P(t)} SC_E^o$$

– Élimination gauche du schéma de compréhension :

$$\frac{\Gamma \vdash t : \{x : A; P(x)\}}{\Gamma \vdash t : A} SC_E^\tau$$

Ces trois règles concernent en fait uniquement le schéma de compréhension. Vu sa description intuitive de la fin de la section 5.1, elles se comprennent assez facilement : pour montrer qu'un programme a le type $\{x : A; P(x)\}$, il suffit de montrer qu'il a le type A et qu'il vérifie la propriété P (règle d'introduction). Si un programme a le type $\{x : A; P(x)\}$, alors il a le type A (élimination gauche) et il vérifie la propriété P (élimination droite).

Interaction entre Typage et Sous-typage

Il n'y a qu'une interaction entre typage et sous typage :

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \subset B}{\Gamma \vdash t : B} \subset$$

Cette règle est claire avec l'intuition donnée en introduction.

Interaction entre Réduction et Logique

Il n'y a qu'une interaction entre typage et réduction, qui est donnée par la règle $Lo_\succ$, qui dit "Tout réduit d'un programme typé lui est égal" (la règle $\succ_\pi$ servant simplement à assurer le bon sortage de v) :

$$\frac{\Gamma \vdash u : A \qquad \vdash u \succ v}{\Gamma \vdash v : \pi} \succ_\pi \qquad \frac{\Gamma \vdash u : A \qquad \vdash u \succ v}{\Gamma \vdash (u = v)} Lo_\succ$$

7.2.3 Exemple

A tout seigneur, tout honneur : nous commençons par un exemple qui concerne le fameux type des booléens, en hommage à George Boole (1815-1864).
Nous posons pour cet exemple les notations suivantes :

- $B_F = \forall_\tau X(X \Rightarrow^\tau (X \Rightarrow^\tau X))$ le type des booléens du système F.
- $\mathsf{true} = \mathsf{fun}\ \mathsf{x} \rightarrow \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{x}$.
- $\mathsf{false} = \mathsf{fun}\ \mathsf{x} \rightarrow \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{x}$.
- $\mathsf{not}(b) = (\ (\ b\ \mathsf{false}\ )\ \mathsf{true}\ )$.
- $\mathsf{not} = \mathsf{fun}\ b \rightarrow \mathsf{not}(b)$.
- $\mathbb{B} = \{b : B_F; b = \mathsf{true} \vee b = \mathsf{false}\}$

Les quatre dérivations suivantes sont usuelles :

- $\vdash B_F : \tau$
- $\vdash \mathsf{true} : B_F$
- $\vdash \mathsf{false} : B_F$
- $b : B_F \vdash \mathsf{not}(b) : B_F$

Nous souhaitons dériver maintenant la spécification suivante (peu concise certes, mais précise) pour le programme not :

$$\vdash \mathsf{not} : \Pi^\tau b : \mathbb{B}.\{r : \mathbb{B}; (b = \mathsf{true} \Rightarrow r = \mathsf{false}) \wedge (b = \mathsf{false} \Rightarrow r = \mathsf{true})\}$$

Elle se lit :
“ not est un programme qui prend en entrée un booléen b, et renvoie en résultat un booléen r tel que, si $b = \mathsf{true}$, alors $r = \mathsf{false}$, et réciproquement”.

Pour inférer ce typage, on construit la dérivation suivante, sans détailler la sous-dérivations de typage π_{Ty} et la sous-dérivation logique π_{Lo} :

$$\dfrac{\dfrac{\begin{array}{cc}\pi_{Ty} & \pi_{Lo} \\ \vdots & \vdots \\ b : \mathbb{B} \vdash \mathsf{not}(b) : \mathbb{B} & b : \mathbb{B} \vdash b = \mathsf{true} \Rightarrow^o \mathsf{not}(b) = \mathsf{false} \wedge b = \mathsf{false} \Rightarrow^o \mathtt{not}(b) = \mathtt{true}\end{array}}{b : \mathbb{B} \vdash \mathtt{not}(b) : \{r : \mathbb{B}; b = \mathtt{true} \Rightarrow^o r = \mathtt{false} \wedge b = \mathtt{false} \Rightarrow^o r = \mathtt{true}\}}\, SC_I}{\vdash \mathtt{not} : \Pi^\tau b : \mathbb{B}.\{r : \mathbb{B}; b = \mathtt{true} \Rightarrow^o r = \mathtt{false} \wedge b = \mathtt{false} \Rightarrow^o r = \mathtt{true}\}}\, \Pi_I^\tau$$

Nous laissons aux lecteurs le soin de compléter les sous-dérivations.

7.2.4 A propos de l'absurde

Pour illustrer une utilisation de la règle $\neg\forall =$, et dans la continuation de l'exemple précédent, nous montrons comment prouver que deux programmes sont distincts, et comment les faux interagissent.

Fait 92 *En reprenant les notations de l'exemple précédent, on montre :*

$$\vdash \neg(\mathit{false} = \mathit{true})$$

preuve: Il est aisé de montrer (avec les règles $\succ$ et un typage adéquat de `true` et `false`) que le séquent suivant est dérivable :

$$(\texttt{false} = \texttt{true}), X : \tau, x : X, Y : \tau, y : Y \vdash x = y$$

On conclut en utilisant $\neg\forall =$. □

Plus généralement, on montrerait que deux termes résolubles non β équivalents sont prouvablement distincts dans notre système (à l'aide du théorème de Böhm).
On montre maintenant que le type vide n'est pas habité, sous la forme suivante :

Fait 93 *Le séquent suivant est dérivable :*

$$\vdash \Pi^o x : \forall XX.\bot_o$$

preuve: Sous l'hypothèse $x : \forall XX$, on déduit $x : \forall A, B\Pi^\tau a : A.\Pi^\tau b : B\{r : A; r = a\}$ et $x : \forall A, B\Pi^\tau a : A.\Pi^\tau b : B\{r : B; r = b\}$, ce qui permet de conclure de la même façon que précédemment. □

Nos exemples montrent simplement comment on peut raisonner sur des λ-termes purs, ce qui est assez peu utile du point de vue de la programmation "réelle". C'est pourquoi nous allons introduire progressivement des types de données plus élaborés, à commencer par les paires.

7.3 Paires

On ajoute au langage de programmations trois nouveaux opérateurs :

- l'opérateur de construction de paires `(.,.)`
- les premières et secondes projections `fst` et `snd`

$$\mathcal{P} = \quad \ldots \quad | \quad (\mathcal{P},\mathcal{P}) \quad | \quad \mathsf{fst} \quad \mathcal{P} \quad | \quad \mathsf{snd} \quad \mathcal{P}$$

Et on ajoute aux types les types produits :

$$\boxed{\frac{\Gamma \vdash A : \tau \qquad \Gamma \vdash B : \tau}{\Gamma \vdash A \times B : \tau}}$$

Les règles de typage et de sous-typage des types produits sont les règles usuelles, plus la paire surjective :

- Introduction de la paire :

$$\frac{\Gamma \vdash p : A \qquad \Gamma \vdash q : B}{\Gamma \vdash (p,q) : A \times B} \times I$$

- Élimination gauche de la paire :

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{fst}\ p : A} \times EG$$

- Élimination droite de la paire :

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{snd}\ \mathsf{p} : B} \times ED$$

- Sous-typage du produit :

$$\frac{\Gamma \vdash A \subset A' \qquad \Gamma \vdash B \subset B'}{\Gamma \vdash A \times B \subset A' \times B'} \subset_{\times}$$

- Paire surjective :

$$\frac{}{\vdash \forall A, B \Pi^{o} c : A \times B.(c = (\mathsf{fst}\ c,\mathsf{snd}\ c))} SurPair$$

Exemple : la curryfication et la décurryfication

On définit les opérateurs curry et uncurry par :

$$\mathsf{curry}= \mathsf{fun\ f} \rightarrow \mathsf{fun\ x} \rightarrow \mathsf{fun\ y} \rightarrow \mathsf{f(x,y)}$$

$$\mathsf{uncurry} = \mathsf{fun\ g} \rightarrow \mathsf{fun\ c} \rightarrow \mathsf{g\ (fst\ c)\ (snd\ c)}$$

Et on peut les typer de la façon suivante :

$$\vdash \mathsf{curry} : \forall A, B \forall C \Pi \mathsf{f} : (\Pi c : A \times B.C(\mathsf{c})).\Pi \mathsf{x} : A.\Pi \mathsf{y} : B.C((\mathsf{x},\mathsf{y}))$$

$$\vdash \mathsf{uncurry} : \forall A, B \forall C \Pi \mathsf{g} : (\Pi \mathsf{x} : A.\Pi \mathsf{y} : B.C(\mathsf{x})(\mathsf{y})).\Pi \mathsf{c} : A \times B.C(\mathsf{fst\ c})(\mathsf{snd\ c})$$

On a, de plus, les deux propriétés :

$$\vdash \begin{array}{c} \forall A, B \forall C \Pi^o g : (A \Rightarrow B \Rightarrow C).\Pi^o x : A.\Pi^o y : B. \\ ((\mathsf{curry\ (uncurry}\ g))\ x\ y) = ((g\ x)\ y) \end{array}$$

$$\vdash \begin{array}{c} \forall A, B \forall C \Pi^o f : (A \times B \Rightarrow C).\Pi^o c : A \times B. \\ (((\mathsf{uncurry\ (curry}\ f))\ c) = (f\ c) \end{array}$$

Les deux égalités se prouvent facilement : la première en utilisant $\succ$, et la deuxième à l'aide de la règle $SurPair$.

On n'a ni (curry (uncurry g)) = g, ni ((uncurry (curry f))=f, même pour un g et un f du bon type comme le montrent les contre-exemples suivants :

- $g = \mathsf{fun\ x} \rightarrow \mathsf{x} : (B \Rightarrow C) \Rightarrow (B \Rightarrow C)$, dans ce cas :

$$(\mathsf{curry\ (uncurry}\ g)) = \mathsf{fun\ x} \rightarrow \mathsf{fun\ y} \rightarrow \mathsf{(x\ y)} \neq g$$

- $f = \mathsf{fun\ x} \rightarrow \mathsf{x} : A \times B \Rightarrow A \times B$, dans ce cas :

$$((\mathsf{uncurry\ (curry}\ f)) = \mathsf{fun\ x} \rightarrow \mathsf{(fst\ x\ ,\ snd\ x)} \neq f$$

Exemple : l'inversion des paires

On définit le terme inv par : $\mathsf{inv} = \mathsf{fun\ c} \rightarrow \mathsf{(snd\ c,\ fst\ c)}$
Le typage suivant :

$$\vdash \mathsf{inv} : \forall A, B(A \times B \Rightarrow B \times A)$$

se vérifie facilement.
Et la proposition suivante :

$$\vdash \forall A, B \Pi^o \mathsf{c} : A \times B.(\mathsf{c} = \mathsf{inv\ (inv\ c)})$$

se prouve à l'aide de la règle $SurPair$.

7.4 Sommes

7.4.1 Types sommes

Nous ajoutons une nouvelle sorte τ_S, qui n'intervient pas dans les constructions de sortes (i.e. on n'a pas $\tau_S : *$), mais qui sert à construire des nouveaux types. On se donne un ensemble dénombrable $\mathcal{C} = \{C_1; C_2; \ldots\}$ de constructeurs muni d'un prédicat d'égalité.

Les règles de construction des types sommes, mutuellement définies avec les règles d'absence d'un constructeur C dans un type somme T (noté $C \notin T$), sont :

$$\frac{WF(\Gamma) \qquad C \in \mathcal{C}}{\Gamma \vdash C : \tau_S}\, C_0 \qquad \frac{\Gamma \vdash A : \tau \qquad C \in \mathcal{C}}{\Gamma \vdash C \text{ of } A : \tau_S}\, C_1$$

$$\frac{\Gamma \vdash T : \tau_S \quad C \in \mathcal{C} \quad C \notin T}{\Gamma \vdash C \mid T : \tau_S}\, Sum_0 \qquad \frac{\Gamma \vdash T : \tau_S \quad \Gamma \vdash A : \tau \quad C \in \mathcal{C} \quad C \notin T}{\Gamma \vdash C \text{ of } A \mid T : \tau_S}\, Sum_1$$

$$\frac{WF(\Gamma) \quad C_1 \in \mathcal{C} \quad C_2 \in \mathcal{C} \quad C_1 \neq C_2}{\Gamma \vdash C_1 \notin C_2}\, \notin_{C_0} \qquad \frac{\Gamma \vdash A : \tau \qquad C_1 \in \mathcal{C} \quad C_2 \in \mathcal{C} \quad C_1 \neq C_2}{\Gamma \vdash C_1 \notin C_2 \text{ of } A}\, \notin_{C_1}$$

$$\frac{\Gamma \vdash T : \tau_S \quad C_1 \in \mathcal{C} \quad C_2 \in \mathcal{C} \quad C_1 \neq C_2 \quad C_1 \notin T}{\Gamma \vdash C_1 \notin C_2 \mid T}\, \notin_{Sum_0}$$

$$\frac{\Gamma \vdash T : \tau_S \quad \Gamma \vdash A : \tau \quad C_1 \in \mathcal{C} \quad C_2 \in \mathcal{C} \quad C_1 \neq C_2 \quad C_1 \notin T}{\Gamma \vdash C_1 \notin C_2 \text{ of } A \mid T}\, \notin_{Sum_1}$$

$$\frac{\Gamma \vdash T : \tau_S}{\Gamma \vdash T : \tau}\, \notin_{Sum_1}$$

On rappelle que les règles encadrées sont des règles de grammaire. Celles ci signifient simplement qu'on ne peut construire un type somme qu'en effectuant la réunion (symbolisée par $|$) que sur des atomes de la forme C ou C of A tous disjoints. La dernière règle exprime simplement qu'un type somme est un type.

7.4.2 Règles de sous-typage

Les règles de sous-typage des types sommes correspondent aux intuitions sous-jacentes de réunion ensembliste.

Notation 15 *Pour économiser les règles :*
- *on notera $C*$ pour C of A ou C.*
- *$P \wedge Q$ servira à noter qu'on a deux règles avec les mêmes prémisses, l'une de conclusion P, l'autre de conclusion Q.*

Nous avons deux règles de sous-typage structurel :

$$\frac{\Gamma \vdash C* \mid T : \tau_S}{\Gamma \vdash C* \subset (C* \mid T) \quad \wedge \quad T \subset (C* \mid T)} \subset_{-}$$

$$\frac{\Gamma \vdash A \subset B}{\Gamma \vdash \; C \text{ of } A \quad \subset \quad C \text{ of } B} \subset_{of}$$

$\subset_{-}$ exprime que l'on peut omettre une des composantes de la réunion à gauche de $\subset$. $\subset_{of}$ exprime que pour sous-typer un type somme atomique de la forme C of B, il faut prendre un sous-type de B.

Les autres règles sont des règles de réécriture. Pour éviter encore les duplications, nous utilisons la notation $A =_{\subset} B$ pour $(A \subset B) \wedge (B \subset A)$.

- Commutativité :

$$\frac{\Gamma \vdash \quad C_1* \mid C_2* : \tau_S}{\Gamma \vdash \quad (C_1* \mid C_2*) \quad =_{\subset} \quad (C_2* \mid C_1*)} Comm_{|}$$

- Associativité :

$$\frac{\Gamma \vdash T : \tau_S \qquad \Gamma \vdash \quad C_1* \mid C_2* \mid T : \tau_S}{\Gamma \vdash \quad (C_1* \mid C_2* \mid T) \quad =_{\subset} \quad (C_2* \mid C_1* \mid T)} Assoc_{|}$$

Nous verrons quelques applications de ces règles dans la section consacrée aux types inductifs.

7.4.3 Typage

Le langage s'enrichit de la façon suivante :

$$\begin{array}{lcl} \mathcal{P} & = & \ldots \quad | \quad \mathcal{C} \quad | \quad \mathcal{C}[\mathcal{P}] \quad | \quad \mathsf{cases}\ \mathcal{P}\ \mathsf{of}\ \mathcal{M} \\ \mathcal{M} & = & \mathcal{N} \quad | \quad \mathcal{N}|\mathcal{M} \\ \mathcal{N} & = & \mathcal{C}\ \mathcal{V} \rightarrow \mathcal{P} \quad | \quad \mathcal{C} \rightarrow \mathcal{P} \quad | \quad \_ \rightarrow \mathcal{P} \end{array}$$

L'ensemble des constructeurs $\mathcal{C}$ du langage étant le même que celui des types sommes.

Les constructions C et $C[p]$ servent à introduire des éléments de types sommes.
La construction $\mathsf{cases}\ p\ \mathsf{of}\ M$ sert à filtrer le programme p selon les différents motifs de M qui peuvent être :
– "p est de la forme $C[x]$, alors on renvoie q" : $Cx \rightarrow q$
– "p est de la forme C, alors on renvoie q" : $C \rightarrow q$
– "Dans les autres cas, on renvoie q (motif universel)" : $\_ \rightarrow q$.

Nous présentons les règles d'introduction puis les règles d'élimination. Les deux règles d'introduction :

– Introduction d'un constructeur seul :

$$\frac{WF(\Gamma) \qquad C \in \mathcal{C}}{\Gamma \vdash C : C}\, Constr0_I$$

– Introduction d'un constructeur avec argument :

$$\frac{\Gamma \vdash p : A \qquad C \in \mathcal{C}}{\Gamma \vdash C[p] : C\ \mathsf{of}\ A}\, Constr1_I$$

Les quatre règles d'élimination :

– Un cas (cas universel) :

$$\frac{\Gamma \vdash A : \tau_S \quad \Gamma \vdash p : A \qquad \Gamma, x : A \vdash f : B(x)}{\Gamma \vdash \mathsf{cases}\ p\ \mathsf{of}\ \_ \rightarrow f\ : B(p)}\ \_$$

– Un cas (constructeur avec argument) :

$$\frac{\Gamma \vdash p : C\ \mathsf{of}\ A \qquad \Gamma, x : A \vdash f : B(C[x])}{\Gamma \vdash \mathsf{cases}\ p\ \mathsf{of}\ C\ x \rightarrow f\ : B(p)}\, Constr11_E$$

– Un cas (constructeur sans argument) :

$$\frac{\Gamma \vdash p : C \qquad \Gamma \vdash f : B(C)}{\Gamma \vdash \mathsf{cases}\ p\ \mathsf{of}\ C \rightarrow f\ \ : B(p)}\ Constr01_E$$

– Deux cas (constructeur sans argument) :

$$\Gamma \vdash C \notin S$$

$$\Gamma \vdash p : C \mid S$$

$$\Gamma \vdash f : B(C)$$

$$\frac{\Gamma, y : S \vdash \mathsf{cases}\ y\ \mathsf{of}\ M(y) : B(y)}{\Gamma \vdash \mathsf{cases}\ p\ \mathsf{of}\ C \rightarrow f \mid M(p) : B(p)}\ Constr02_E$$

– Deux cas (constructeur avec argument) :

$$\Gamma \vdash C \notin S$$

$$\Gamma \vdash p : (C\ \mathsf{of}\ A) \mid S$$

$$\Gamma, x : A \vdash f : B(C[x])$$

$$\frac{\Gamma, y : S \vdash \mathsf{cases}\ y\ \mathsf{of}\ M(y) : B(y)}{\Gamma \vdash \mathsf{cases}\ p\ \mathsf{of}\ C\ x \rightarrow f \mid M(p) : B(p)}\ Constr12_E$$

Ces quatre règles expriment simplement l'analyse par cas pour les deux dernières : soit l'on est dans $C*$, dans ce cas on obtient le résultat donné par le premier motif, soit l'on est dans T, et alors on obtient le résultat donné par le motif suivant.

7.4.4 Règles logiques

Les règles logiques servent d'une part à discriminer les constructeurs, et d'autre part à donner les règles correspondant à la réunion.

– Distingabilité des constructeurs :

$$\frac{C, C' \in \mathcal{C} \qquad C \neq C'}{\vdash \forall A \Pi^o x : A.C \neq C'[x]}\ C_{\neq}$$

Nous verrons un exemple d'application de cette règle dans le typage de la fonction partielle tail sur les listes.

– Propriétés des atomes sans arguments :

$$\frac{C \in \mathcal{C}}{\vdash \quad \Pi^o x : C.P(x) \quad \Longleftrightarrow \quad P(C)} C_{log}$$

– Propriétés des atomes avec argument :

$$\frac{C \in \mathcal{C}}{A : \tau \vdash \quad \Pi^o x : (C \text{ of } A).P(x) \quad \Longleftrightarrow \quad \Pi^o x : A.P(C[x])} C \text{ of }_{log}$$

– Propriétés des sommes :

$$\frac{\Gamma \vdash A|B : \tau_S \qquad \Gamma, x : A \vdash P(x) \qquad \Gamma, x : B \vdash P(x)}{\Gamma \vdash \Pi^o x : (A|B).P(x)} |_{log}$$

– Typage par disjonction :

$$\frac{\Gamma \vdash A|B : \tau_S \qquad \Gamma, x : A \vdash p : T(x) \qquad \Gamma, x : B \vdash p : T(x)}{\Gamma, x : A|B \vdash p : T(x)} |_{typ}$$

7.4.5 Exemples

Les booléens (2)

Il est immédiat de programmer les booléens à l'aide des types sommes : nous montrons ici comment spécifier un ifthenelse dans notre langage. Posons :

– Bool = True | False
– ifthenelse = fun b → fun x → fun y → cases b of | True→x | False→y

Et on infère que ifthenelse a le type suivant :

$$\forall A \Pi^\tau b : \mathsf{Bool}.\Pi^\tau x : A.\Pi^\tau y : A.\{r : A; (b = \mathsf{True} \Rightarrow r = x) \wedge (b = \mathsf{False} \Rightarrow r = y)\}$$

Type "Option"

On définit le type paramétré Option qui contient ou non un élément de type A :

$$\mathsf{Option} = \lambda A.\mathsf{None} \mid \mathsf{Some\ of}\ A$$

7.5 Types inductifs

Les types inductifs, dont le prototype est celui des entiers naturels, apparaissent via la notion de plus petit point fixe telle que présentée dans le chapitre 2.
Nous les restreignons aux cas gardés par des constructeurs et aux cas croissants pour obtenir les types usuels.

7.5.1 Algèbricité

On rajoute au langage de programmation un nouvel opérateur rec :

$$\mathcal{P} = \quad \ldots \quad | \quad (\ \mathsf{rec}\ \mathcal{P}\ )$$

Pour construire les types inductifs, on se donne deux conditions de gardes que nous appelons conditions d'algébricité :

Définition 54 *Soit* Γ *un contexte bien formé. On dit qu'un terme* F *est* ***algébrique*** *sous* Γ *(et l'on note* $\Gamma \vdash Alg(F)$ *si : les séquents suivants sont démontrables :*

$$(sum) \qquad \Gamma, X : \tau \vdash F(X) : \tau_S$$

$$(cresc) \quad \Gamma, X : \tau, Y : \tau, X \subset Y \vdash F(X) \subset F(Y)$$

On note $\Gamma \vdash Alg(F)$ *le fait que* F *soit algébrique sous* Γ *.*

(sum) sert à restreindre les types inductifs aux cas gardés par un constructeur.
$(cresc)$ sert à montrer que F est croissante, garantissant la condition qui fait de $\mu X.F(X)$ un plus petit point fixe
On ajoute aux types les types inductifs introduits par le lieur μ :

$$\frac{\Gamma \vdash Alg(F)}{\Gamma \vdash \mu X.F : \tau}\ \mu_\tau$$

7.5.2 Logique et typage

Les deux règles de base sont le *principe d'induction*, règle logique, et la *règle du récurseur*, règle de typage :

– Principe d'induction :

$$\frac{\Gamma, \quad A:\tau, \quad \Pi^o x : A.P(x), \quad A \subset \mu X.F(X) \quad \vdash \quad \Pi^o x : F(A).P(x)}{\Gamma \vdash \Pi^o x : (\mu X.F(X)).P(x)} \, Induc$$

– Règle du récurseur (μ) :

$$\frac{\Gamma, \quad A:\tau, \quad f : \Pi^\tau x : A.B(x), \quad A \subset \mu X.F(X) \quad \vdash \quad t : \Pi^\tau x : F(A).B(x)}{\Gamma \vdash (\mathsf{rec\ fun}\ f \to t) : \Pi x : (\mu X.F(X)).B(x)} \, Rec_\mu$$

Ces deux règles sont très similaires, la seule chose qui les distingue vraiment est que l'une porte sur la sorte o, l'autre sur la sorte τ.

On donne, en plus des deux règles précédentes, qui correspondent à la récurrence "à un pas", la règle de récurrence bien fondée. Celle-ci correspond au principe de démonstration suivant sur les "vrais" entiers naturels :

$$\forall n : \mathbb{N}(\forall m : \mathbb{N}.(m < n \Rightarrow P(m)) \Rightarrow P(n)) \quad \Rightarrow \quad \forall n : \mathbb{N}.P(n).$$

On généralise ce principe en définissant tout d'abord ce qu'est une relation bien fondée :

Définition 55 (Relation bien fondée) *On dit qu'une relation $R : \pi \to \pi \to o$ est bien fondée sur le type A si :*

$$\vdash \forall^o_{\pi \to o} P.(\Pi^o y : A(\Pi^o x : A.(R(x,y) \Rightarrow P(x)) \Rightarrow P(y)) \quad \Rightarrow \quad \Pi^o y : A.P(y))$$

On le notera $BF(R, A)$.

On donne la règle de récurrence bien fondée :

$$\frac{\Gamma, \quad y : \mu x F(x), \quad f : \Pi^\tau x : \{\mu x.F(x); R(x,y)\}.B(x) \vdash t : B(y) \qquad \Gamma \vdash BF(R, \mu x.F(x))}{\Gamma \vdash (\mathsf{rec\ fun}\ f \to \mathsf{fun}\ y \to t) : \Pi x : (\mu X.F(X)).B(x)} \, BFRec_\mu$$

Pour comprendre cette règle, on peut prendre pour $\mu x.F(x)$ son type inductif préféré (entiers, listes, arbres,...) et pour relation un argument usuel d'induction ($x < y$, longueur(x)<longueur(y), hauteur(x)<hauteur(y), ...).

7.5.3 Sous-typage

On se donne les deux règles suivantes de sous-typage pour μ :

– Un type inductif est un point fixe :

$$\frac{}{\vdash (F(\mu X.F(X)) =_{\subset} \mu x.F(X))}\ \mu_{PF}$$

– Un type inductif est le plus petit point fixe :

$$\frac{\Gamma \vdash F(A) \subset A}{\Gamma \vdash \mu X F(X) \subset A}\ \subset_{\mu}$$

7.5.4 Exemples

Les entiers relatifs

Nous voyons ici comment typer de façon très simple et très intuitive l'ensemble des entiers relatifs, vus comme la réunion de l'ensemble des successeurs de zéro et de l'ensemble des prédécesseurs de zéro.
On pose :
– $\mathsf{Pos} = \mu X.(\mathsf{Zero} \mid \mathsf{Succ\ of}\ X)$
– $\mathsf{Neg} = \mu X.(\mathsf{Zero} \mid \mathsf{Pred\ of}\ X)$
Et on pose :

$$\mathsf{Rel} = \ \mathsf{Pred\ of\ Neg} \mid \mathsf{Zero} \mid \mathsf{Succ\ of\ Pos}$$

Voyons par exemple comment typer le successeur sur cet ensemble :

$$\vdash \mathsf{fun\ n} \rightarrow \mathsf{cases\ n\ of\ Pred\ x} \rightarrow \mathsf{x} \mid \mathsf{y} \rightarrow \mathsf{Succ\ [y]} : \mathsf{Rel} \Rightarrow \mathsf{Rel}$$

Remarque: Nous ne sommes pas parvenus à donner le type voulu dans L'extension proposée par Objective Caml (version 3.06) des Variants Polymorphes qui infère pour le terme succ le type suivant :

$$\mathsf{val\ succ} : (\mathsf{[>\ `Pred\ of\ [>\ `Succ\ of\ 'a]\ as\ 'b]\ as\ 'a)} \rightarrow \mathsf{'b}$$

Et refuse de le typer avec le type voulu.

Listes

Donnons nous le type des paramétré des listes :

$$\mathsf{List}\ = \lambda A.\mu X.(\mathsf{Nil} \mid \mathsf{Cons\ of}\ A \times X)$$

La fonction qui prend la tête d'une liste reçoit le type :

$$\vdash \mathsf{fun\ l} \rightarrow \mathsf{cases\ l\ of\ Cons\ y} \rightarrow \mathsf{(fst\ y)} : \forall^{\tau} A.\Pi^{\tau} l : \{l : \mathsf{List}(A); l \neq \mathsf{Nil}\}.A$$

D'autres exemples plus sophistiqués utilisant le récurseur seront donnés en annexes.

7.6 Types co-inductifs

On ajoute aux types les types co-inductifs introduits par le lieur ν :

$$\frac{\Gamma \vdash Alg(F)}{\Gamma \vdash \nu X.F : \tau}\,\nu_\tau$$

La condition $Alg(F)$ a été définie à la section précédente, et nous l'utilisons pour les mêmes raisons.

7.6.1 Typage et logique

On dispose d'une règle logique :
– Principe de co-induction :

$$\frac{\Gamma \vdash \Pi^o x : F(\nu F).P}{\Gamma \vdash \Pi^o x : \nu F.P}\,Coinduc_o$$

Nous avons plus de règles algorithmiques que pour les types inductifs, car nous disposons de règles de création de valeurs avec contenu algorithmique :

– Coinduction "avec" contenu algorithmique :

$$\frac{\Gamma \vdash t : \Pi^\tau x : F(\nu F).B(x)}{\Gamma \vdash t : \Pi^\tau x : \nu F.B(x)}\,Coinduc_\tau$$

– Création de valeurs de type coinductif (1) :

$$\frac{\Gamma \vdash t : \forall X(X \Rightarrow F(X))}{\Gamma \vdash (\mathsf{rec}\ t) : \nu F}\,\nu_{creat1}$$

– Création de valeurs de type coinductif (2) :

$$\frac{\Gamma \vdash t : \forall X((A \Rightarrow X) \Rightarrow (A \Rightarrow F(X)))}{\Gamma \vdash (\mathsf{rec}\ t) : A \Rightarrow \nu F}\,\nu_{creat2}$$

Les règles de sous-typage des types co-inductifs sont les suivantes :

– Point fixe :

$$\frac{}{\Gamma \vdash (\nu F =_{\subset} F(\nu F))}\ \nu_{PF}$$

– Plus grand point fixe :

$$\frac{\Gamma \vdash A \subset F(A)}{\Gamma \vdash A \subset \nu X.F(X)}\ \subset_{\nu}$$

7.6.2 Exemple : listes et streams

Dans le même esprit que l'on a défini le type des listes dans la section précédente, on définit ici le type des streams qui peuvent être des listes (Stream_1), et celui des streams nécessairement infinis (Stream_2) :

$$\mathsf{Stream}_1 \quad = \quad \lambda A.\nu X.(\ \mathsf{Nil} \mid \mathsf{Cons\ of}\ A \times X\ )$$

$$\mathsf{Stream}_2 \quad = \quad \lambda A.\nu X.(\ \mathsf{Cons\ of}\ A \times X\ )$$

On a, par exemple, pour tout type A :

$$\vdash \mathsf{List}(A) \subset \mathsf{Stream}_1(A)$$

et

$$\vdash \mathsf{Stream}_2(A) \subset \mathsf{Stream}_1(A)$$

Le stream des entiers peut être construit de la façon suivante :

$$\vdash (\mathsf{rec\ fun\ r} \rightarrow \mathsf{fun\ n} \rightarrow \mathsf{Cons[(n,\ r\ Succ[n])]} : \mathsf{Pos} \Rightarrow \mathsf{Stream}_2(\mathsf{Pos})$$

à l'aide de la règle de création des types coinductifs.

7.7 Enregistrements

7.7.1 Types d'enregistrements

Comme pour les types sommes, on ajoute une nouvelle sorte τ_R qui sert à construire des nouveaux types. Comme pour les types sommes, on se donne un ensemble dénombrable $\mathcal{L} = \{l_1; l_2; \ldots\}$ de labels muni d'un prédicat d'égalité. Nous disposons de deux sortes de types enregistrements :
– les types enregistrements usuels (de la forme $\{l_1 : A; l_2 : B\}$)
– les types enregistrements forts (de la forme $\{l_1 : A; l_2 : B\}_S$).

Les règles de construction des types enregistrements et les règles d'absence des labels dans les types enregistrements sont :

$$\frac{\Gamma \vdash A : \tau \qquad l \in \mathcal{L}}{\Gamma \vdash \{l : A\} : \tau_R} R_0$$

$$\frac{\Gamma \vdash \{T\} : \tau_R \quad \Gamma \vdash A : \tau \quad l \in \mathcal{L} \quad l \notin T}{\Gamma \vdash \{l : A \quad ; \quad T\} : \tau_R} R_;$$

$$\frac{\Gamma \vdash A : \tau \qquad l_1 \in \mathcal{L} \quad l_2 \in \mathcal{L} \quad l_1 \neq l_2}{\Gamma \vdash l_1 \notin \{l_2 : A\}} \notin_{R_0}$$

$$\frac{\Gamma \vdash A : \tau \quad l_1 \in \mathcal{L} \quad l_2 \in \mathcal{L} \quad l_1 \neq l_2 \quad l_1 \notin \{T\}}{\Gamma \vdash l_1 \notin \{l_2 : A \quad ; \quad T\}} \notin_{R_;}$$

$$\frac{\Gamma \vdash \{T\} : \tau_R}{\Gamma \vdash \{T\}_S : \tau_R} \{\}_S$$

$$\frac{\Gamma \vdash \{T\} : \tau_R}{\Gamma \vdash \{T\} : \tau} \tau_R 2 \tau$$

Le lecteur peut se demander à quoi vont servir les enregistrements forts. Leur seule raison d'être est qu'avec le sous-typage, certains champs peuvent ne pas apparaître dans le type. Comme on dispose d'un type pour caractériser les types normalisables (voir la dernière section) , il paraît naturel de pouvoir certifier qu'un enregistrement ne contient que des objets de types déclarés dans son type : c'est le rôle de $\{T\}_S$.

7.7.2 Sous-typage

Les règles de sous-typage des enregistrements correspondent à l'idée qu'on peut avoir que ce sont des programmes qui attendent qu'on leur donne un constructeur pour fournir une valeur du type correspondant. Ainsi, oublier un champs n'altère pas le type du programme.

Il y a deux règles de sous-typage structurel, une règle de permutation et une règle d'enregistrement forcé qui sert à remettre un enregistrement dans son type le plus général :

– Oubli d'un champs :

$$\frac{\Gamma \vdash \{T_1 \quad ; \quad l : A \quad ; \quad T_2\} : \tau}{\Gamma \vdash \{T_1 \quad ; \quad l : A \quad ; \quad T_2\} \subset \{T_1 \quad ; \quad T_2\}} \{\}^{-}_{\subset}$$

– Sous-typage du type d'un champs :

$$\frac{\Gamma \vdash \{T_1 \quad ; \quad l : A \quad ; \quad T_2\} : \tau_R \qquad \Gamma \vdash A \subset A'}{\Gamma \vdash \{T_1 \quad ; \quad l : A \quad ; \quad T_2\} \subset \{T_1 \quad ; \quad l : A' \quad ; \quad T_2\}} \{\}^{\subset}_{\subset}$$

– Permutation des champs :

$$\frac{\Gamma \vdash \{l_1 : A_1; \ldots; l_n : A_n\} : \tau_R \quad \sigma \text{ permutation de } \{1; \ldots; n\}}{\Gamma \vdash \{l_1 : A_1; \ldots; l_n : A_n\} =_{\subset} \{l_{\sigma 1} : A_{\sigma 1}; \ldots; l_{\sigma n} : A_{\sigma n}\}} \{\}^{\sigma}_{\subset}$$

– Enregistrements forcés :

$$\frac{\Gamma \vdash \{T\} : \tau_R}{\Gamma \vdash \{T\}_S \subset \{T\}} \{\}^{S}_{\subset}$$

Exemple

On voit ici comment la règle d'oubli d'un champs peut masquer dans un type enregistrement la présence d'un élément non normalisable. Prenons le type suivant :

$$\{\mathsf{depart} : \mathsf{Pos} \,; \mathsf{suite} : \mathsf{Stream}_2(\mathsf{Pos})\}$$

représentant par exemple un champs contenant un entier "départ" et la suite des entiers commençant par ce dernier. On a :

$$\{\mathsf{depart} : \mathsf{Pos} \,; \mathsf{suite} : \mathsf{Stream}_2(\mathsf{Pos})\} \subset \{\mathsf{depart} : \mathsf{Pos}\}$$

ce qui entraîne une perte d'information non négligeable quand à la terminaison d'un habitant de ce type ! On ne rencontrera pas ce problème avec les enregistrements forts, qui représentent exactement le type voulu.

7.7.3 Règles de typage

Le langage s'enrichit des constructions suivantes :

$$\begin{array}{rcl} \mathcal{P} & = & \ldots \quad | \quad < \mathcal{R} > \quad | \quad \mathcal{P}.\mathcal{L} \quad | \quad [\mathcal{L} = \mathcal{P}] + \mathcal{P} \quad | \quad \mathcal{P} \text{ as } l_{\mathcal{L}} \\ \mathcal{R} & = & \mathcal{L} = \mathcal{P} \quad | \quad \mathcal{R}; \mathcal{L} = \mathcal{P} \\ l_{\mathcal{L}} & = & \mathcal{L} \quad | \quad \mathcal{L}, l_{\mathcal{L}} \end{array}$$

$\mathcal{R}$ correspond aux "records", $p.l$ à la projection, et la notation "as" sert à récupérer certains champs d'un record dans un autre record.
Les règles de typage sont assez économiques, car il n'y a que deux règles de base :

– Construction d'un enregistrement :

$$\frac{\Gamma \vdash \{l_1 : A_1; \ldots; l_n : A_n\} : \tau_R \qquad \Gamma \vdash p_1 : A_1 \quad \ldots \quad \Gamma \vdash p_n : A_n}{\Gamma \vdash < l_1 = p_1; \ldots; l_n = p_n > : \{l_1 : A_1; \ldots; l_n : A_n\}} \{\}_I$$

– Projection d'un champs

$$\frac{\Gamma \vdash p : \{l : A\}}{\Gamma \vdash p.l : A} \{\}_E$$

Les autres règles concernent les constructions moins habituelles :

– Ajout d'un champs :

$$\frac{\Gamma \vdash \{l : A; L\} : \tau_R \qquad \Gamma \vdash p : \{L\} \qquad \Gamma \vdash q : A}{\Gamma \vdash [l = q] + p : \{l : A; L\}} \{\}_+$$

– Sélection forcée :

$$\frac{\Gamma \vdash p : \{l_1 : A_1; \ldots; l_n : A_n\}}{\Gamma \vdash p \text{ as } l_1, \ldots, l_n : \{l_1 : A_1; \ldots; l_n : A_n\}_S} \{\}_S$$

7.7.4 Exemples

Entiers et suites d'entiers

On reprend le type donné précédemment. A l'aide des exemples vus dans la section des types coinductifs, on peut supposé construit un programme p qui correspond à la description souhaitée.
La construction as permet de récupérer uniquement le champs qui termine :

$$\frac{\vdash p : \{\mathsf{depart} : \mathsf{Pos}; \mathsf{suite} : \mathsf{Stream}_2(\mathsf{Pos})\} \quad \vdash \{\mathsf{depart} : \mathsf{Pos}\ ; \mathsf{suite} : \mathsf{Stream}_2(\mathsf{Pos})\} \subset \{\mathsf{depart} : \mathsf{Pos}\}}{\dfrac{\vdash p : \{\mathsf{depart} : \mathsf{Pos}\}}{\vdash p\ \mathsf{as\ depart}\ : \{\mathsf{depart} : \mathsf{Pos}\}_S}}$$

Arbres binaires

On définit les types des arbres binaires étiquetés et des arbres binaires taggés par :

$$\mathsf{Btree} \quad = \quad \mu X.(\ \mathsf{Leaf} \mid \mathsf{Node\ of}\ \{\ \mathsf{g} : X\ ;\ \mathsf{e} : A ;\ \mathsf{d} : X\ \}\ )$$

$$\mathsf{TagBtree} \quad = \quad \mu X.(\ \mathsf{Leaf} \mid \mathsf{Node\ of}\ \{\ \mathsf{g} : X\ ;\ \mathsf{e} : A ;\ \mathsf{d} : X\ ;\ \mathsf{h} : \mathsf{Pos}\}\ )$$

On a :

$$\vdash \mathsf{TagBtree} \subset \mathsf{Btree}$$

par

$$\vdash\ \mathsf{Leaf} \mid \mathsf{Node\ of}\ \{\ \mathsf{g} : \mathsf{Btree}\ ;\ \mathsf{e} : A ;\ \mathsf{d} : \mathsf{Btree}\ ;\ \mathsf{h} : \mathsf{Pos}\}\ \subset \mathsf{Btree}$$

Supposant définie une fonction hauteur sur les arbres :

$$\vdash \mathsf{height} : \mathsf{Btree} \Rightarrow^{\tau} \mathsf{Pos}$$

elle est par sous-typage également définie sur les arbres binaires taggés, et on peut définir le type des arbres binaires taggés dont la marque h est la hauteur :

$$\mathsf{HTagBtree} = \{t : \mathsf{TagBtree}; (\mathsf{height}\ t) = t.\mathsf{h}\}$$

7.8 Abstractions

On définit une nouvelle famille d'opérateurs d'abstraction pour les termes :

$$\mathsf{Some}^{s_1,\ldots,s_n} : (s_1,\ldots,s_n \to \tau) \to \tau$$

On définit également des projections :

$$\frac{\Gamma \vdash p : \pi \qquad \Gamma \vdash T : s_1,\ldots,s_n \to \tau}{\Gamma \vdash p :^{s_1,\ldots,s_n} T.i : s_i}\, Dot$$

(avec $s_1,\ldots,s_n$ n sortes et $1 \leq i \leq n$)
L'opérateur Some sert à créer un type dont certaines composantes sont abstraites, et les projections $.^{s_1,\ldots,s_n}$ à "projeter" les composantes abstraites relativement à un programme.

Notation 16 *On notera* Some $x_1,\ldots,x_n.T$ *pour* Some $\lambda x_1 \ldots \lambda x_n T$.

7.8.1 Règles de l'abstraction

Les règles correspondent, comme pour les types sommes, à l'intuition de la réunion :

– Sous-typage droit(un type concret est un sous-type d'un type abstrait) :

$$\frac{\Gamma \vdash t_1 : s_1 \qquad \ldots \qquad \Gamma \vdash t_n : s_n}{\vdash T[x_1 := t_1 \ldots x_n := t_n] \subset \mathsf{Some}^{s_1,\ldots,s_n} x_1,\ldots,x_n.T}\, \subset^r_{\mathsf{Some}}$$

– Sous-typage gauche :

$$\frac{\Gamma \vdash B : \tau \qquad \Gamma, x_1 : s_1,\ldots,x_n : s_n \vdash T \subset B}{\Gamma \vdash \mathsf{Some}^{s_1,\ldots,s_n} x_1,\ldots,x_n.T \subset B}\, \subset^l_{\mathsf{Some}}$$

– Projection :

$$\frac{\Gamma \vdash p : \mathsf{Some}^{s_1,\ldots,s_n}(T)}{\Gamma \vdash p : T((p :^{s_1,\ldots,s_n} T).1) \ldots ((p :^{s_1,\ldots,s_n} T).n)}\, \mathsf{Some}_{proj}$$

Exemple : abstraction dans un enregistrement

On peut, par exemple, abstraire sur un type paramétré dans un enregistrement :

$$< \mathsf{head} = \mathsf{u}\,;\, \mathsf{tail} = \mathsf{v} >:$$
$$\vdash$$
$$\mathsf{Some}^{\tau\to\tau}\ list.\{\quad \mathsf{head} : \forall A(list(A) \Rightarrow A) \quad ; \quad \mathsf{tail} : \forall A(list(A) \Rightarrow list(A)) \quad \}$$

En laissant au lecteur le soin de compléter selon son imagination (il y a un piège).

7.8.2 Programmes abstraits

Dans le but de permettre la manipulation de programmes dont le type parle d'eux-mêmes, nous disposons déjà du schéma de compréhension. Mais celui-ci n'est pas assez souple pour prendre en compte une forme d'auto-référencement particulière, que nous illustrons par un exemple :
Supposons que nous ayons dérivé $\vdash p_a : T_a$, et $\vdash p_b : T_b(p_a)$. Nous pouvons en déduire $\vdash < a = p_a; b = p_b >: \{a : T_a; b : T_b(p_a)\}$, mais nous souhaiterions pouvoir écrire un type de la forme $\{a : T_a; b : T_b(a)\}$. C'est ce que va nous permettre le nouvel opérateur d'abstraction sur les programmes :

$$\frac{\Gamma \vdash T : \pi \to \tau}{\Gamma \vdash \mathsf{Self}\ T : \tau}$$

Cet opérateur étant en fait un lieur, on adoptera la même convention d'écriture que pour les autres lieurs :

Notation 17 *On notera* $\mathsf{Self}\ s.T(s)$ *pour* $\mathsf{Self}\ (\lambda s.T)$.

On dispose de deux règles de typage sur les programmes abstraits :

– Introduction de l'abstraction de programme :

$$\frac{\Gamma \vdash p : T(p)}{\vdash \Gamma \vdash p : \mathsf{Self}\ s.T(s)}\ \mathsf{Self}_I$$

– Élimination de l'abstraction de programmes :

$$\frac{\Gamma \vdash p : \mathsf{Self}\ s.T(s)}{\Gamma \vdash p : T(p)}\ \mathsf{Self}_E$$

Et d'une règle de sous-typage :

– Sous-typage de Self

$$\frac{\Gamma, x : \mathsf{Self}t.A(t) \vdash A(x) \subset B(x)}{\vdash \Gamma \vdash \mathsf{Self}\ s.A(s) \subset \mathsf{Self}\ s.B(s)}\ \mathsf{Self}_\subset$$

Ainsi, dans l'exemple présenté en introduction, on noterait :

$$\vdash < a = p_a; b = p_b >: \mathsf{Self}\ s.\{a : T_a; b : T_b(s.a)\}$$

7.9 Terminaison

Nous donnons dans cette section quelques moyens de déterminer si un type ne contient que des programmes qui terminent (en évaluation gauche).
On ajoute au langage des termes une constante

$$NORM : \tau$$

Les règles suivantes permettent de déterminer si un type (ou un programme) termine :

– Les constructeurs sont normalisables :

$$\frac{C \in \mathcal{C}}{\vdash C \subset NORM}$$

– Les constructeurs -appliqués- à un argument normalisable sont normalisables :

$$\frac{C \in \mathcal{C}}{\vdash \forall A(A \subset NORM \quad \Rightarrow \quad C \ \mathsf{of}\ A \subset NORM)}$$

– Les sommes de normalisables sont normalisables :

$$\frac{\Gamma \vdash A \in NORM \qquad \Gamma \vdash B \in NORM \qquad \Gamma \vdash A \mid B : \tau_S}{\Gamma \vdash A \mid B \subset NORM}$$

– Paires normalisables :

$$\frac{}{\vdash \forall A, B(A \subset NORM \Rightarrow B \subset NORM \Rightarrow (A \times B) \subset NORM)}$$

– Enregistrements forts normalisables :

$$\frac{\Gamma \vdash A_1 \subset NORM \qquad \Gamma \vdash \{l_1 : A_1; \ldots l_n : A_n\}_S : \tau_R}{\Gamma \vdash \{l_1 : A_1; \ldots l_n : A_n\}_S \subset NORM}$$

– Normalisabilité des types inductifs (cette règle est en fait dérivable) :

$$\frac{\Gamma \vdash F(NORM) \subset NORM}{\Gamma \vdash \mu X F(X) \subset NORM}$$

Exemples de types normalisants

En reprenant les exemples précédents, nous pouvons montrer :

$$\vdash B_F \subset NORM \qquad\qquad \vdash \mathsf{Rel} \subset NORM$$

$$\vdash A \subset NORM \Rightarrow \mathsf{Btree}(A)_S \subset NORM$$

7.10 Conclusion

Le langage que nous venons de présenter dans ce chapitre comporte, outre des traits de programmation classiques, un langage de spécifications très riche.
Nous allons exploiter ses possibilités dans le chapitre suivant, où nous définirons un langage de module permettant de décrire aisément des programmes construits de façon incrémentale (on définit un programme a, puis un programme b qui dépend de a, puis un programme c qui dépend de a et b, etc ...) avec des types qui comporteront non seulement le typage de chaque composante (a, b, c, ...) du module, mais aussi des propriétés portant sur ces composantes, ainsi que des définitions de types, types paramétrés, etc ...
Il nous reste à vérifier, afin que ce que l'on a raconté fasse sens, les fondements théoriques de ce langage, ce qui sera fait dans le chapitre 9 où nous énoncerons et prouverons les énoncés de correction.

Chapitre 8

Modules

8.1 Introduction

L'objectif ce ce chapitre est de définir *à l'intérieur* de notre langage un système adapté à la manipulation de modules, vus comme des programmes typés qui ont pour vocation de permettre de structurer, d'organiser des développements. Le but d'un langage de modules est, à notre sens, le suivant : il doit permettre d'écrire des programmes (que nous appellerons *structures*) qui comportent une série de "définitions" de programmes, et des types correspondants (que nous appellerons *signatures*) qui comportent également des "définitions" et des propriétés portant sur les programmes définis dans les structures, les "définitions" et énoncés étant écrits de façon *incrémentale*, c'est à dire dépendant les uns des ordres dans l'ordre de lecture, et leur réutilisation.
Par exemple, on peut songer à un module qui définirait les entiers naturels et les opérations usuelles que l'on fait dessus, avec les spécifications correspondantes. La structure contiendrait les implémentations des algorithmes usuels (addition, multiplication, division, ...) et la signature associée contiendrait, outre le type des opérations, une série d'énoncés décrivant les propriétés usuelles (par exemple les propriétés algébriques des fonctions, comme l'associativité, la distributivité, ...), dont on pourra avoir besoin par la suite pour construire et prouver d'autres algorithmes plus sophistiqués.

Notre présentation s'articule ainsi : nous présentons d'abord les grammaires de programmes et de types qui étendent le langage de base, puis les règles de construction de modules, et les règles de destruction (projection), et enfin les règles de sous-typage.
Les règles sont fortement inspirées de celles présentées par Xavier Leroy dans par exemple [35], [36], et dans les notes de cours de François Pottier [46], sauf que :
- Pour éviter la capture de noms[1], nous utilisons la même convention que Judicaël Courant [18], inspirée de Mark Lillibridge et Robert Harper [25] qui consiste à distinguer nom de variable liée et label à l'aide de la construction "as".
- nous n'utilisons pas la notion de chemin, qui aurait sa place dans un développement plus

[1] comme par exemple dans ((fun x → let a = ... ;let b= x.n) a)

important donné aux modules en vue d'une implémentation, car permettant d'appeler les types abstraits d'une façon plus simple, mais qui n'apporterait rien ici du point de vue de la théorie.
- nous ajoutons des spécifications (des formules sans contenu algorithmique destinées à exprimer les propriétés des programmes) et des définitions de toutes sortes (dans le sens des sortes $\tau, o \ldots$) dans les signatures.

8.2 Grammaire des structures et des signatures

8.2.1 Structures

Nous étendons le langage de programmation avec les structures :

$$\begin{array}{lcl} \mathcal{P} & = & \ldots \mid \mathsf{Struct} \quad \mathcal{S} \\ \mathcal{S} & = & \mathsf{endSruct} \mid \mathsf{let}\ \mathcal{V}\ \mathsf{as}\ \mathcal{L} = \mathcal{P}\,!\quad \mathcal{S} \end{array}$$

Les occurrences libres de la variable x dans S sont liées dans $\mathsf{let}\ x\ \mathsf{as}\ l = p\,!\ S$.
La notion de substitution correcte est étendue naturellement à cette nouvelle construction.

Notation 18 *Pour alléger les écritures, nous utiliserons le sucre syntaxique suivant :*

$$\mathit{let}\ l = p\ !\ S(l) \quad \mathit{pour} \quad \mathit{let}\ x\ \mathit{as}\ l = p\ !\ S(x)$$

8.2.2 Signatures

On se donne :
- Un ensemble dénombrable $\mathcal{N}$ de noms, muni d'un prédicat d'égalité. On notera les noms n, m, ...
- Un ensemble fini de sortes Σ qui contient au moins τ et $\tau \to \tau$.

Les règles de formation des signatures sont :

- Signature vide :

$$\frac{}{\Gamma \vdash \mathsf{endSig} : \tau_{sig}}\ sig_{Ax}$$

- Définition concrète d'un objet de sorte s :

$$\frac{\Gamma \vdash A : s \qquad \Gamma, t : s \vdash S : \tau_{sig} \qquad s \in \Sigma \qquad n \in \mathcal{N} \qquad n \notin S}{\Gamma \vdash \mathsf{def}_s\ t\ \mathsf{as}\ n = A\,!\ S : \tau_{sig}}\ sig_{Def}$$

- Définition abstraite d'un objet de sorte s :

$$\frac{\Gamma, t : s \vdash S : \tau_{sig} \qquad s \in \Sigma \qquad n \in \mathcal{N} \qquad n \notin S}{\Gamma \vdash \mathsf{def}_s\ t\ \mathsf{as}\ n\,!\ S : \tau_{sig}}\ sig_{ADef}$$

– Définition de type associé à une valeur :

$$\frac{\Gamma \vdash A : \tau \qquad \Gamma, x : A \vdash S : \tau_{sig} \qquad l \in \mathcal{L} \qquad l \notin S}{\Gamma \vdash \ \mathsf{val}\ x\ \mathsf{as}\ l : A\,!\,S : \tau_{sig}}\ sig_{Prog}$$

– Propriété :

$$\frac{\Gamma \vdash P : o \quad \Gamma \vdash S : \tau_{sig}}{\vdash \ \mathsf{fact}\ P\,!\,S : \tau_{sig}}\ sig_{Fact}$$

– Clôture de la signature :

$$\frac{\Gamma \vdash S : \tau_{sig}}{\Gamma \vdash \mathsf{Sig}\ S : \tau}\ \tau_{sig} <: \tau$$

Les dérivations de la forme $l \notin S$ ou $n \notin S$ sont analogues aux dérivations d'absence de champs dans les enregistrements :

$$\frac{l \in \mathcal{L}}{\vdash l \notin \mathsf{endSig}}\ l^{\notin}_{Ax} \qquad \frac{\vdash l \notin S \quad \vdash l' \in \mathcal{L} \quad l \neq l'}{\vdash l \notin \ \mathsf{val}\ x\ \mathsf{as}\ l' : A\,!\,S}\ l^{\notin}_{Prog}$$

$$\frac{\vdash l \notin S}{\vdash l \notin \ \mathsf{fact}\ P\,!\,S}\ l^{\notin}_{Fact} \qquad \frac{\vdash l \notin S}{\vdash l \notin \ \mathsf{def}_s\ t\ \mathsf{as}\ n\ (\,=A\,)\,!\,S}\ l^{\notin}_{Def}$$

$$\frac{n \in \mathcal{N}}{\vdash n \notin \mathsf{endSig}}\ n^{\notin}_{Ax} \qquad \frac{\vdash n \notin S}{\vdash n \notin \ \mathsf{val}\ x\ \mathsf{as}\ l' : A\,!\,S}\ n^{\notin}_{Prog}$$

$$\frac{\vdash n \notin S}{\vdash n \notin \ \mathsf{fact}\ P\,!\,S}\ n^{\notin}_{Fact} \qquad \frac{\vdash n \notin S \quad n' \in \mathcal{N} \quad n \neq n'}{\vdash n \notin \ \mathsf{def}_s\ t\ \mathsf{as}\ n'(\,=A\,)\,!\,S}\ n^{\notin}_{Def}$$

On voit ainsi qu'une signature peut comporter :
– des définitions (abstraites ou non) de types ou de types paramétrés :

$$\mathsf{def}_\tau\ x\ \mathsf{as}\ n\ (\,=A\,) \quad ou \quad \mathsf{def}_{\tau\to\tau}\ x\ \mathsf{as}\ n\ (\,=A\,)$$

– des déclarations de programmes : $\mathsf{val}\ x\ \mathsf{as}\ l : A$
– des énoncés de propositions : $\mathsf{fact}\ P$.

8.2.3 Liaisons et convention d'écriture

Les constructeurs def_s et val sont des lieurs, ce qui se définit formellement par :

Définition 56 (liaisons de noms et α-conversion)
– *Les occurrences libres de t dans S sont liées dans $\mathsf{def}_s\ t\ \mathsf{as}\ n = A\,!\,S$.*
– *Les occurrences libres de x dans S sont liées dans $\mathsf{val}\ x\ \mathsf{as}\ l : A\,!\,S$.*

L'alpha-conversion des variables nommées est identique à l'alpha-conversion usuelle.

Tout comme pour les structures, on adopte la convention d'écriture suivante :

Notation 19 *Lorsque cela ne comportera pas d'ambiguïté, on notera :*

$$\mathsf{val}\ l : A\ !\ S(l) \quad pour \quad \mathsf{val}\ x\ \mathsf{as}\ l : A\ !\ S(x)$$

et

$$\mathsf{def}\ n\ (= A\ )\ !\ S(n) \quad pour \quad \mathsf{def}\ x\ \mathsf{as}\ n\ (= A\ )\ !\ S(x)$$

8.3 Typage des structures : construction

Les règles de typage pour la construction des structures sont les suivantes (on ajoute des parenthèses pour améliorer la lisibilité) :

– Structure "vide" :

$$\frac{WF(\Gamma)}{\Gamma \vdash \quad \mathsf{endStruct} :_M \mathsf{endSig}}\, M_{end}$$

– Ajout d'un champs :

$$\frac{\Gamma, x : A, x = p \vdash m :_M S \qquad \Gamma \vdash p : A \qquad l \notin S}{\Gamma \vdash \quad (\mathsf{let}\ x\ \mathsf{as}\ l = p\,!\ m) :_M (\mathsf{val}\ x\ \mathsf{as}\ l : A\,!\ S)}\, M_{val}$$

– Ajout d'un théorème :

$$\frac{\Gamma, P \vdash m :_M S \qquad \Gamma \vdash P}{\Gamma \vdash m :_M (\mathsf{fact}\ P\,!\ S)}\, M_{fact}$$

– Ajout d'une définition :

$$\frac{\Gamma, t : s, t = A \vdash m :_M S \qquad \Gamma \vdash A : s \qquad s \in \Sigma \qquad n \notin S}{\Gamma \vdash m :_M (\mathsf{def}_s\ t\ \mathsf{as}\ n = A\,!\ S)}\, M_{Term}$$

– Clôture :

$$\frac{\Gamma \vdash m :_M S}{\Gamma \vdash \mathsf{Struct}\ m : \mathsf{Sig}\ S}\, M_{struct}$$

La règle de clôture est en fait la seule qui introduise un terme, les autres servant à le bâtir de façon incrémentale. Dans les règles M_val et M_fact, les prémisses $l \notin S$ et $n \notin S$ servent à assurer que le nom n'est pas déjà défini dans la suite de la signature.
On remarque que nos règles permettent également de définir des structures vides associées à des signatures composées uniquement de théorèmes et de définitions (on peut par exemple songer à un module qui contiendrait une théorie et ses théorèmes principaux ...).

Exemple : un module pour les booléens

Supposons que l'on veuille construire un module pour les booléens, ne pas permettre d'utiliser la définition du type bool en dehors du module, et spécifier le comportement des opérateurs usuels :

Struct			Sig	
			def_τ bool = TRUE | FALSE	!
let true = TRUE	!		val true : bool	!
let false = FALSE	!		val false : bool	!
let ifthenelse = ...	!	:	val ifthenelse : $\forall A(\text{bool} \Rightarrow A \Rightarrow A \Rightarrow A)$	!
...	!		...	!
			fact $\forall A \Pi^o x, y : A$ ifthenelse true x y = x	!
			...	
endStruct			endSig	

Exemple : des groupes paramétrés

Nous donnons ici comme premier exemple la construction des groupes $\mathcal{Z}/n\mathcal{Z}$.
On commence par définir le type paramétré d'un groupe :

$group(E) =$

Sig	
val e : E	!
val plus : $E \Rightarrow E \Rightarrow E$	!
val op : $E \Rightarrow E$	!
fact $\Pi^o x : E.(\text{plus } x \text{ e}) = x$	!
fact $\Pi^o x : E.(\text{plus } x \text{ (op } x)) = \text{e}$	!
fact $\Pi^o x : E.\Pi^o y : E.\Pi^o z : E$ (plus (plus x y) z) = (plus x (plus y z))	!
endSig	

On suppose données les opérations arithmétiques élémentaires sur l'ensemble des entiers naturels POS tels que définis page 128, et parmi celles-ci la relation d'ordre et la fonction modulo :

$$\begin{array}{ll} \text{leq} & : POS \Rightarrow POS \Rightarrow BOOL \\ \text{modulo} & : POS \Rightarrow POS \Rightarrow POS \end{array}$$

qui nous permet de définir le prédicat $\leq$ par :

$$x \leq y \quad = \quad (\text{leq } x\ y) = \text{TRUE}$$

et le type intervalle :

$$[a; b] = \{x : POS; a \leq x \wedge x \leq b\}$$

On définit le programme suivant :

$$quot = \mathsf{fun}\ n \rightarrow \left(\begin{array}{ll} \mathsf{Struct} & \\ \quad \mathsf{let\ e} = Z & ! \\ \quad \mathsf{let\ plus} = \mathsf{fun\ x} \rightarrow \mathsf{fun\ y} \rightarrow \mathsf{modulo\ (x+y)}\ n & ! \\ \quad \mathsf{let\ op} = \mathsf{fun\ x} \rightarrow \mathsf{modulo}\ ((n\text{-}1)\times \mathsf{x})\ n & ! \\ \mathsf{endStruct} & \end{array} \right)$$

Et on infère, en posant $SPOS = \{r : POS; r > Z\}$:

$$\vdash quot : \Pi^{r} n : SPOS.group([Z; (n-1)])$$

On voit un intérêt de disposer de modules de premier ordre : on peut les faire dépendre d'autres programmes "ordinaires".

8.4 Typage des structures : destruction

8.4.1 Définitions projetées et substitutions

On introduit un nouveau constructeur de type, pour permettre la projection des noms définis dans les modules :

Définition 57 (Définition projetée) *La règle de sortage d'une définition projetée* $(m :_s S).n$ *est :*

$$\frac{\Gamma \vdash m : \mathsf{Sig}\ S \qquad S = S_1\ !\ \mathsf{def}_s\ x\ \mathsf{as}\ n\ (= A\ )\ !\ S_2}{\Gamma \vdash (m :_s S).n : s}\ M_{struct}$$

Dans l'optique de simplifier les notations, une définition projetée $(m :_s S).n$ pouvant être une expression extrêmement longue, on utilise la convention suivante :

Notation 20 *Lorsque cela ne présente pas d'ambiguïté, on notera :*

$$m.n \qquad pour \qquad (m :_s S).n$$

On utilisera cette notation dans deux cas :

– $m : \mathsf{Sig}\ S$ est dans le contexte Γ.
– $m : \mathsf{Sig}\ S$ est la prémisse d'une règle.

Le premier cas, généralisé aux modules apparaissant à l'intérieur de modules définis dans le contexte, nous permettrait de définir des notions de chemin. Mais, comme nous l'avons précisé dans l'introduction de ce chapitre, nous n'abordons pas ce problème en détail.
Pour utiliser les signatures, on a besoin de la notion suivante de substitution :

Définition 58 (Substitution liée à une signature) *Si*

$$\Gamma \vdash m : \mathsf{Sig}\ S$$

avec

$$S = S_1\ !\ S_2$$

on définit la substitution :

$$\{z \leftarrow m.z \mid z \textit{ lié dans } S_1\}$$

par :

$$\begin{aligned}
\{z \leftarrow m.z \mid z \textit{ lié dans } \emptyset\} &= Id \\
\{z \leftarrow m.z \mid z \textit{ lié dans } \mathsf{val}\ x\ \mathsf{as}\ l : A\ !S'\} &= \{x \leftarrow m.l\} o \{z \leftarrow m.z \mid z \textit{ lié dans } S'\} \\
\{z \leftarrow m.z \mid z \textit{ lié dans } \mathsf{def}_s\ x\ \mathsf{as}\ n = A\ !S'\} &= \{x \leftarrow (m :_s S).n\} o \{z \leftarrow m.z \mid z \textit{ lié dans } S'\}
\end{aligned}$$

8.4.2 Règles de destruction

Nos modules comportant essentiellement trois types de composantes, nous définissons trois types de projections :

– Projection d'une définition :

$$\frac{\Gamma \vdash m : \mathsf{Sig}\ S \qquad S = S_1\ !\ \mathsf{def}_s\ x\ \mathsf{as}\ n = A\ !\ S_2}{\Gamma \vdash m.n = A\{z \leftarrow m.z \mid z \text{ lié dans } S_1\}}\ Proj_{def}$$

– Projection d'une composante :

$$\frac{\Gamma \vdash m : \mathsf{Sig}\ S \qquad S = S_1\ !\ \mathsf{val}\ x\ \mathsf{as}\ l : A\ !\ S_2}{\Gamma \vdash m.l : A\{z \leftarrow m.z \mid z \text{ lié dans } S_1\}}\ Proj_{val}$$

– Projection d'un fait :

$$\frac{\Gamma \vdash m : \mathsf{Sig}\ S \qquad S = S_1\ !\ \mathsf{fact}\ P\ !\ S_2}{\Gamma \vdash P\{z \leftarrow m.z \mid z \text{ lié dans } S_1\}}\ Proj_{fact}$$

Exemples

– Nous commençons par un exemple très simple avec des types abstraits (inspiré de [36]) :

$$\frac{\Gamma \vdash m : \mathsf{Sig}\ \mathsf{def}_\tau\ \mathsf{t}\ !\ \mathsf{def}_\tau\ \mathsf{u}\ !\ \mathsf{def}_\tau\ \mathsf{v} = \mathsf{t} \times \mathsf{u}\ !\ \mathsf{endSig}}{\Gamma \vdash m.\mathsf{v} = m.\mathsf{t} \times m.\mathsf{u}}$$

Avec les notations abrégées convenues.

– En reprenant le type $group(E)$ défini précédemment, et en utilisant les règles de projection ci-dessus, on obtient que le programme :

```
fun g → fun h →
 Struct
  let e = ( h.e , g.e )                                                    !
  let plus = fun x → fun y → (g.plus(fst x)(fst y), h.plus(snd x)(snd y))   !
  let op = fun x → ( g.op (fst x) , h.op (snd x))                          !
 endStruct
```

reçoit le type :

$$\forall E, F(group(E) \Rightarrow group(F) \Rightarrow group(E \times F))$$

8.5 Sous-typage des signatures

Il y a cinq règles de sous-typage pour les modules :

– Sous-typage de champs :

$$\frac{\Gamma \vdash A \subset B}{\Gamma \vdash \quad (\mathsf{Sig}\ S_1\ \mathsf{val}\ x\ \mathsf{as}\ l : A\,!\ S_2) \quad \subset \quad (\mathsf{Sig}\ S_1\ \mathsf{val}\ x\ \mathsf{as}\ l : B\,!\ S_2)}\ Sig^{\subset}_{\subset}$$

– Omission de champs :

$$\frac{\Gamma \vdash \mathsf{Sig}\ S_1 \quad S_2 : \tau_{sig}}{\Gamma \vdash \quad (\mathsf{Sig}\ S_1\ \mathsf{val}\ x\ \mathsf{as}\ l : A\,!\ S_2) \quad \subset \quad (\mathsf{Sig}\ S_1 \quad S_2)}\ Sig^{\subset}_{val}$$

– Omission de définition :

$$\frac{\Gamma \vdash \mathsf{Sig}\ S_1 \quad S_2 : \tau_{sig}}{\Gamma \vdash \quad (\mathsf{Sig}\ S_1\ \mathsf{def}_s\ t = A\,!\ S_2) \quad \subset \quad (\mathsf{Sig}\ S_1 \quad S_2)}\ Sig^{\subset}_{def}$$

– Abstraction de définition :

$$\frac{}{\Gamma \vdash \quad (\mathsf{Sig}\ S_1\ \mathsf{def}_s\ t = A\,!\ S_2) \quad \subset \quad (\mathsf{Sig}\ S_1\ \mathsf{def}_s\ t\,!\ S_2)}\ Sig^{\subset}_{adef}$$

– Omission de théorème :

$$\frac{\Gamma \quad \mathsf{Sig}\ S_1 \quad S_2 : \tau_{sig}}{\Gamma \vdash \quad (\mathsf{Sig}\ S_1\ \mathsf{fact}\ P\,!\ S_2) \quad \subset \quad (\mathsf{Sig}\ S_1 \quad S_2)}\ Sig^{\subset}_{fact}$$

On constate que les règles $Sig^{\subset}_{val}$ et $Sig^{\subset}_{fact}$ sont essentiellement de même nature :

- $Sig^{\subset}_{val}$ consiste à omettre un calcul intermédiaire, qui n'a pas d'importance dans le résultat final (x n'est pas libre dans la suite).
- $Sig^{\subset}_{fact}$ consiste à omettre une proposition intermédiaire, autrement dit, un lemme.

La règle $Sig^{\subset}_{adef}$ permet de passer d'une définition concrète à une définition abstraite : en effet, les règles de typage des structures ne permettent que de construire des signatures avec des définitions explicites, et donc pas de définitions abstraites.

Exemple

Si on reprend l'exemple de la section 6.3 de ce chapitre, avec les règles de sous-typage, on pourrait donner à la structure des booléens le type plus simple :

```
Sig
  def_τ bool        !
  val true : bool   !
  val false : bool  !
endSig
```

8.6 Conclusion

Nous avons vu dans ce chapitre comment construire et utiliser des modules. Notre approche permet de plus d'utiliser les structures et les signatures comme des programmes et des types ordinaires, ce qui permet une certaine souplesse, contrebalancée par le manque de concision des projections de définitions. Nous verrons dans le chapitre de conclusion comment pallier à ce défaut.
Le chapitre qui suit est consacré à la correction du langage de programmation étendu par les modules.

Chapitre 9

Correction du langage

Nous validons dans ce chapitre les règles des chapitres 7 et 8, en partie par traduction dans le système ST[1], et en partie directement pour ce qui concerne les règles d'évaluation, la traduction étant indifférente à la stratégie d'évaluation.

9.1 Méthode et résultats

9.1.1 Plongement superficiel/ profond

La traduction d'un langage dans une logique peut s'effectuer de deux manières (nous nous inspirons du polycopié [43] pour les définitions de la terminologie) :
- Par plongement profond (deep embedding), qui consiste à se donner dans la logique les types concrets définissant le langage, ainsi que les axiomes correspondants aux règles.
- Par plongement superficiel (shallow embedding), qui consiste à traduire les objets du langage dans la logique et à démontrer les propriétés correspondant aux règles.

La partie du langage vérifiée par traduction consiste en un plongement superficiel dans la logique du système ST étendu avec les axiomes de non trivialité, de description définie et d'antisymétrie ($ST_{\neg TRIV,\Delta,AS}$)

9.1.2 Résultats

un programme p sera traduit en un type $\widetilde{p}$, un type A en un type $\widetilde{A}$, et un contexte Γ en un contexte $\widetilde{\Gamma}$.
Nous montrons les trois propositions suivantes :
La première (prouvée dans les sections 2 et 3) exprime la correction de la traduction :

Proposition 94 *Pour tout programme p, pour tout type A :*

$$Si \vdash p : A \text{ alors } \vdash_{ST_{\neg TRIV,\Delta,AS}} \widetilde{p} \in \widetilde{A}$$

[1] une grande partie des validations ont été faites dans le logiciel PhoX, avec une simplification pour les modules, qui sera expliquée en note dans la section 2.

On rappelle que $t \in A$ signifie $\mathcal{S}(t) \wedge t \subset A$.
La seconde (prouvée dans la section 4 comme les suivantes), la traduction de l'évaluation, en effectuant un abus d'écriture consistant à utiliser la β-réduction gauche (β_l) sur les λ-termes avec les types :

Proposition 95 *Pour tous programmes p et p'*

$$Si\ p \succ p'\ alors\ \widetilde{p}\ \beta_l\ \widetilde{p'}$$

La troisième, la correction de l'évaluation (non-blocage) :

Proposition 96 *Si $\vdash p : A$, sans règles de types abstraits ou de modules, alors :*
– *Soit p est une valeur.*
– *Soit il existe p' tel que $p \succ p'$ et $\vdash p' : A$*

Cette dernière proposition sera en fait prouvée indépendamment du système ST, car rappelons-le ce dernier identifie tous les traduits de λ-termes $\beta\eta$ équivalents, ce qui interdit de montrer une propriété relative à $\succ$ (dans ST, il n'est pas interdit d'appliquer un entier à un autre entier...)
Le traduit du type $NORM$ étant le type des programmes normalisants tel que défini dans le chapitre 5, nous déduisons le corollaire suivant :

Corollaire 97 *Pour tout programme p :*

$$Si\ \vdash p : NORM,\ alors\ l'évaluation\ de\ p\ termine$$

preuve: Par la proposition précédente, le programme étant typé, son évaluation ne bloque pas. Par le théorème 62 page 78, l'évaluation du traduit termine (car il s'agit de l'évaluation gauche par la proposition ci-dessus), donc l'évaluation du programme termine. □
Nous montrons la correction par traduction du langage dans la section 1, des modules dans la section 2. La correction de l'évaluation est montrée dans la section 3.

9.2 Validation des règles du langage

La traduction des contextes, des programmes et des types est définie par induction mutuelle. On notera indifféremment $\widetilde{\Gamma}$, $\widetilde{p}$, $\widetilde{A}$, ...les traductions des contextes, programmes et types.

9.2.1 Traduction des contextes, des termes et des programmes

Nous commençons par présenter la traduction qui ne concerne pas les programmes, mais simplement les termes (propositions, types , ...).
La traduction des sortes est simplement la suivante :
– $\widetilde{\tau} = \tau$
– $\widetilde{o} = o$

– $\widetilde{o'} = o$
– $\widetilde{\pi} = \tau$
– $\widetilde{s_1 \to s_2} = \widetilde{s_1} \to \widetilde{s_2}$

Cela signifie qu'au niveau des sortes, un élément de la sorte "programme" (π) sera vu comme un type.

La traduction des contextes est la suivante :

– $\widetilde{\emptyset} = \emptyset$
– $\widetilde{\Gamma, x : \pi} = \widetilde{\Gamma}, x : \tau, \mathcal{S}(x)$
– $\widetilde{\Gamma, x : s} = \widetilde{\Gamma}, x : \widetilde{s},$
– $\widetilde{\Gamma, x : A} = \widetilde{\Gamma}, x : \tau, \mathcal{S}(x), x \subset \widetilde{A}$

Il faut ensuite traduire les dérivations de constructions de termes (page 100 et suivantes) :

– Les constantes $\forall$, $\Rightarrow$ et $\subset$ sont traduites à l'identique. L'égalité est traduite par l'égalité de Leibnitz, et la relation de réduction $\succ$ est traduite par $\subset$.
– La λ-abstraction et l'application sont traduites à l'identique.
– Les produits Π^o et Π^τ, et le schéma de compréhension sont traduits de la façon suivante :

$$\begin{aligned}
\widetilde{\Pi^\tau x : A.B(x)} &= \forall x.(\mathcal{S}(x) \twoheadrightarrow x \subset A \twoheadrightarrow (x \Rightarrow^\tau B(x))) \\
\widetilde{\Pi^o x : A.P(x)} &= \forall x.(\mathcal{S}(x) \Rightarrow x \subset A \Rightarrow^o P(x))) \\
\widetilde{\{x : A; P(x)\}} &= \cup_x (x \restriction (\mathcal{S}(x) \wedge (x \subset A) \wedge P(x)))
\end{aligned}$$

Les programmes du noyau fonctionnel sont traduits par leur contrepartie en types (tels que présentés page 62), les autres traductions seront présentées au fur et à mesure.

9.2.2 Validation des règles

Les règles sont validées par induction sur la structure de la dérivation, on montre :

– Si $\Gamma \vdash t : s$, alors $\widetilde{\Gamma} \vdash \widetilde{t} : \widetilde{s}$
– Si $\Gamma \vdash P$, alors $\widetilde{\Gamma} \vdash \widetilde{P}$
– Si $\Gamma \vdash p : A$, alors $\widetilde{\Gamma} \vdash \widetilde{p} \in \widetilde{A}$

On utilisera assez souvent le lemme suivant :

Lemme 98 (Singletonicité des programmes) *Si* $\Gamma \vdash p : A$, *alors* $\widetilde{\Gamma} \vdash \mathcal{S}(\widetilde{p})$

preuve: Immédiat par le théorème 47 page 69, en remarquant que dans $\widetilde{\Gamma}$, toutes les variables libres de p sont déclarées comme singletons. □

Noyau fonctionnel

– Ax^τ : c'est une simple introduction de la conjonction.
– Π_I^τ : On suppose que :

$$\widetilde{\Gamma}, x : \tau, \mathcal{S}(x), x \subset A \vdash \mathcal{S}(t) \wedge t \subset B$$

On en déduit :

$$\widetilde{\Gamma} \vdash \forall x(x \Rightarrow^{\tau} t) \subset \forall x(\mathcal{S}(x) \twoheadrightarrow x \subset A \twoheadrightarrow x \Rightarrow B)$$

et

$$\widetilde{\Gamma} \vdash \mathcal{S}(\forall x(x \Rightarrow^{\tau} t))$$

par le lemme 98.

– Π^{τ}_{E} : On a, par hypothèse d'induction :

$$\widetilde{\Gamma} \vdash \mathcal{S}(u) \wedge (u \subset \Pi^{\tau} x : A.B) \quad et \quad \widetilde{\Gamma} \vdash \mathcal{S}(v) \wedge (v \subset A)$$

On en déduit que :

$$\widetilde{\Gamma} \vdash u \subset (\mathcal{S}(v) \twoheadrightarrow (v \subset A) \twoheadrightarrow (v \Rightarrow B[x := v]))$$
$$et$$
$$\widetilde{\Gamma} \vdash (\mathcal{S}(v) \twoheadrightarrow (v \subset A) \twoheadrightarrow (v \Rightarrow B[x := v])) \subset (v \Rightarrow B[x := v]))$$

La conclusion est immédiate avec la proposition 28 page 52.

– Les règles logiques sont celles de $ST_{\neg TRIV}$, ainsi que les règles de sous-typage $\subset_{Ax}$, $\subset_{Refl}$ et $\subset_{Trans}$.

– $\subset_{\Pi^{\tau}}$: On suppose que :

$$\widetilde{\Gamma} \vdash A' \subset A \quad et \quad \widetilde{\Gamma}, x : \tau, \mathcal{S}(x), x \subset A' \vdash B(x) \subset B'(x)$$

On en déduit facilement que :

$$\widetilde{\Gamma}, x : \tau \vdash \Pi^{\tau} x : A.B(x) \subset (\mathcal{S}(x) \twoheadrightarrow (x \subset A') \twoheadrightarrow B'(x))$$

la conclusion est alors immédiate.

– $\subset_{SC-CR\subset}$ et $\subset_{SC-CR\Rightarrow^{o}}$ sont immédiates en utilisant le fait 91 page 105.

– SC_{I} : l'hypothèse d'induction donne :

$$\widetilde{\Gamma} \vdash \mathcal{S}(t) \wedge (t \subset A) \quad et \quad \widetilde{\Gamma} \vdash P(t)$$

et la conclusion est immédiate par définition de SC.

– SC^{o}_{E} et $SC\tau_{E}$ utilisent les mêmes propriétés que $\subset_{SC-CR\subset}$ et $\subset_{SC-CR\Rightarrow^{o}}$.

– $Lo_{\succ}$: par le lemme 98, on a $\widetilde{\Gamma} \vdash \mathcal{S}(u)$, et on a également $\widetilde{\Gamma} \vdash \mathcal{S}(v)$ par les mêmes raisons. On a de plus, par le théorème 40 page 62, $\widetilde{\Gamma} \vdash u \subset v$. En utilisant la proposition 48 page 69, on en déduit que $u = v$.

On remarque que la notation $\Rightarrow^{\tau}$ est justifiée par le fait 32 page 55, (3).

Paires

On définit la traduction des produits en utilisant les définitions de la page 95 :

– $\widetilde{A \times B} = \widetilde{A} \times \widetilde{B}$

– $\widetilde{(p,q)} = (p, q)$

– $\widetilde{\mathsf{fst}\ p} = p[\mathtt{True}]$
– $\widetilde{\mathsf{snd}\ p} = p[\mathtt{False}]$
Il reste à vérifier que les règles sont correctes :
– $\times I$, $\times EG$, $\times ED$, et $SurPair$ sont immédiates par le fait 82 page 95.
– $\subset_\times$ correspond à la proposition 83 page 96.

Sommes

On définit la traduction des types sommes en utilisant les entiers, le type produit et la réunion binaire :
– Chaque constructeur C est traduit par un entier de Church, qu'on notera $\overline{C}$
– Pour les types :
  – $\widetilde{C} = \overline{C} \times \Lambda X.X$
  – $\widetilde{C\ \mathsf{of}\ A} = \overline{C} \times \widetilde{A}$
  – $\widetilde{A \| B} = A \cup B$
– Pour les programmes :
  – $\widetilde{C} = (\overline{C}, \Lambda X.X)$
  – $\widetilde{C[p]} = (\overline{C}, p)$
  – $\mathsf{cases}\ p\ \mathsf{of}\ M$ est traduit à l'identique, avec la définition 48 page 97.
Le lemme technique suivant montre que tout type somme est de la forme $C \times A$, avec C inclus dans l'ensemble des entiers de Church, et qu'un type somme est formé de types disjoints :

Lemme 99 *Pour tout contexte* Γ *:*

1. *si* $\Gamma \vdash T : \tau_S$*, alors* $\widetilde{\Gamma} \vdash \widetilde{T} \subset (\cup_n(N^\tau(n)) \times \top)$ *(avec* N^τ *défini à la définition 45 page 89).*
2. *Si* $\Gamma \vdash C \notin T$*, alors* $\widetilde{\Gamma} \vdash \forall X : T.(\mathtt{Pi1}[X] \neq C \wedge N^o(\mathtt{Pi1}[X]))$
3. *si* $\Gamma \vdash \mathsf{cases}\ p\ \mathsf{of}\ M : A$*, alors il existe* T *tel que* $\Gamma \vdash T : \tau_S$ *et* $\Gamma \vdash p : T$*.*

preuve:

1. Les cas atomiques étant évidents (C_0 et C_1), il suffit de regarder les cas d'ajout de constructeur. Il est immédiat que, pour tout constructeur C :
   – $\vdash N^o(C)$ (par le lemme 76 page 89)
   – $\vdash C \subset N^\tau(C)$ (par le théorème 75 page 89)
   En d'autres termes, il suffit donc de montrer :

   $$\vdash \forall C, A, T(N^o(C) \Rightarrow (T \subset (\cup_n(N^\tau(n)) \times \top)) \Rightarrow (C \times A) \cup T \subset (\cup_n(N^\tau(n)) \times \top))$$

   En utilisant le 1. du fait 32 page 55, il suffit de montrer, sous les hypothèses :

   $$N^o(C), (T \subset (\cup_n(N^\tau(n)) \times \top)), \mathcal{S}(x), x \subset (C \times A)$$

   que l'on a $x \subset (\cup_n(N^\tau(n)) \times \top)$). Par la propriété d'équivalence des singletons (48 page 69), on en déduit que $\mathcal{S}_\cup(x)$. On vient de voir que $\vdash C \subset N^\tau(C)$, et on a évidemment $A \subset \top$. On a donc bien le résultat.

2. Par induction sur $C \notin T$: on a évidemment $\vdash T : \tau_S$, donc par ce qu'on vient de voir, on a $N^o(\texttt{Pi1}[X])$. On vérifie maintenant que $X : T \vdash \texttt{Pi1}[X] \neq C$, par induction sur le jugement $C \notin T$:
   - $\notin_{C_0}$ et $\notin_{C_1}$: on peut par exemple utiliser la technique utilisée pour montrer que $(\neg(\mathsf{false} = \mathsf{true})$.
   - $\notin_{Sum_0}$ et $\notin_{Sum_1}$: par induction, on a $\forall X : T(\texttt{Pi1}[X] \neq C \wedge N^o(\texttt{Pi1}[X]))$. On met en hypothèse $X \subset C_2 \cup T, \mathcal{S}(X)$. Par la propriété d'équivalence des singletons (48 page 69), on en déduit que $\mathcal{S}_\cup(X)$ et donc $(X \subset C_2) \vee (X \subset T)$. Sous l'hypothèse $X \subset C_2$, on raisonne comme pour les cas de base, et sous l'hypothèse $X \subset T$, on utilise l'hypothèse d'induction.
3. Il suffit d'examiner les règles de typage du case pour se rendre compte qu'elles correspondent aux dérivations de τ_S.

□

Il reste à voir la correction des règles :
- $Constr0_I$ et $Constr1_I$: immédiates par les règles de typage des paires.
- $\_$, $Constr01_E$ et $Constr11_E$: On utilise la proposition 84 page 97 : il faut donc vérifier :
  - Pour $\_$: on a simplement une hypothèse en plus par rapport à la proposition 84 1., qui ne pose pas de condition sur le type A.
  - Pour $Constr11_E$: La première prémisse de la proposition 84 2. correspond à la traduction de $p : C$, la seconde est immédiate du fait que C est traduit par un entier de Church, la troisième vient par introduction du produit $(\widetilde{\Pi^o})$
  - Pour $Constr01_E$: La première prémisse de la proposition 84 3. correspond à la traduction de $p : C$ of A, la seconde est immédiate du fait que C est traduit par un entier de Church, la troisième est identique.
- $Constr02_E$ et $Constr12_E$: on utilise la proposition 85 page 98 :
  - $Constr02_E$:On vérifie les prémisses P_1 à P_6 de la proposition 85 1. :
    - P_1 correspond à $p : \widetilde{C \mid S}$.
    - P_2 vient du fait que $\widetilde{C}$ est un entier de Church.
    - P_3 vient de $C \notin S$ et du lemme 99 2.
    - P_4 vient de $\Gamma \vdash f : B(C)$.
    - P_5 vient de $\Gamma, y : S \vdash \mathsf{cases}\ y\ \mathsf{of}\ M(y) : B(y)$.
  - $Constr12_E$:On vérifie les prémisses P_1 à P_6 de la proposition 85 1. :
    - P_1 correspond à $p : \widetilde{C\mathsf{of}\ A \mid S}$.
    - P_2 vient du fait que $\widetilde{C}$ est un entier de Church.
    - P_3 vient de $C \notin S$ et du lemme 99 2.
    - P_4 vient de $\Gamma, x : A \vdash \widetilde{f : B(C[x])}$.
    - P_5 vient de $\widetilde{\Gamma} \vdash \mathcal{S}(\mathsf{fun}\ x \to f)$, car $\Gamma \vdash \mathsf{fun}\ x \to f : \Pi^\tau x : A.B(C[x])$, et avec le lemme de singletonicité des programmes (98 page 151).
    - P_6 vient de $\Gamma, y : S \vdash \mathsf{cases}\ y\ \mathsf{of}\ M(y) : B(y)$.
- Règles logiques :
  - $C_\neq$: Si $\mathcal{S}(x) \vdash \widetilde{C[x]} = \widetilde{C'}$, alors $(C, x) = (C', \Lambda X X)$, ce qui entraîne $C = C'$ par la

première projection.
- C_{log} et C of $_{log}$: Le sens gauche-droite est immédiat et démontrable dans notre système. Le sens droite-gauche vient de $\widetilde{C} = (C, \Lambda XX)$ (respectivement $\widetilde{C[x]} = (C, x)$) et de la paire surjective (fait 82 page 95).
- $|_{log}$ et $|_{typ}$: on suppose $x : \widetilde{A \mid B}$, soit $x \subset A \cup B, \mathcal{S}(x)$; par l'équivalence des singletons (48 page 69), on a $\mathcal{S}_\cup(x)$, et donc $(x \subset A) \vee (x \subset B)$, ce qui permet d'utiliser les prémisses.

Types inductifs

On aura besoin de deux lemmes. Le premier est un lemme général sur le produit :

Lemme 100 (relation entre produits) *On a la relation suivante :*

$$\forall X, A \forall B(X \subset \Pi^\tau x : A.B(x) \Leftrightarrow \Pi^o x : A(X \subset (x \Rightarrow B(x))))$$

preuve: Ce sont de simples applications des règles de sous-typage de $\forall$ et de $\twoheadrightarrow$. □

Le deuxième est un lemme sur le sous-typage du schéma de compréhension :

Lemme 101 (sous-typage du schéma de compréhension) *On a les propriétés suivantes*

1. $\vdash \forall P \forall x(\mathcal{S}(x) \quad \Rightarrow \quad (P(x) \Longleftrightarrow x \subset \{r : \top; P(r)\}))$
2. $\vdash \forall A \forall P(A \subset \{r : \top; P(r)\} \Longleftrightarrow \Pi^o x : A.P(x))$

(en notant $\top = \cup X.X$)

preuve:

1. On met en hypothèse $\mathcal{S}(x)$.
   - Sens gauche-droite : On met en hypothèses $P(x)$ et $(\forall x(x \restriction (\mathcal{S}(x) \wedge (x \subset \top) \wedge P(x))) \subset K)$, et il faut en déduire $x \subset K$, ce qui est immédiat par les propriétés de l'opérateur restriction (fait 30 page 54).
   - Sens droite-gauche : On suppose $x \subset \{r : \top; P(r)\}$. Par définition de $\mathcal{S}(x)$, cela implique $\exists y(x \subset (y \restriction (S(y) \wedge y \subset \top \wedge P(y)))$. Pour éliminer le $\exists$, supposons $x \subset (y \restriction (S(y) \wedge (y \subset \top) \wedge P(y)))$. Comme on sait que $x \neq \emptyset$ par définition de $\mathcal{S}$, on en déduit par les propriétés de l'opérateur restriction (fait 30) que $(S(y) \wedge y \subset \top \wedge Py)$.De plus, comme $(y \restriction (S_\tau(y) \wedge Py) \subset y)$, il vient $x \subset y$, d'où l'on déduit, en utilisant l'équivalence des singletons (propriété 48 page 69), et par définition de $\mathcal{S}_\subset$, que $y \subset x$, et donc que $x = y$ et finalement $P(x)$. On en déduit que $P(x)$ en éliminant le $\exists$.
2. - Sens gauche-droite : on met en hypothèses $A \subset \{r : \top; P(r)\}$, $\mathcal{S}(x)$ et $x \subset A$. Par 1. on a $P(x)$.
   - Sens droite-gauche : on met en hypothèses $\Pi^o x : A.P(x)$. En utilisant le 1. du fait 32 page 55, il suffit de montrer $\Pi^o x : A.(x \subset \{r : \top; P(r)\})$, ce qui est immédiat.

□

Pour définir la traduction de l'opérateur de point fixe, on pose :
- $\Delta_F = \Lambda X.F[X[X]]$
- $\Psi = \Lambda F.\Delta_F[\Delta_F]$

On étend la traduction en posant :

$$\widetilde{(\mathsf{rec}\ p)} = \Delta_{\tilde{p}}[\Delta_{\tilde{p}}]$$

L'opérateur μ est traduit à l'identique (définition 34 page 58).
On a, de plus, le fait évident suivant :

Fait 102 *Si* $\Gamma \vdash \mu X.F(X) : \tau$, *alors* $\widetilde{\Gamma} \vdash Cresc(F)$, $Cresc$ *étant défini dans la définition 35 page 59.*

preuve: Ceci correspond à la prémisse (*cresc*) de la règle μ_τ. □

On montre maintenant les règles :
- *Induc* : On met en hypothèse $\forall A(\Pi^o x : A.P(x) \Rightarrow A \subset \mu X.F(X) \Rightarrow \Pi^o x : F(A).P(x))$, et il faut montrer que $\Pi^o x : \mu X.F(X).P(x)$. En utilisant le 2. du fait 101, il suffit de montrer que $\mu X.F(X) \subset \{x : \top; P(x)\}$. Cela vient de $\mu X.F(X) \subset \mu X.F(X) \cap \{x : \top; P(x)\}$, que l'on montre en utilisant la règle de sous-typage (3) du fait 38 page 59. Il faut donc montrer $\forall A(A \subset \mu X.F(X) \cap \{x : \top; P(x)\} \Rightarrow F(A) \subset \mu X.F(X) \cap \{x : \top; P(x)\})$. Ceci est immédiat en utilisant le fait que $Cresc(F)$, et le 2. du fait 101.
- Rec_μ : On montre en fait, sous l'hypothèse $Cresc(F)$, le typage suivant :

$$\Psi \subset \forall A(\Pi^\tau x : A.B(x) \Rightarrow^\tau (A \subset \mu X.F(X)) \twoheadrightarrow \Pi^\tau x : F(A).B(x)) \Rightarrow^\tau \Pi^\tau x : \mu X.F(X).B(x)$$

En utilisant le fait 28 page 52, et en posant $H = \forall A(\Pi^\tau x : A.B(x) \Rightarrow^\tau (A \subset \mu X.F(X)) \twoheadrightarrow \Pi^\tau x : F(A).B(x))$ il suffit de montrer que :

$$\Delta_H[\Delta_H] \subset \Pi^\tau x : \mu X.F(X).B(x)$$

En utilisant le théorème 40 page 62 et le lemme 100, et la règle *Induc*, il suffit de montrer :

$$\forall A((\Pi^o x : A.\Delta_H[\Delta_H] \subset x \Rightarrow^\tau B(x)) \Rightarrow A \subset \mu X.F(X) \Rightarrow (\Pi^o x : F(A).\Delta_H[\Delta_H] \subset x \Rightarrow^\tau B(x)))$$

On met en hypothèses $(\Pi^o x : A.\Delta_H[\Delta_H] \subset x \Rightarrow^\tau B(x)), A \subset \mu X.F(X)$, et il suffit d'en déduire que $\Pi^o x : F(A).H[\Delta_H[\Delta_H]] \subset x \Rightarrow^\tau B(x)$. Ceci vient de simplement de la définition de H, du lemme 100, et de quelques règles faciles de sous-typage. Pour prouver que la règle est correcte, on voit simplement que, par hypothèse d'induction et ce qu'on vient de voir :

$$\Psi[\widetilde{\mathsf{fun}\ x \to t}] \subset \Pi^\tau x : \mu X.F(X)$$

De plus, on a $\mathcal{S}(\Psi[\widetilde{\mathsf{fun}\ x \to t}])$, $\mathcal{S}((\widetilde{\mathsf{rec\ fun}\ x \to t}))$ et $\Psi[\widetilde{\mathsf{fun}\ x \to t}] \subset (\widetilde{\mathsf{rec\ fun}\ x \to t})$ En utilisant la propriété d'équivalence des singletons, on a $\mathcal{S}_{\subset}(\Psi[\widetilde{\mathsf{fun}\ x \to t}])$, ce qui entraîne donc :

$$(\widetilde{\mathsf{rec\ fun}\ x \to t})] \subset \Pi^{\tau} x : \mu X.F(X)$$

– $BFRec_{\mu}$: Le prédicat "être une relation bien fondée" est traduit à l'identique. On montre en fait, sous l'hypothèse que R est bien fondée sur le type A, une relation de sous-typage un peut plus générale, valable pour n'importe quel type (pas nécessairement inductif) :

$$\Psi \subset \forall x(x \in A \twoheadrightarrow ((\Pi^{\tau} y : A.((Ryx) \twoheadrightarrow B(y))) \Rightarrow x \Rightarrow B(x))) \Rightarrow \Pi^{\tau} x : A.B(x)$$

Pour cela, on pose :

$$F = \forall x(x \in A \twoheadrightarrow ((\Pi^{\tau} y : A.((Ryx) \twoheadrightarrow B(y))) \Rightarrow x \Rightarrow B(x)))$$

et on montre :

$$F \Rightarrow \Delta_F[\Delta_F] \subset F \Rightarrow \Pi^{\tau} x : A.B(x)$$

Il suffit donc de montrer

$$\Delta_F[\Delta_F] \subset \Pi^{\tau} x : A.B(x)$$

Soit, sous les hypothèses $\mathcal{S}(x)$ et $x \subset A$, on a donc à montrer :

$$\Delta_F[\Delta_F] \subset x \Rightarrow B(x)$$

On fait la preuve par induction, il s'agit de montrer :

$$\Pi^{o} x : A(\Pi^{o} y : A.(Ryx \Rightarrow (\Delta_F[\Delta_F] \subset (y \Rightarrow B(y)))) \Rightarrow (\Delta_F[\Delta_F] \subset (x \Rightarrow B(x))))$$

On met en hypothèse $\Pi^{o} y : A.(Ryx \Rightarrow (\Delta_F[\Delta_F] \subset (y \Rightarrow B(y))))$. Par les propriétés de l'application, il suffit de montrer que
$\Delta_F \subset (\Delta_F \Rightarrow x \Rightarrow B(x))$.
Pour cela, on montre :

$$\Delta_F \Rightarrow F[\Delta_F[\Delta_F]] \subset \Delta_F \Rightarrow x \Rightarrow B(x)$$

Encore par les propriétés de l'application, il suffit de montrer que

$$F \subset \Delta_F[\Delta_F] \Rightarrow x \Rightarrow B(x)$$

En remplaçant F par sa définition, il suffit de monter :

$$\forall x(x \in A \twoheadrightarrow ((\Pi^{\tau} y : A.((Ryx) \twoheadrightarrow B(y))) \Rightarrow x \Rightarrow B(x))) \subset \Delta_F[\Delta_F] \Rightarrow x \Rightarrow B(x)$$

On obtient facilement que cela découle de :

$$\Delta_F[\Delta_F] \subset \forall x(x \in A \twoheadrightarrow (\Pi^\tau y : A.((Ryx) \twoheadrightarrow B(y)))$$

En utilisant l'inversion des flèches (axiome 1 page 42), en mettant en hypothèses $y \in A$ et Ryx, on obtient tout de suite le résultat.

- μ_{PF} : Immédiat avec le fait précédent et le fait 39 page 59.
- $\subset_\mu$: On utilise le (3) du fait 38 page 59, en ayant en plus, par le fait précédent, $Cresc(F)$. On met en hypothèse $F(A) \subset A$, et il faut en déduire $\forall B(B \subset A \Rightarrow F(B) \subset A)$. C'est immédiat par $Cresc$.

Types co-inductifs

La traduction du plus petit grand fixe d'un opérateur est la suivante :

$$\nu X.F(X) = \cup X.(X \restriction \forall A(X \subset A \Rightarrow X \subset F(A)))$$

Comme pour les types inductifs, le terme "plus grand point fixe" est un abus dans le cas général. C'est en effet le plus grand point fixe d'un opérateur F dans le cas où F est croissant, mais dans le cas général, on a seulement $\nu X.F(X) \subset F(\nu X.F(X))$. Les faits que nous établissons maintenant sont les frères (duaux) de ceux établis dans le cas inductifs.

Fait 103 *On montre :*

(1) $\forall F \forall X(\nu X.F(X) \subset X \Leftrightarrow \forall Y(\forall A(Y \subset A \Rightarrow Y \subset F(A)) \Rightarrow Y \subset X))$
(2) $\forall F \forall X(\nu X.F(X) \subset X \Rightarrow \nu X.F(X) \subset F(X))$
(3) $\forall F \forall X(\forall A(X \subset A \Rightarrow X \subset F(A)) \Rightarrow X \subset \nu X.F(X))$

preuve:

- (1) On montre plus généralement que $\vdash \forall P, \forall X(\cup X.(X \restriction P(X)) \subset X \Leftrightarrow \forall Y(P(Y) \Rightarrow Y \subset X))$:
  - Sens gauche-droite : il suffit de montrer que $\forall Y(P(Y) \Rightarrow Y \subset \cup X.(X \restriction P(X))$. Pour cela, on suppose $P(Y)$ et $\forall X(X \restriction P(X) \subset K)$, et on en déduit que $Y \restriction P(Y) \subset K$. De plus, comme $Y \subset Y \restriction P(Y)$, on a bien $Y \subset K$, ce qui termine la preuve.
  - Sens droite-gauche : il suffit de montrer sous l'hypothèse $\forall Y(P(Y) \Rightarrow Y \subset X)$ que $\forall Y(Y \restriction P(Y) \subset X) \twoheadrightarrow X \subset X$, ce qui découle de $\forall Y(Y \restriction P(Y) \subset X)$, qui est une conséquence de l'hypothèse.
- (2) est immédiat à partir de (1).
- (3) est une conséquence du fait plus général suivant : $\forall P \forall X(P(X) \Rightarrow X \subset \cup X.(X \restriction P(X))$, qui est immédiat.

□

Le fait suivant donne sens à l'appellation "plus grand point fixe" :

Fait 104 *On montre :*

(1) $\vdash \forall F(\nu X.F(X) \subset F(\nu X.F(X)))$
(2) $\vdash \forall F(Cresc(F) \Rightarrow \nu X.F(X) = F(\nu X.F(X)))$
(3) $\vdash \forall F(Cresc(F) \Rightarrow \forall I(F(I) = I \Rightarrow I \subset \nu X.F(X)))$

preuve:
- (1) est une conséquence du fait précédent (2).
- (2) : il suffit de montrer que $F(\nu X.F(X)) \subset \nu X.F(X)$. En utilisant le fait précédent (3), il suffit de montrer que $\forall A(F(\nu X.F(X)) \subset A \Rightarrow F(\nu X.F(X)) \subset F(A))$; on suppose que $F(\nu X.F(X)) \subset A$, et par (1), on obtient que $\nu X.F(X) \subset A$, et par croissance que $F(\nu X.F(X)) \subset F(A)$.
- (3) : par le fait précédent (3), il suffit de montrer sous les hypothèses $Cresc(F)$ et $F(I) = I$ que $\forall A(I \subset A \Rightarrow I \subset F(A))$, ce qui est immédiat.

□

Après ces faits que l'on peut qualifier de préliminaires, on montre les faits qui nous intéressent plus directement : les règles concernant l'opérateur de point fixe.

Fait 105 *On montre :*

$$(1) \quad \vdash \forall F, A(\Psi \subset \forall X((A \Rightarrow X) \Rightarrow (A \Rightarrow F(X))) \Rightarrow A \Rightarrow \nu X.F(X))$$
$$(2) \quad \vdash \forall F(\Psi \subset \forall X(X \Rightarrow F(X)) \Rightarrow \nu X.F(X))$$

(avec Ψ défini comme au paragraphe précédent)

preuve:
- (1) : Si on pose $H = \forall X((A \Rightarrow X) \Rightarrow (A \Rightarrow F(X)))$, il suffit d'en déduire $\Delta_H[\Delta_H] \subset A \Rightarrow \nu X.F(X)$, donc que $\Delta_H[\Delta_H][A] \subset \nu X.F(X)$ D'après le fait 103 (3), il suffit de montrer que $\forall X(\Delta_H[\Delta_H][A] \subset X \Rightarrow \Delta_H[\Delta_H][A] \subset F(X)$. On met en hypothèse $\Delta_H[\Delta_H][A] \subset X$, d'où l'on déduit que $\Delta_H[\Delta_H] \subset A \Rightarrow X$, et donc que $H[\Delta_H[\Delta_H]] \subset A \Rightarrow F(X)$. Comme $\Delta_H[\Delta_H] \subset H[\Delta_H[\Delta_H]]$, on obtient bien que $\Delta_H[\Delta_H][A] \subset F(X)$.
- (2) : La preuve est encore plus simple : on pose $H = \forall X(X \to F(X))$, et il faut en déduire $\Delta_H[\Delta_H] \subset \nu x.F(X)$. D'après le fait 103 (3), il suffit de montrer que $\forall X(\Delta_H[\Delta_H] \subset X \Rightarrow \Delta_H[\Delta_H] \subset F(X)$. On met en hypothèse $\Delta_H[\Delta_H] \subset X$, d'où l'on déduit que $H[\Delta_H[\Delta_H]] \subset F(X)$. Comme $\Delta_H[\Delta_H] \subset H[\Delta_H[\Delta_H]]$, on obtient bien que $\Delta_H[\Delta_H] \subset F(X)$, ce qui termine la preuve.

□

On montre maintenant les règles :
- $Coinduc_o$: Avec le sous-typage du schéma de compréhension (fait 101), il suffit de montrer sous l'hypothèse $F(\nu F) \subset \{x : \top; P(x)\}$, que $\nu F \subset \{x : \top; P(x)\}$. Avec le fait 104 (1), c'est immédiat.
- $Coinduc_\tau$: Avec le fait 100, il suffit de montrer, en notant $T = \widetilde{t}$, que sous l'hypothèse $\Pi_o x : F(\nu F).(T \subset x \Rightarrow B(x))$, on a $\Pi_o x : \nu F.(T \subset x \Rightarrow B(x))$. Ceci vient de la règle $Coinduc_o$, que l'on vient de vérifier.
- ν_{creat1} : On suppose $\widetilde{t} \subset \forall X(X \Rightarrow F(X))$. Par la croissance de l'application et le fait 105 (2), on a $\Psi[\widetilde{t}] \subset \nu F$. On conclut comme dans la preuve de Rec_μ.
- ν_{creat2} : On suppose $\widetilde{t} \subset \forall X((A \Rightarrow X) \Rightarrow (A \Rightarrow F(X)))$. Par la croissance de l'application et le fait 105 (1), on a $\Psi[\widetilde{t}] \subset A \Rightarrow \nu F$. On conclut comme dans la preuve de Rec_μ.

- ν_{PF} : On remarque, comme pour μ, que si $\nu f : \tau$, alors $Cresc(F)$. En utilisant le fait 104 (2), on obtient bien l'égalité voulue.
- $\subset_\nu$: Avec le fait 103 (3), il suffit de montrer, sous l'hypothèse que $A \subset F(A)$, que $\forall Y(A \subset Y \Rightarrow A \subset F(Y))$. Ceci est immédiat par croissance de F.

Enregistrements

Chaque label $l \in \mathcal{L}$ est traduit par un entier de Church, qu'on notera $\overline{l}$.
La traduction des types d'enregistrements, des enregistrements, des projections et de l'ajout de champs est très directe, avec ce qui a été fait dans le chapitre 6 :

$$\begin{array}{lcl}
\{l_1 : A_1; \ldots; l_n : A_n\} & = & \{\overline{l_1} : \widetilde{A_1}; \ldots; \overline{l_n} : \widetilde{A_n}\} \\
< l_1 = t_1; \ldots; l_n = t_n > & = & < \overline{l_1} = \widetilde{t_1}; \ldots; \overline{l_n} = \widetilde{t_n} > \\
\widetilde{p.l} & = & \widetilde{p}.\overline{l} \\
[l = p]+q & = & [\overline{l} = \widetilde{p}] + \widetilde{q}
\end{array}$$

En utilisant les définitions 49 page 100 et 50 page 100.
Rappelant ici que le traduit d'un programme est un *type*, on donne la traduction des enregistrements forts, et de la construction "as" :

$$\begin{array}{lcl}
\{l_1 : A_1; \ldots; l_n : A_n\}_S & = & \cup X_1 \in A_1 \ldots X_n \in A_n < \overline{l_1} = X_1; \ldots; \overline{l_n} = X_n > \\
p \text{ as } l_1, \ldots, l_n & = & < l_1 = p.l_1; \ldots; l_n = p.l_n >
\end{array}$$

On fera dans la suite un léger abus d'écriture consistant à prendre directement les définitions dans ST pour ne pas surcharger de $\widetilde{}$.

Il ne reste plus qu'à vérifier la correction des règles (on ne vérifie pas que l'on a des singletons à chaque fois, car ceci est désormais usuel, comme dans le lemme 98 page 151) :

- $\{\}^-_\subset$: il s'agit simplement du cas (2) de la proposition 88 page 102.
- $\{\}^\subset_\subset$: il s'agit simplement du cas (1) de la proposition 88 page 102.
- $\{\}^\sigma_\subset$: On remarque que l'ordre des labels n'a pas d'importance dans la traduction, car il intervient entre des membres d'une réunion.
- $\{\}^S_\subset$: on utilise la caractérisation du sous-typage par les singletons (fait 32 page 55, 1.) : on suppose $\mathcal{S}(x)$, $x \subset \{l_1 : A_1; \ldots; l_n : A_n\}_S$, et il faut montrer que $x \subset \{l_1 : A_1; \ldots; l_n : A_n\}$. Par application répétée de la définition de singleton, il suffit de montrer sous les hypothèses :

$$y_1 : \tau, y_1 \in A_1, \ldots, y_n : \tau, y_n \in A_n, x \subset < l_1 = y_1; \ldots; l_n = y_n >$$

  que $x \subset \{l_1 : A_1; \ldots; l_n : A_n\}$, ce qui est immédiat.
- $\{\}_I$: on utilise de manière répétée les (1) et (2) de la proposition 86 page 100. Il suffit de vérifier que les prémisses sont vérifiées :
  - (1) : c'est clair car on a $N^o(\overline{l})$.

- (2) : il faut vérifier la dernière prémisse. Cela est immédiat par définition de τ_R, qui impose que les labels soient distincts.

– $\{\}_E$: immédiat par la proposition 87 page 101.
– $\{\}_+$: Il s'agit du (3) de la proposition 86 page 100. Les prémisses sont vérifiées par τ_R.
– $\{\}_S$: sous l'hypothèse $p \in \{l_1 : A_1; \ldots; l_n : A_n\}$, on déduit $p.l_i \in A_i$ (pour $i = 1, \ldots, n$), et donc p $\widetilde{\mathsf{as}\ l_1, \ldots, l_n} \subset \{l_1 : A_1; \ldots; l_n : A_n\}$ par introductions successives de la réunion.

Abstractions

Les traductions des opérateurs $\mathsf{Some}^{s_1,\ldots,s_n}$ et des projections "$p :^{s_1,\ldots,s_n} T.i$" sont, en posant $s = s_1, \ldots, s_n$:

$$\begin{array}{lcl} \widetilde{\mathsf{Some}^s T} & = & \cup^s x_1 \ldots x_n.(\widetilde{T}(x_1) \ldots (x_n)) \\ \widetilde{p :^s T.i} & = & \widetilde{p}.\widetilde{T}.i \end{array}$$

En reprenant les notations de la définition 52 et du fait 89 page 103.

On vérifie les règles :
– $\subset^r_{\mathsf{Some}}$: immédiat par le fait 89 3..
– $\subset^l_{\mathsf{Some}}$: immédiat par définition de la réunion.
– Some_{proj} : par induction, on a $\mathcal{S}(\widetilde{p})$ et $\widetilde{p} \subset \cup^s T$. Cela permet d'utiliser directement le fait 89 5.

La traduction de l'opérateur Self s'exprime avec le (traduit du) schéma de compréhension :

$$\widetilde{\mathsf{Self}\ T} = \{x : \top; x \subset T(x)\}$$

On vérifie les règles :
– Self_I : Par hypothèse d'induction, on a $p \in T(p)$, et donc $p \in \{x : \top; x \subset T(x)\}$.
– Self_E : Par hypothèse d'induction, on a $p \in \{x : \top; x \subset T(x)\}$. En utilisant SC^τ_E, on en déduit $p \subset T(p)$. Comme on a $\mathcal{S}(p)$, la conclusion est immédiate.
– $\mathsf{Self}_\subset$: On a $x : \mathsf{Self}\ t.A(t) \vdash A(x) \subset B(x)$. On en déduit :

$$\mathcal{S}(x), x \subset \mathsf{Self}\ t.A(t) \vdash x \subset B(x)$$

et donc

$$\mathcal{S}(x), x \subset \mathsf{Self}\ t.A(t) \vdash x \subset \mathsf{Self}\ t.B(t)$$

On conclut par le (1) du fait 32 page 55.

Terminaison

On pose :

$$\widetilde{NORM} = \mathcal{N}$$

Le type $\mathcal{N}$ étant défini à la page 79.
Pour regarder la validité des règles, on commence par un petit lemme sur le produit, base de nos types de données :

Lemme 106 *Le séquent suivant est dérivable :*

$$\vdash (A \subset \mathcal{N}) \Rightarrow (B \subset \mathcal{N}) \Rightarrow (A \times B \subset \mathcal{N})$$

preuve: En utilisant le fait 32 page 55, On met en hypothèses

$$A \subset \mathcal{N}, B \subset \mathcal{N}, \mathcal{S}(x), x \subset A \times B$$

et il faut en déduire $x \subset \mathcal{N}$. En utilisant le fait 82 page 95, on en déduit $x \subset A\overline{\times}B$ ($A\overline{\times}B$ étant le type produit du système F), et donc il suffit de montrer $A\overline{\times}B \subset \mathcal{N}$. On met en hypothèses $Adequ(N, N_o)$, et on en déduit $A \subset N$ et $B \subset N$. On a donc :

$$\vdash A\overline{\times}B \subset N\overline{\times}N$$

On conclut par la chaîne de sous-typage :

$$\begin{array}{rcr} N\overline{\times}N & \subset & (N \Rightarrow (N \Rightarrow N)) \Rightarrow N \\ & \subset & (N \Rightarrow (N \Rightarrow N_o)) \Rightarrow N \\ & \subset & (N \Rightarrow N_o) \Rightarrow N \\ & \subset & N_o \Rightarrow N \\ & \subset & N \end{array}$$

□

On vérifie maintenant la validité des règles :
- Constructeurs normalisables : comme $\widetilde{C} = \overline{C} \times \Lambda X.X$, que $\overline{C} \subset \mathcal{N}$ et $\Lambda X.X \subset \mathcal{N}$, on a donc le résultat par le lemme précédent.
- Constructeur appliqué à un argument normalisable : idem.
- Sommes normalisables : immédiat par propriété de la réunion
- Paires normalisables : il s'agit du lemme précédent.
- Enregistrements forts normalisables : on met en hypothèses $A_1 \subset \mathcal{N}, \ldots, A_n \subset \mathcal{N}$, et il faut en déduire $\{l_1 : A_1; \ldots; l_n : A_n\} \subset \mathcal{N}$. En utilisant le fait 32 page 55, on ajoute aux hypothèses $\mathcal{S}(x), x \subset \{l_1 : A_1; \ldots; l_n : A_n\}$, et il suffit d'en déduire $x \subset \mathcal{N}$. En utilisant la définition de singleton, il suffit de montrer sous les hypothèses $y_1 \in A_1, \ldots, y_n \in A_n$ que $< l_1 = y_1; \ldots; l_n = y_n > \subset \mathcal{N}$ Les y_i étant en position positive dans $< l_1 = y_1; \ldots; l_n = y_n >$, on a $< l_1 = y_1; \ldots; l_n = y_n > \subset < l_1 = \mathcal{N}; \ldots; l_n = \mathcal{N} >$ et on conclut comme dans la preuve du théorème 62 , page 79.
- Normalisabilité des types inductifs : claire par le fait qu'elle est dérivable à l'aide de la règle $\subset_\mu$.

Ceci clôt la validation des règles du langage.

9.3 Validation des règles des modules

Les modules sont traduits directement dans le système ST, via une "grosse" conjonction qui exprime leur comportement[2]. Pour simplifier les écritures, on utilisera les notations Some et Self directement dans le système ST.

9.3.1 Traduction des structures et des signatures

Structures

Rappelons tout d'abord la traduction d'un enregistrement : il s'exprime par un `fun` x `->` `cases` x `of` $M(x)$, le motif M étant défini à partir de la succession de déclarations de l'enregistrement.
Autrement dit, la construction d'un enregistrement se fait en trois temps :

1. Définition du motif
2. Clôture du motif par la construction `cases`
3. Clôture de l'enregistrement par l'abstraction `fun`

Les clôtures seront en fait effectuées par la construction Struct. Donnons maintenant la traduction :

– La structure vide est traduite par un champs avec un nom de label (i.e. un entier) l_0 supposé interdit (il suffit par exemple de prendre pour les labels autorisés les entiers strictement positifs, et $l_0 = 0$) :

$$\widetilde{\mathsf{end}} \quad = \quad (l_0 = l_0)$$

– L'ajout d'un champs est une liaison :

$$\mathsf{let}\ x\ \mathsf{as}\ \widetilde{l = p}\,!\ S(x) \quad = \quad (l = p)\,;\ S(p)$$

– La fermeture d'une structure est la construction d'un enregistrement :

$$\widetilde{\mathsf{Struct}}\ S \quad = \quad < S >$$

Signatures et projections

Les noms sont traduits par des entiers de Church, la seule propriété qu'on leur demande étant en fait d'être distinguables par le système (et non nécessairement distinguables par un programme).
Prenons comme ensemble de sortes les quatre suivantes (types, types paramétrés, propositions et prédicats) :

$$\tau, \tau \to \tau, o, \tau \to o$$

[2]Les règles sont validées dans le logiciel PhoX, à ceci près que l'on n'a considéré qu'une seule sorte de définition abstraite : celle des types

Pour cet ensemble de sortes (notons-les s_1 à s_4), les signatures en cours de construction attendent en arguments quatre fonctions (T_1 à T_4 de sortes respectives $\tau \to s_1$ à $\tau \to s_4$), et un type ($X : \tau$) : elles seront donc de la forme $\lambda T_1.\lambda T_2.\lambda T_3.\lambda T_4.\lambda X.s(T_1)(T_2)(T_3)(T_4)(X)$. Pour alléger les écritures, on notera simplement :

$$\lambda \overrightarrow{T}, X.s(\overrightarrow{T})(X)$$

– La signature vide est le prédicat vrai :

$$\widetilde{\mathsf{end}} = \lambda \overrightarrow{T}, X.true$$

(avec *true* une tautologie telle que $\forall X(X \Rightarrow X)$).

– L'ajout d'une déclaration de valeur :

$$\mathsf{val}\ x\ \mathsf{as}\ \widetilde{l : A}\,!\ S(x) \quad = \quad \lambda \overrightarrow{T}, X.(X.l \in A \wedge (\widetilde{S(X.l)}(\overrightarrow{T})(X)))$$

– L'ajout d'un fait :

$$\widetilde{\mathsf{fact}\ P\,!}\ S \quad = \quad \lambda \overrightarrow{T}, X.(P \wedge (\widetilde{S}(\overrightarrow{T})(X)))$$

– L'ajout d'une déclaration d'objet de sorte s_i ($i = 1, \ldots, 4$) concret :

$$\begin{array}{lcr}
\mathsf{def}_\tau\ t\ \mathsf{as}\ \widetilde{n = A}\,!\ S(t) & = & \lambda \overrightarrow{T}, X.(T_1(n) = A \wedge S(A)(\overrightarrow{T})(X)) \\
\mathsf{def}_{\tau\to\tau}\ f\ \widetilde{\mathsf{as}\ n = F}\,!\ S(f) & = & \lambda \overrightarrow{T}, X.(\forall Y(T_2(n)(Y) = F(Y)) \wedge S(F)(\overrightarrow{T})(X)) \\
\mathsf{def}_o\ p\ \widetilde{\mathsf{as}\ n = P}\,!\ S(p) & = & \lambda \overrightarrow{T}, X.(T_3(n) \Leftrightarrow P \wedge S(P)(\overrightarrow{T})(X)) \\
\mathsf{def}_{\tau\to o}\ g\ \widetilde{\mathsf{as}\ n = G}\,!\ S(g) & = & \lambda \overrightarrow{T}, X.(\forall Y(T_3(n)(Y) \Leftrightarrow G(Y)) \wedge S(G)(\overrightarrow{T})(X))
\end{array}$$

– L'ajout d'une déclaration d'objet de sorte s_i ($i = 1, \ldots, 4$) abstrait :

$$\mathsf{def}_{s_i}\ \widetilde{t\ \mathsf{as}\ n}\,!\ S(t) \quad = \quad \lambda \overrightarrow{T}, X.(\widetilde{S(T_i(n))}(\overrightarrow{T})(X)$$

– Clôture d'une signature :

$$\widetilde{\mathsf{Sig}\ S} = \mathsf{Some}\ \overrightarrow{T}.SC\lambda X.\widetilde{S}(\overrightarrow{T}(X)$$

– Projections (on note $s = s_1, \ldots, s_4$) :

$$(\widetilde{m :_{s_i} S}).n = T_i(m)(n)$$

avec

$$T_i(m) = (m :^s \lambda \overrightarrow{T}.SC\lambda Y.\widetilde{S}\overrightarrow{T}(Y)).i$$

9.3.2 Exemple de traduction

Reprenons l'exemple de la signature des booléens vu dans le chapitre des modules (en supprimant les "..") :

$$
\begin{array}{ll}
\mathsf{Sig} & \\
\quad \mathsf{def}_{\tau}\ \mathsf{bool} = \mathsf{TRUE} \mid \mathsf{FALSE} & ! \\
\quad \mathsf{val\ true} : \mathsf{bool} & ! \\
\quad \mathsf{val\ false} : \mathsf{bool} & ! \\
\quad \mathsf{val\ ifthenelse} : \forall A(\mathsf{bool} \Rightarrow A \Rightarrow A \Rightarrow A) & ! \\
\quad \mathsf{fact}\ \forall A \Pi^{o} x, y : A\ \mathsf{ifthenelse\ true\ x\ y} = \mathsf{x} & ! \\
\mathsf{end} &
\end{array}
$$

Son traduit sera la conjonction :

$$
\mathsf{Some}\ T_1, \ldots, T_4 . SC \lambda m. \left(\begin{array}{c}
T_1(\mathsf{bool}) = \mathsf{TRUE} \mid \mathsf{FALSE} \\
\wedge \\
\mathsf{m.true} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\
\wedge \\
\mathsf{m.false} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\
\wedge \\
\mathsf{m.ifthenelse} \in \forall A(\ \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow A \Rightarrow A \Rightarrow A) \\
\wedge \\
\forall A \Pi^{o} x, y : A\ \mathsf{m.ifthenelse\ m.true\ x\ y} = \mathsf{x}
\end{array} \right)
$$

On remarque qu'il ne s'agit en fait pas d'un type enregistrement, mais d'une conjonction exprimant les propriétés vérifiées par les fonctions de projection (Some...) et par le programme (SC...).

9.3.3 Règles d'introduction

On traduit $m :_S S$ par le prédicat inductif suivant (qui reflète les règles de typage des modules) :

$$
\begin{array}{l}
\forall X \\
\left(\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad X(\mathsf{end})(\mathsf{end}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\forall a, A, m, s, C(\left(\begin{array}{c} X(m(a))(s(a)) \\ \wedge \\ a \in A \\ \wedge \\ N^o(C) \\ C \notin s(a) \end{array}\right) \Rightarrow X(\mathsf{let}\ x\ \mathsf{as}\ C = a\ m(x))(\mathsf{val}\ x\ \mathsf{as}\ C\ \mathsf{:A}\ s(x))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\forall P, m, s(\left(\begin{array}{c} X(m)(s) \\ \wedge \\ P \end{array}\right) \Rightarrow X(m)(\mathsf{fact}\ P\,!\mathsf{s})) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\forall A, C, m, s(\left(\begin{array}{c} X(m)(s(A)) \\ \wedge \\ C \notin s(A) \end{array}\right) \Rightarrow X(m)(\mathsf{def}_\tau\ t\ \mathsf{as}\ C = A\,!\ (\mathsf{s}\ \mathsf{t}))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\forall A, C, m, s(\left(\begin{array}{c} X(m)(s(A)) \\ \wedge \\ C \notin s(A) \end{array}\right) \Rightarrow X(m)(\mathsf{def}_{\tau\to o}\ t\ \mathsf{as}\ C = A\,!\ (\mathsf{s}\ \mathsf{t}))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad X(m)(S)
\end{array}\right)
\end{array}
$$

(La traduction du prédicat $C \notin S$ sera donné dans la section des "lemmes techniques"). La proposition suivante exprime la correction des règles de construction des modules :

Proposition 107 *La règle suivante est dérivable :*

$$\vdash \forall m, S(m :_S S \Rightarrow \mathit{Struct}\ m \in \mathit{Sig}\ S)$$

La preuve nécessite quelques lemmes techniques que nous détaillons ci-après.

Lemmes techniques

On définit le prédicat d'absence d'un nom par :

$$
\begin{array}{l}
C \notin S = \\
\forall X \\
\left(\begin{array}{lcll}
\forall C(N^o(C) & \Rightarrow & X(C)(\mathsf{end})) & \Rightarrow \\
\forall P, s(X(C)(s) & \Rightarrow & X(C)(\mathsf{fact}\ P\,!s)) & \Rightarrow \\
\\
\forall A, D \forall s(\left(\begin{array}{c} X(C)(s(A)) \\ \wedge \\ \neg(C =_{\subset} D) \end{array} \right) & \Rightarrow & X(C)(\mathsf{def}_\tau\ t\ \mathsf{as}\ D = A\,!\ (s(t)))) & \Rightarrow \\
\vdots \\
\forall A, D \forall s(\left(\begin{array}{c} X(C)(s(A)) \\ \wedge \\ \neg(C =_{\subset} D) \end{array} \right) & \Rightarrow & X(C)(\mathsf{def}_{\tau\to o}\ t\ \mathsf{as}\ D = A\,!\ (s(t)))) & \Rightarrow \\
\\
\forall s, D, C, A, x(\left(\begin{array}{c} X(C)(sx) \\ \wedge \\ N^o(C) \\ \wedge \\ N^o(D) \\ \wedge \\ \neg(C =_{\subset} D) \end{array} \right) & \Rightarrow & X(C)(\mathsf{val}\ x\ \mathsf{as}\ D : A\,!\ (s(x)))) & \Rightarrow \\
\\
& & X(C)(S) &
\end{array} \right)
\end{array}
$$

Et on définit les fonctions d'ajout d'un choix aux fonctions de sorte $\tau \to s_i$ par :

- $T +_\tau (C, A) = \lambda Y (Y = C \twoheadrightarrow A) \cap (Y \neq C \twoheadrightarrow T(Y))$

- $T +_{\tau\to\tau} (C, F) = \lambda Y \lambda X (Y = C \twoheadrightarrow F(X)) \cap (Y \neq C \twoheadrightarrow T(Y)(X))$

- $T +_o (C, P) = \lambda Y (Y = C \Rightarrow P) \wedge (Y \neq C \Rightarrow T(Y))$

- $T +_{\tau\to o} (C, F) = \lambda Y \lambda X (Y = C \Rightarrow F(X)) \wedge (Y \neq C \Rightarrow T(Y)(X))$

Ces fonctions sont en fait analogues aux enregistrements. Si on note $\emptyset_\tau = \lambda Y.\emptyset$ on a, par exemple, en supposant les C_i deux à deux distincts :

$$((C_1, A_1) + (C_2, A_2) + (C_3, A_3) + \emptyset_\tau)(C_i) = A_i$$

Ces fonctions vont permettre de définir un prédicat $(m, T_1, T_2, T_3, T_4) :_S S$, analogue à $m :_S S$ sauf qu'on ajoute quatre fonctions. On note :

$$\emptyset_\tau = \lambda Y.\emptyset \quad \emptyset_{\tau\to\tau} = \lambda Y.\lambda X.\emptyset \quad \emptyset_o = \lambda Y.False \quad \emptyset_{\tau\to o} = \lambda Y.\lambda X.False$$

$(m, T_1, T_2, T_3, T_4) :_S S =$

$$\begin{array}{l}
\forall X \\
\left(\begin{array}{ll}
 & X(\mathsf{end})(\emptyset_\tau)(\emptyset_{\tau\to\tau})(\emptyset_o)(\emptyset_{\tau\to o})(\mathsf{end}) \\
 & \Rightarrow \\
\forall a, A, m, s, C & \\
\left(\begin{array}{c} Xm(a)\overrightarrow{T}(a)s(a) \\ \wedge \\ a \in A \\ \wedge \\ N^o(C) \\ C \notin s(a) \end{array}\right) & \Rightarrow \quad X(\mathsf{let}\ x\ \mathsf{as}\ C = a\ m(x))((\overrightarrow{T}(a))(s(a)))(\mathsf{val}\ x\ \mathsf{as}\ C\ \mathsf{:A}\ s(x))) \\
 & \Rightarrow \\
\forall P, m, s & \\
\left(\begin{array}{c} X(m)(\overrightarrow{T})(s) \\ \wedge \\ P \end{array}\right) & \Rightarrow \quad X(m)(\overrightarrow{T})(\mathsf{fact}\ P\ !\mathsf{s})) \\
 & \Rightarrow \\
\forall A, C, m, s & \\
\left(\begin{array}{c} X(m)(\overrightarrow{T})(s(A)) \\ \wedge \\ C \notin s(A) \end{array}\right) & \Rightarrow \quad Xm(T_1 + (C = A))T_2T_3T_4(\mathsf{def}_\tau\ t\ \mathsf{as}\ C = A\,!\ (\mathsf{s}\ \mathsf{t}))) \\
 & \Rightarrow \\
\vdots & \\
 & \Rightarrow \\
\forall A, C, m, s & \\
\left(\begin{array}{c} X(m)(\overrightarrow{T})(s(A)) \\ \wedge \\ C \notin s(A) \end{array}\right) & \Rightarrow \quad XmT_1T_2T_3(T_4 + (C = A))(\mathsf{def}_{\tau\to o}\ t\ \mathsf{as}\ C = A\,!\ (\mathsf{s}\ \mathsf{t}))) \\
 & \Rightarrow \\
 & X(m)(\overrightarrow{T})(S)
\end{array}\right)
\end{array}$$

Ce prédicat sert simplement à instancier les types "concrets" à l'aide des quatre fonctions T_i.
On montre le lemme suivant :

Lemme 108

$$\vdash \forall m, \overrightarrow{T}, S((m, \overrightarrow{T}) :_S S \Rightarrow m \in SC\lambda x.S(\overrightarrow{T})(x)$$

preuve: Par induction :
– Le cas vide est facile.
– Le cas let nécessite d'établir la propriété suivante : Pour tout enregistrement $<m>$:

$$\vdash \forall C, S, a, A(C \notin S \Rightarrow a \in A \Rightarrow S(\overrightarrow{T})(m) \Rightarrow S(\overrightarrow{T})(<C = a; m > m))$$

Si $<r>$ est un enregistrement, et si $C \neq D$, alors :

$$\mathcal{S}(a) \vdash <C = a; r>.D = <r>.D$$

Cela se prouve par induction sur le prédicat $C \notin S$, et en utilisant la propriété suivante : Si $<r>$ est un enregistrement, alors :

$$\vdash \forall C, D, a, A(N^o(C) \Rightarrow N^o(D) \Rightarrow C \neq D \Rightarrow a \in A \Rightarrow <C = a; r>.D = <r>.D)$$

qui se prouve par induction sur la structure de r.
– Les cas def_s se prouvent en utilisant les faits suivant :

$$\vdash \forall m, C, T, A^s, S(C \notin S \Rightarrow S\overrightarrow{T}(m) \Rightarrow S(\overrightarrow{T} +_s (C = A))(m))$$

qui se prouve par induction sur $C \notin S$, en notant $\overrightarrow{T} +_s (C = A)$ l'ajout d'un nouveau champs à la composante adéquate.

□

On peut maintenant prouver la proposition 107 :
preuve: On montre d'abord :

$$\vdash \forall m, S(m :_S S \Rightarrow \exists \overrightarrow{T}(m, \overrightarrow{T}) :_S S)$$

Ce qui est immédiat par induction.
Maintenant, il suffit de dériver $m \in \mathsf{Struct}\ S$ à partir de $\exists \overrightarrow{T}((m, \overrightarrow{T}) :_S S)$. Par le lemme précédent, si $(m, \overrightarrow{T}) :_S S$, alors $m \in SC\lambda x.S(\overrightarrow{T})(x)$, d'où il vient immédiatement $m \in \mathsf{Some}\ \overrightarrow{T}SC\lambda x.S(\overrightarrow{T})(x)$, ce qui est le résultat voulu. □

9.3.4 Règles d'élimination

On traduit :

$$\widetilde{(m :_{s_i} S)}.n \quad = \quad ((m :^{s_1,\ldots,s_4} \lambda\overrightarrow{T}SC\lambda x.\widetilde{S}(\overrightarrow{T})(x)).i)(n)$$

On obtient donc, en appliquant les règles d'élimination des types abstraits, la règle suivante :

$$\frac{\Gamma \vdash m \in \mathsf{Struct}\ S}{\Gamma \vdash S\overrightarrow{(m :_{s_i} S)}(m)}$$

– Projection d'une définition : On suppose :

$$(\mathsf{def}_s\ x\ \mathsf{as}\ n = A\,!\ S(x))\overrightarrow{(m :_{s_i} S)}(m)$$

donc par traduction :

$$(m :_s S)(n) = A \quad \wedge \quad S(A)(\overrightarrow{(m :_{s_i} S)})(m))$$

d'où :

$$(m :_s S)(n) = A \quad \wedge \quad S((m :_s S)(n))(\overrightarrow{(m :_{s_i} S)})(m))$$

– Projection : On suppose

$$(\mathsf{val}\ x\ \mathsf{as}\ l : A\,!\ S(x))(\overrightarrow{T})(m)$$

donc par traduction :

$$m.l \in A \quad \wedge \quad S(m.l)(\overrightarrow{T})(m)$$

– Projection d'un fait : On suppose

$$(\mathsf{fact}\ P\,!\ S)(\overrightarrow{T})(m)$$

donc par traduction :

$$P \wedge S(\overrightarrow{T})(m)$$

Ceci termine la validation des règles.

9.3.5 Exemple

Nous prenons la suite de l'exemple vu précédemment avec les booléens pour voir comment cela fonctionne. On suppose que l'on a un programme p qui a le type traduit :

$$\vdash p \in \mathsf{Some}\ T_1, \ldots, T_4.SC\lambda m. \left(\begin{array}{c} T_1(\mathsf{bool}) = \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ \mathsf{m.true} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ \mathsf{m.false} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ \mathsf{m.ifthenelse} \in \forall A(\ \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow A \Rightarrow A \Rightarrow A) \\ \wedge \\ \forall A \Pi^o x, y : A\ \mathsf{m.ifthenelse\ m.true\ x\ y} = \mathsf{x} \end{array} \right)$$

On en déduit, par la règle d'élimination de Some :

$$\vdash p \in SC\lambda m. \left(\begin{array}{c} (p : \lambda \overline{T} SC \ldots).1(\mathsf{bool}) = \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ \mathsf{m.true} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ \mathsf{m.false} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ \mathsf{m.ifthenelse} \in \forall A(\ \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow A \Rightarrow A \Rightarrow A) \\ \wedge \\ \forall A \Pi^o x, y : A\ \mathsf{m.ifthenelse\ m.true\ x\ y} = \mathsf{x} \end{array} \right)$$

Puis, par le fait 91 page 105 :

$$\vdash \left(\begin{array}{c} (p : \lambda \overline{T} \mathsf{Self}\ \ldots).1(\mathsf{bool}) = \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ p.\mathsf{true} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ p.\mathsf{false} \in \mathsf{TRUE} \mid \mathsf{FALSE} \\ \wedge \\ p.\mathsf{ifthenelse} \in \forall A(\ \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow A \Rightarrow A \Rightarrow A) \\ \wedge \\ \forall A \Pi^o x, y : A\ p.\mathsf{ifthenelse\ } p.\mathsf{true\ x\ y} = \mathsf{x} \end{array} \right)$$

Ce qui donne bien les règles de projection souhaitées.

9.4 Validation des règles d'évaluation

La validation des règles d'évaluation repose sur quatre faits :
- La notion de type de valeur permet de cerner la forme d'un programme clos habitant ce type de valeur
- La progression assure qu'un terme clos typé soit se réduit soit est une valeur
- La réduction du sujet assure que le réduit d'un terme typé a le même type
- La traduction assure que si un programme se réduit, alors son traduit se réduit à gauche sur le traduit du terme réduit, et par la traduction de $NORM$, cela assure que si un programme a pour type $NORM$ et que sa réduction ne bloque pas, alors elle s'arrête.

La combinaison des trois premiers faits assure que la réduction ne bloque pas, et avec le quatrième, cela permet d'assurer la terminaison des programmes de type $NORM$.

9.4.1 Valeurs et types de valeurs

Définition 59 *Une valeur est :*
- *Soit une fonction :* $t = \mathsf{fun}\ x \rightarrow u$
- *Soit une paire :* $t = (a\ ,\ b)$
- *Soit un constructeur :* $t = C$
- *Soit un constructeur appliqué à une valeur :* $t = C[v]$
- *Soit un enregistrement :* $t = <R>$
- *Soit un ajout de champs :* $t = [l = a] + p$

Définition 60 *Un type de valeur est :*
- *Soit un produit fonctionnel :* $T = \Pi^{\tau} x : A.B$
- *Soit un produit cartésien :* $T = A \times B$
- *Soit un type somme.*
- *Soit un type inductif* $\mu X.F(X)$*, avec* $F[X := \mu X.F(X)]$ *un type de valeur.*
- *Soit un type co-inductif* $\nu X.F(X)$*, avec* $F[X := \mu X.F(X)]$ *un type de valeur.*
- *Soit un type enregistrement :* $T = \{R\}$
- *Soit de la forme* $\forall^{\tau}_{s} x_i.T$*, avec* T *un type de valeur.*
- *Soit de la forme* $\{x : A; P(x)\}$ *avec* A *un type de valeur.*

Définition 61 *Un programme* t *est un représentant canonique d'un type de valeur* T *si :*
- $t = \mathsf{fun}\ x \rightarrow u$ *et* $T = \Pi^{\tau} x : A.B$
- $t = (a,b)$ *et* $T = A \times B$
- $t = C$ *et* T *est un type somme qui contient* C
- $t = C[a]$ *et* T *est un type somme qui contient* $C\ \mathsf{of}\ A$
- $T = \mu X.F(X)$ *ou* $\nu X.F(X)$ *et* t *est un représentant de* $F(X := T)$.
- $t = <L>$ *ou* $t = [l = a] + p$ *et* $T = \{R\}$
- $T = \{x : A; P(x)\}$ *et* t *est un représentant canonique de* A.

Le lemme sur les valeurs et les types de valeurs est le suivant :

Lemme 109 (lemme des valeurs) *Si $\vdash t : T$, si T est un type de valeur, et si t ne se réduit pas, alors t est un représentant canonique de T*

preuve: Par induction, en regardant le dernière règle utilisée : on fait la totalité de la preuve pour Π, les autres cas sont analogues, c'est à dire soit on a une règle d'introduction adéquate, soit on a une règle d'élimination qui se traite par induction, soit on a une règle de sous-typage qui ne modifie pas la forme du type, soit on a une règle du schéma de compréhension qui ne modifie pas la forme du type :
- $T = \Pi x : A.B$: on regarde les règles possibles vu la forme du type.
  - Π_I^τ, c'est bon.
  - Π_E^τ : par HI, on aurait $u = \mathsf{fun}\ \ldots$: contradiction
  - $\forall_I^\tau$, $\forall_E^\tau$, : par HI
  - $\subset$: par HI et par le lemme de sous-typage précédent.
  - SC_E^τ : par HI.
  - $\times EG$, $\times ED$: par HI, on a $p = (a,\ b)$, et donc contradiction (t se réduit).
  - $\_$, $Constr i, j_E$: par HI, on a p est de la forme d'un constructeur (éventuellement appliqué à un argument) : t se réduit, contradiction.
  - $\_$, $Constr i, j_E$: par HI, on a p est de la forme d'un constructeur (éventuellement appliqué à un argument) : t se réduit, contradiction.
  - Les règles avec l'opérateur de récursion donnent un terme qui se réduit : contradiction.
  - $Coinduc_\tau$: par HI.
  - $\{\}_E$: par HI, $p.l$ se réduit : contradiction.
- $T = A \times B$: de même.
- type somme : de même pour les règles considérées. Si c'est une règle de sous-typage de types inductifs ou coinductifs, le type de la prémisse a la même forme que T.
- Type enregistrement : de même.
- Schéma de compréhension : si la dernière règle est SC_I, alors on conclut par HI.

□

9.4.2 Progression

Lemme 110 *Si $\vdash t : A$, alors :*
- *Soit t est une valeur*
- *Soit il existe t' tel que $t \succ t'$*

preuve: Par induction sur la dérivation, en regardant la dernière règle utilisée (on supprime les règles d'introduction qui introduisent directement des valeurs) :
- Ax^τ : impossible.
- Π_E^τ : Si u se réduit, alors c'est bon. Sinon, d'après le lemme 109, u est de la forme $\mathsf{fun}\ x \to p$, donc $(u\ v)$ se réduit.
- $\forall_I^\tau$, $\forall_E^\tau$: par HI.
- SC_I, SC_E, $\subset$: par HI.
- $\times EG$, $\times ED$: si p se réduit, alors c'est bon. Sinon, d'après le lemme 109, p est de la forme (a,b), donc t se réduit.

- $\_$, $Constri, j_E$: si p se réduit, alors c'est bon. Sinon, d'après le lemme 109, p est de la forme C ou $C[a]$, donc t se réduit.
- Règles avec le récurseur : se réduit.
- $Coinduc_\tau$: par HI.
- $\{\}_E$: si p se réduit, c'est bon. Sinon, d'après le lemme 109, c'est un représentant canonique du type enregistrement, donc $p.l$ se réduit.

□

9.4.3 Réduction du sujet

Le résultat repose sur cinq lemmes techniques.

Lemme 111 (lemme de substitution) *Si* $\Gamma, x : A \vdash p : B(x)$ *et* $\Gamma \vdash v : A$, *alors* $\Gamma \vdash p[x := v] : B[x := v]$

preuve: On montre par induction sur la dérivation que :

$$\begin{cases} \text{si } \Gamma, x : A, \Delta(x) \vdash p : B(x), \text{ alors } \Gamma, \Delta(v) \vdash p[x := v] : B[x := v] \\ \text{si } \Gamma, x : A, \Delta(x) \vdash P(x), \text{ alors } \Gamma, \Delta(v) \vdash P[x := v] \\ \text{si } \Gamma, x : A, \Delta(x) \vdash t : s, \text{ alors } \Gamma, \Delta(v) \vdash t[x := v] : s \end{cases}$$

- Pour les constructions de termes, c'est clair. La règle $Prog2\pi$ se déduit par HI.
- Les règles algorithmiques sont immédiates par induction.
- Les règles logiques également.
- Les règles de sous-typage également.
- Les règles qui font intervenir la réduction viennent du fait que si $p \succ p'$, alors $p[x := v] \succ p'[x := v]$.

□

Le lemme suivant sert à contrôler le sous-typage des types de données :

Lemme 112 (sous-typage des types de données)
1. *Si* $\Gamma \vdash \Pi x : A'.B' \subset \Pi x : A'.B'$, *alors* $\Gamma, x : A' \vdash B \subset B'$ *et* $\Gamma \vdash A' \subset A$.
2. *Si* $\Gamma \vdash A \times B \subset A' \times B'$, *alors* $\Gamma \vdash A \subset A'$ *et* $\Gamma \vdash B \subset B'$.
3. *Si* $\Gamma \vdash C$ *of* $A \subset C$ *of* A', *alors* $\Gamma \vdash A \subset A'$
4. *Si* $\Gamma \vdash \{L_1; l : A; L_2\} \subset \{R_1; l : A'; R_2\}$, *alors* $\Gamma \vdash A \subset A'$

preuve: On regarde la dernière règle de sous-typage utilisée : dans tous les cas, si c'est la règle $\subset_{Refl}$, ou si c'est la règle qui utilise comme prémisse la conclusion voulue, c'est clair. On examine les autres cas, par induction sur la longueur de la dérivation de sous-typage en regardant la dernière règle utilisée, qui est forcément $\subset_{Trans}$:

1. On a $\Gamma \vdash \Pi x : A.B \subset \Pi x : C.D$ et $\Gamma \vdash \Pi x : C.D \subset \Pi x : A'.B'$: par HI, on a $\Gamma \vdash C \subset A$ et $\Gamma \vdash A' \subset C$. Par transitivité, on a donc $\Gamma \vdash A' \subset A$. Un lemme facile montre que si $\Gamma, x : U \vdash t : V$ et $\Gamma \vdash W \subset U$, alors $\Gamma, x : W \vdash t : V$. Par HI, on a $\Gamma, x : C \vdash B \subset D$, on a donc $\Gamma, x : A' \vdash B \subset D$. Par HI, on a aussi et $\Gamma, x : A' \vdash D \subset B'$. On a donc $\Gamma, x : A' \vdash B \subset B'$

Les autres cas sont immédiats par induction. □

Lemme 113 (Elimination des $\forall$-coupures et des SC-coupures) *Si $\Gamma \vdash t : A$, alors il existe une dérivation de ce séquent dans laquelle :*
– *une règle $\forall_I^\tau$ n'est jamais suivie d'une règle $\forall_E^\tau$.*
– *une règle SC_I n'est jamais suivie d'une règle SC_E.*

preuve: On considère une dérivation de longueur minimale du séquent $\Gamma \vdash t : A$. Si elle contient une $\forall$-coupure :
– à l'étape k, on a $\Gamma, x : s \vdash t : A$
– à l'étape $k+1$, on a $\Gamma \vdash t : \forall_s^\tau x.A(x)$
– à l'étape $k+2$, on a $\Gamma \vdash t : A(x := v)$
Or, si $\Gamma, x : s \vdash t : A$, on a $\Gamma \vdash t : A[x := v]$ avec une dérivation de même longueur (k), ce qui est absurde.
Idem avec SC.

□

Lemme 114 *Si $\Gamma \vdash t : A$, alors il existe une dérivation de même conclusion dans laquelle :*

1. *Une règle $\subset$ n'est jamais suivie d'une règle $\forall_E^\tau$*
2. *Une règle $\subset$ n'est jamais suivie d'une règle SC_E*

preuve: Par récurrence sur la somme des hauteurs des dérivations de $\subset$ concernées :

1. C'est facile : il n'y a pas d'autre règle de sous-typage autre que la réflexivité qui permette de sous-typer un type qui commence par $\forall$: par suite, on peut éliminer cette instance de $\subset$.
2. On regarde les deux règles possibles de sous-typage du schéma de compréhension :
   – Si c'est $SC - CR \subset$, on peut faire permuter les deux règles, ce qui fait décroître la hauteur de la dérivation de sous-typage.
   – Si c'est $SC - CR \Rightarrow^o$, on peut supprimer cette instance.

□

Lemme 115 *1. Si $\Gamma \vdash \mathsf{fun}\ x \to p : \Pi x : A.B(x)$, alors $\Gamma, x : A \vdash p : B(x)$.*
2. Si $\Gamma \vdash (a, b) : A \times B$, alors $\Gamma \vdash a : A$
3. Si $\Gamma \vdash C[a] : C\ \mathsf{of}\ A$, alors $\Gamma \vdash a : A$
4. Si $\Gamma \vdash <R_1; l = aR_2> : \{L_1; l : A; L_2\}$ alors $a : A$
5. Si $\Gamma \vdash [l = a] + p : \{l : A\}$ alors $a : A$

preuve:
On considère une dérivation dépourvue de $\forall$-coupures et de SC-coupures, et où les règles $\subset$ ne sont pas suivies de règles $\forall_E$ ou SC_E : il en existe d'après les deux lemmes précédents. Par induction sur la longueur de cette dérivation, en regardant la dernière règle utilisée :

1. Pour Π :
   - Π_I^τ : c'est clair.
   - $\forall_E^\tau$: impossible car par les propriétés de la dérivation, la règle précédente ne peut pas être $\forall_I$, ni $\subset$: la seule règle possible est donc SC_E. De même, la règle précédant SC_E est nécessairement $\forall_E$: toutes ces règles ayant dans la prémisse un type commençant par $\forall$ ou $\{$, il est impossible de rencontrer une règle Π_I auparavant : c'est absurde.
   - SC_E : comme pour $\forall_E$.
   - $\subset$: par le lemme de sous-typage des types de données, on a une dérivation plus courte de $\Gamma \vdash \mathsf{fun}\ x \to p : \Pi x : A'.B'$, et donc par HI $\Gamma, x : A' \vdash p : B'(x)$. On a de plus $\Gamma \vdash A \subset A'$. On a donc $\Gamma, x : A \vdash p : B'(x)$. Par le lemme précédent encore, on a $\Gamma, x : A \vdash B'(x) \subset B(x)$, donc $\Gamma, x : A \vdash p : B(x)$.
2. Pour $\times$:
   - Si la dernière règle est $\times_I$, c'est clair.
   - Ce ne peut être $\forall_E$ ou SC_E par le même argument que précédemment.
   - Si c'est $\subset$, par le lemme de sous-typage précédent, on a une dérivation plus courte de $\Gamma \vdash (a, b) : A' \times B'$, avec $\Gamma \vdash A' \subset A$. Par HI, on a $\Gamma \vdash a : A'$, et donc $\Gamma \vdash a : A$
3. Pour les constructeurs :
   - Si la dernière règle est $Constr1_I$, c'est clair.
   - Ce ne peut être $\forall_E$ ou SC_E par le même argument que précédemment.
   - Si c'est $\subset$, par le lemme de sous-typage précédent, on a une dérivation plus courte de $\Gamma \vdash C[A']$, avec $\Gamma \vdash A' \subset A$. Par HI, on a $\Gamma \vdash a : A'$, et donc $\Gamma \vdash a : A$.
4. Pour les enregistrements $<R; l = a; R'>$:
   - Si la dernière règle est $\{\}_I$, c'est clair.
   - Ce ne peut être $\forall_E$ ou SC_E par le même argument que précédemment.
   - Si c'est $\subset$, par le lemme de sous-typage précédent, on a soit une dérivation plus courte de $\Gamma \vdash <R_1; l = a; R_2> : \{L_1; l : A; L_2\}$, soit une dérivation plus courte de $\Gamma \vdash <R_1; l = a; R_2> : \{L_1; l : A'; L_2\}$, avec $\Gamma \vdash A' \subset A$. Dans les deux cas, par HI, on a $\Gamma \vdash a : A'$, et donc $\Gamma \vdash a : A$.
5. Pour les ajouts de champs :
   - Si la dernière règle est $\{\}_+$, c'est clair.
   - Ce ne peut être $\forall_E$ ou SC_E par le même argument que précédemment.
   - Si c'est $\subset$, par le lemme de sous-typage précédent, on a soit une dérivation plus courte de $\Gamma \vdash [l = a] + p : \{l : A; L\}$, soit une dérivation plus courte de $\Gamma \vdash [l = a] + p : \{l : A; L\}$, avec $\Gamma \vdash A' \subset A$. Dans les deux cas, par HI, on a $\Gamma \vdash a : A'$, et donc $\Gamma \vdash a : A$.

□

On montre maintenant la réduction du sujet :
preuve:
- Ax^τ : ne se réduit pas
- Π_I^τ : ne se réduit pas

- Π^{τ}_{E} : si $u \succ u'$, par HI, $\Gamma \vdash u' : \Pi^{\tau} x : A.B$, donc c'est bon. Sinon, par le lemme des valeurs, $u = \mathsf{fun}\ x \to p$, et donc par le lemme précédent, $\Gamma, x : A \vdash p : B$, et par le lemme de substitution, on a $\Gamma \vdash p[x := v] : B[x := v]$.
- règles sans contenu algorithmique : elles ne modifient pas le terme, donc c'est immédiat par HI.
- $\times_I$: ne se réduit pas.
- $\times_{EG}$, $\times_{ED}$: si p se réduit, on a le résultat par HI. Sinon, si $\mathsf{fst}p$ se réduit, c'est que $p = (a,b)$. Par le lemme précédent, on a $\Gamma \vdash a : A$.
- $Contr0_I$: ne se réduit pas
- $Constr1_I$: immédiat par HI.
- _ : si p se réduit, c'est bon par HI. Sinon, le programme se réduit en f, et c'est clair.
- $Constr01_E$: si p se réduit, c'est clair. Sinon, $p = C$ et c'est clair par la prémisse droite.
- $Constr11_E$: si p se réduit, c'est clair. Sinon $p = C[a]$. Par le lemme précédent, $\Gamma \vdash a : A$, et par le lemme de substitution, $\Gamma \vdash f[x := a] : B(C[a])$.
- $Constr02_E$: si p se réduit, c'est bon par HI. Sinon, il est de la forme D ou $D[a]$. On montre le lemme suivant par induction :
  si $\Gamma \vdash D* : C** \mid S$, alors $C = D$ ou $\Gamma \vdash D : S$.
  (avec $*=$ rien ou $[a]$ et $**=$ rien ou A)
  Si $D = C$, on a $f : B(C)$. Sinon, $p : S$, se traite avec le lemme de substitution.
- $Constr12_E$: comme $Constr02_E$.
- Rec_{μ} : on montre le lemme :
  Si $\Gamma, A : \tau, f : \Pi^{\tau} x : A.B(x), A \subset T, \Delta(A) \vdash Concl(A)$, alors $\Gamma, f : \Pi^{\tau} x : T.B(x), \Delta(T) \vdash Concl(T)$.
  Il s'ensuit que $\mathsf{fun}\ f \to t : \Pi^{\tau} f : (\Pi^{\tau} x : \mu X.F(X).B(x)).(\Pi^{\tau} x : F(\mu X.F(X)).B(x))$, et donc $(\mathsf{fun}\ f \to t\ \mathsf{rec}\ \mathsf{fun}\ f \to t\ ) : \Pi^{\tau} x : F(\mu X.F(X)).B(x))$. Par les règles de sous-typage, on a $\Pi^{\tau} x : F(\mu X.F(X)).B(x)) \subset \Pi^{\tau} x : \mu X.F(X).B(x))$, donc c'est bon.
- $BFRec_{\mu}$: Il suffit de montrer que

$$\mathsf{fun}\ f \to \mathsf{fun}\ y \to t : \Pi f : T.T$$

  avec $T = \Pi x : \mu X.F(X).B(x)$. Pour cela, on montre en fait

$$\Gamma, f : \Pi x : \{\mu X.F(X); True\}.B(x), y : \mu X.F(X) \vdash t : B(y)$$

  En effet, on a facilement $\{x : \mu X.F(X); R(x,y)\} \subset \{\mu X.F(X); True\}$, et donc

$$\Pi x : \{\mu X.F(X); True\}.B(x) \quad \subset \quad \Pi x : \{\mu X.F(X); R(x,y)\}.B(x)$$

  Partant, on peut substituer dans toutes les feuilles de l'arbre de dérivation gauche les instances d'axiomes portant sur f par un axiome et une règle de sous-typage. Il vient donc la conclusion.
- $Coinduc_{\tau}$: par HI.
- $\nu_{creat1,2}$: immédiates par HI et sous-typage.
- $\{\}_E$: : on suppose que $p.l$ se réduit, donc il y a deux cas à regarder :

- Soit $p = <L_1; l = a; L_2>$: dans ce cas, on a par le lemme précédent $a : A$, donc c'est bon.
- Soit $p = [l' = a] + p$:
  - Si $l = l'$, alors on a $a : A$ par le lemme précédent.
  - Sinon, on a une dérivation plus courte de $p.l : A$, donc on conclut par HI.

□

9.4.4 Traduction et conclusion

Lemme 116 *Si $p \succ p'$, alors $\widetilde{p}\ \beta_l\ \widetilde{p'}$.*

preuve: Les règles *Pop* de la machine présentée en annexe B reflètent simplement les règles de réduction gauche - en fait, de tête - des termes traduits et correspondent aux étapes de réduction présentées pour $\succ$. On regarde les cas de $\succ$:

1. Réduction λ : identique.
2. Récurseur : identique.
3. Première projection : identique.
4. Seconde projection : identique.
5. Projection sur un programme qui n'est pas une paire : par la réduction du sujet, le programme est typable donc son traduit ne se réduit en une abstraction que lorsqu'il s'agit d'une paire. Il est bien en tête par traduction.
6. Réduction de tête pour les cases : il suffit de vérifier que le test d'égalité sur les entiers de Church se réduit en tête à un booléen.
7. Réduction de tête pour les cases (argument qui n'est pas filtré) : le test d'égalité fait venir u en tête, d'où la règle d'évaluation.
8. Projection d'enregistrement : immédiat par la traduction via les cases.
9. Projection de champs ajouté $(=)$: idem.
10. Projection de champs ajouté $(\neq)$: idem.
11. Projection sur un programme qui n'est pas un enregistrement : vient du fait que la projection est une application, et donc u se met en tête de pile.
12. Réduction sous un constructeur : Le traduit de $C[u]$ étant une paire dont le premier élément est sous forme normale, si $u\ \beta_l\ u'$, alors $C[u]\ \beta_l\ C[u']$. C'est en fait le seul endroit où l'on ait besoin de la réduction gauche.

□

Proposition 117 *Si $\vdash t : NORM$, alors l'évaluation de t s'arrête.*

preuve: D'après les lemmes de progression et de valeurs, si $\vdash t : NORM$, alors t est une valeur ou il existe t' tel que $t \succ t'$. Par le lemme de préservation du sujet, on a donc $\vdash t' : NORM$, ce qui implique que la machine ne bloque pas. De plus, par traduction, on a $\vdash \widetilde{t} \subset \widetilde{NORM}$, ce qui implique que t est normalisable. Par le lemme précédent, on a $\widetilde{t}\ \beta_l\ \widetilde{t'}$. La réduction gauche étant une stratégie normalisante, la réduction gauche de $\widetilde{t}$ s'arrête, ce qui entraîne donc que la réduction de t s'arrête. □

9.5 Conclusion

On a vu dans ce chapitre la correction du langage : celle-ci justifie partiellement les règles des chapitres précédents, car du point de vue de l'évaluation, nous ne nous sommes pas penchés sur les types abstraits et les modules. Nous pouvons retenir cependant que la correction repose sur deux piliers :

- La correction de la traduction : celle-ci est un guide pour l'écriture correcte des règles.
- La correction au niveau du langage "du haut", qui permet la compilation du langage vers d'autres cibles que le λ-calcul, du moment qu'on respecte l'évaluation.

Chapitre 10

Conclusion

Ce travail a permis de confirmer l'hypothèse selon laquelle le système ST est un formalisme suffisamment riche pour exprimer de très nombreux types de données, et avons montré qu'il permettait également de contrôler la terminaison des programmes. Cependant, le travail serait à compléter dans plusieurs directions, tant théoriques que pratique.

10.1 Points théoriques

10.1.1 Logique classique avec sous-typage et omission de contenu algorithmique

Nous avons présenté dans [51] une utilisation possible de l'omission de contenu non-algorithmique dans le cadre du λC calcul (voir les règles dans la figure 10.1.1). Les termes sont les mêmes que dans le système ST, excepté les formules de sous-typage, et on se donne en plus la loi de Pierce. La sémantique est adaptée à partir de celle de [33].
On se demande quels sont les axiomes de sous-typage que l'on peut ajouter au système tout en restant consistant (le résultat 61 vu au chapitre 5 montre qu'on ne peut pas tout ajouter !).

10.1.2 ST comme sous-système de λC et pouvoir expressif

Nous présentons ici rapidement en quoi les termes du systèmes ST peuvent être vus comme obtenus par "pruning" à partir de λC termes.
Nous pensons pouvoir montrer que le pouvoir du système ST propositionnel, muni de tous ses axiomes, est le même que celui du λC calcul, dans le sens suivant :

Théorème 118 (Conjecture) *Il existe une transformation $.^*$ des types de ST dans les types de la logique classique d'ordre supérieur, et une fonction d'effacement prune des λC-termes dans les termes du λ-calcul pur, tels que : Pour tout terme t et tout type A de ST tel que $\vdash t : A$, il existe un λC terme t' tel $\vdash t' : A^*$ et $t = prune(t')$, et de plus, si t est normalisable, alors t' aussi et $FN(t) = FN(t')$.*

Axiome $\dfrac{}{\Gamma, x : A \vdash x : A}$	Axiome' : $\dfrac{}{\Gamma, P \vdash P}$
$\Rightarrow^\tau$ - Intro : $\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$	$\Rightarrow^o$ - Intro : $\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \to' Q}$
$\Rightarrow^\tau$ - Elim : $\dfrac{\Gamma \vdash u : A \Rightarrow^\tau B \qquad \Gamma \vdash v : A}{\Gamma \vdash u(v) : B}$	$\Rightarrow^o$ - Elim : $\dfrac{\Gamma \vdash P \Rightarrow^o Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$
$\forall^\tau$ - Intro : $\dfrac{\Gamma, x : s \vdash t : A}{\Gamma \vdash t : \forall^\tau_s x.A}$	$\forall^o$ - Intro : $\dfrac{\Gamma, x : s \vdash P}{\Gamma \vdash \forall^o_s x.P}$
$\forall^\tau$ - Elim $\dfrac{\Gamma \vdash t : \forall^\tau_s x.A \qquad \Gamma \vdash v : s}{\Gamma \vdash t : A[x := v]}$	$\forall^o$ - Elim : $\dfrac{\Gamma \vdash \forall^o_s x.P \qquad \Gamma \vdash v : s}{\Gamma \vdash P[x := v]}$
$\twoheadrightarrow$- Intro : $\dfrac{\Gamma, P \vdash t : A}{\Gamma \vdash t : P \twoheadrightarrow A}$	$\twoheadrightarrow$- Elim : $\dfrac{\Gamma \vdash t : P \twoheadrightarrow A \qquad \Gamma \vdash P}{\Gamma \vdash t : A}$
Loi de Pierce : $\dfrac{}{\Gamma \vdash C : \forall^\tau A \forall^\tau B(((A \to B) \to A) \to A)}$	Loi de Pierce' : $\dfrac{}{\Gamma \vdash \forall^o P \forall^o Q(((P \to Q) \to P) \to P)}$

FIG. 10.1 – Règles de λC avec omission de contenu algorithmique

Ce qui entraîne en particulier le corollaire suivant :

Corollaire 119 *Si* $\vdash t : \mathbb{N} \Rightarrow \mathbb{N}$*, alors la fonction récursive que* t *représente est prouvablement totale dans l'arithmétique classique du second ordre.*

La preuve se déroule en trois étapes :
- On définit la traduction des types
- On interprète les axiomes et on rajoute du contenu algorithmique aux règles de dérivation qui n'en ont pas.
- On étudie les transformations des dérivations ainsi induites

Nous allons dans une première section brièvement décrire notre langage-cible qui est le λC-calcul, puis dans une seconde section, nous donnerons la preuve du résultat ci-dessus.

10.1.3 λC-calcul d'ordre supérieur

Syntaxe et évaluation

Il s'agit d'une variante du λC-calcul qui étend la quantification à n'importe quel ordre.

Définition 62 (Termes et types) *Les termes sont les termes simplement typés, les types étant formés à partir d'un unique atome* τ*, et le calcul étant muni des constantes suivantes :*
- $\Rightarrow : \tau \to \tau \to \tau$
- $\forall^s : (s \to \tau) \to \tau$ *pour chaque sorte* s

La constante $\bot_\tau$ *est définie par* $\forall X.X$*.*

Définition 63 (Piles et λC termes) *L'ensemble des termes et des piles est défini par induction mutuelle par :*

$$\begin{array}{rcl} T & = & V \mid \mathcal{C} \mid (T)T \mid k_\Pi \\ \Pi & = & \epsilon \mid T.\Pi \end{array}$$

On appelle processus un couple formé d'un terme et d'une pile. Un processus t, π *sera noté* $t \star \pi$*.*

Les règles d'évaluation des processus sont les suivantes :
- push : $(u)v \star \pi \succ u \star v.\pi$
- pop : $\lambda x.t \star u.\pi \succ t[x := u] \star \pi$
- store : $\mathcal{C} \star u.\pi \succ u \star k_\pi.\pi$
- restore : $k_\pi \star t.\pi' \succ t \star \pi$

Règles

Un contexte Γ est un ensemble fini de couples $x : A$ avec A une expression de sorte τ et x une λ-variable. Les règles de dérivation sont les suivantes :

1. Axiome :

$$\frac{}{\Gamma, x : A \vdash x : A}\, Ax$$

2. Introduction de $\Rightarrow$:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B} \Rightarrow^i$$

3. Elimination de $\Rightarrow$:

$$\frac{\Gamma \vdash u : A \Rightarrow B \qquad \Gamma \vdash v : A}{\Gamma \vdash (u)v : B} \Rightarrow^e$$

4. Introduction de $\forall$ (pour chaque sorte s)

$$\frac{\Gamma \vdash t : F(X^s)}{\Gamma \vdash t : \forall^s X.F(X)}\, \forall^i$$

(Avec X^s non libre dans Γ)

5. Élimination de $\forall$ (pour chaque sorte s)

$$\frac{\Gamma \vdash t : \forall^s X F(X) \qquad U \text{ de sorte } s}{\Gamma \vdash t : F(U)}\, \forall^e$$

6. Règle classique :

$$\frac{\Gamma \vdash t : ((A \Rightarrow \bot) \Rightarrow \bot)}{\Gamma \vdash (\mathcal{C})t : A}\, Class$$

Comme on le voit, ce système est bien plus simple que ST : c'est pourquoi on le choisit pour en étudier l'expressivité. Dans la suite, on appellera l'ensemble du système ainsi décrit, le λC-calcul d'ordre supérieur, F^C_ω.

10.1.4 Traductions

Traduction des types

A chaque sorte s de ST on associe une sorte s^* en remplaçant toutes les occurrences de o par τ.

Définition 64 *A chaque expression A de ST de sorte s on associe une expression A^* de F_ω^C de sorte s^* de la façon suivante :*

$$\begin{array}{lcl}
(X^s)^* & = & X^{s^*} \\
(\Rightarrow^o)^* & = & \Rightarrow^\tau \\
(\Rightarrow^\tau)^* & = & \Rightarrow^\tau \\
(\twoheadrightarrow)^* & = & \Rightarrow^\tau \\
(\subset)^* & = & \Rightarrow^\tau \\
(\forall_s^o)^* & = & \forall^s \\
(\forall_s^\tau)^* & = & \forall^s \\
(\lambda x^s.t)^* & = & \lambda x^{s^*}.t^* \\
((u)v)^* & = & (u)^* v^*
\end{array}$$

Il est ainsi clair que la traduction $.^*$ ne modifie pas les types qui sont déjà dans F_ω^C.

Interprétation algorithmique des axiomes

On montre le fait suivant :

Fait 120 *Pour chaque axiome Ax_i de ST, il existe un terme θ_i de F_ω^C tel que $\vdash \theta_i : Ax_i^*$*

preuve: On regarde chaque axiome :

1. $Ax_1 = \forall A(A \subset A)$. $\theta_1 = \lambda x.x$ convient.
2. $Ax_2 = \forall A, B, C((A \subset B) \Rightarrow (B \subset C) \Rightarrow (A \subset C))$. $\theta_2 = \lambda x \lambda y \lambda a(y)(x)a$ convient.
3. Pour chaque sorte s : $Ax_3^s = \forall^{(s\to\tau)} B \forall^s x(\forall B \subset (Bx)).\theta_3 = \lambda x.x$ convient.
4. Pour chaque sorte s : $Ax_4 = \forall A \forall^{(s\to\tau)} B(\forall^s x(A \subset (Bx)) \Rightarrow (A \subset \forall B))$.$\theta_4 = \lambda x \lambda a(x)a$ convient.
5. $Ax_5 = \forall P, \forall A(P \Rightarrow ((P \twoheadrightarrow A) \subset A))$. $\theta_5 = \lambda x \lambda y(y)x$ convient.
6. $Ax_6 = \forall P, \forall A, B((P \Rightarrow (A \subset B)) \Rightarrow (A \subset (P \twoheadrightarrow B)))$.$\theta_6 = \lambda f \lambda a \lambda p((f)p)a$ convient.
7. $Ax_7 = \forall A, B, C, D((A \subset B) \Rightarrow (C \subset D) \Rightarrow ((B \Rightarrow C) \subset (A \Rightarrow D)))$. $\theta_7 = \lambda f \lambda g \lambda h \lambda a(g)(h)(f)a$ convient.
8. $Ax_8 = \forall P \forall A, B((P \twoheadrightarrow (A \Rightarrow B)) \subset (A \Rightarrow (P \twoheadrightarrow B)))$. $\theta_8 = \lambda f \lambda a \lambda p((f)p)a$ convient.
9. $Ax_9 = \forall A, B(\forall x(A(x) \Rightarrow B(x)) \subset (\forall x A(x) \Rightarrow \forall x B(x)))$. $\theta_9 = \lambda f \lambda a(f)a$ convient.
10. $Ax_1 0 = \forall A, B(A \subset (\bot_\tau \Rightarrow B))$. $\lambda a \lambda x x$ convient.

□

Il restera alors à valider les règles, ce qui paraît faisable.

10.1.5 Typage

Il est clair que le typage comme le sous-typage dans notre système sont indécidable. On aimerait cependant au niveau du typage, quitte à restreindre par exemple le polymorphisme, regagner une forme d'inférence de type partielle. Pour ce qui concerne la vérification de type, il serait ainsi intéressant de développer un algorithme de semi-décision qui engendrerait des obligations de preuves à partir d'une paire (programme,type), dans la même idée que l'inversion de C. Parent [39].
De même au niveau des modules, la projection est extrêmement lourde. Nous pourrions développer une notion de chemin, analogue par exemple à ce qu'a fait X.Leroy dans [35].

10.2 Points pratiques

10.2.1 Extensions des traits du langage

On aimerait ajouter à notre langage, en conservant la méthodologie (i.e. par traduction) d'autres traits des langages de programmation.

Exceptions

Les exceptions pourraient être simulées par un type analogue au type "sumor" du calcul des constructions [39] :

$$A + P = \forall X((A \Rightarrow X) \Rightarrow (P \twoheadrightarrow X) \Rightarrow X)$$

On définit :
- $\texttt{abort} = \lambda x \lambda y y$
- $\texttt{try}\ t\ \texttt{as}\ x\ \texttt{in}\ u\ \texttt{or\_else}\ v = (t \quad \lambda x u \quad v)$
- $\texttt{tilde}\ a = \lambda x \lambda y (x a)$
- $\texttt{[}f\ \ a\texttt{]} = ((a(\lambda x((f(\lambda y((\texttt{tilde}(yx))))))\texttt{abort})))\texttt{abort})$

Les règles suivantes sont facilement dérivables :

$$\frac{}{\Gamma, P \vdash \texttt{abort} : A + P} \qquad \frac{}{\Gamma, a : A \vdash \texttt{tilde}\ a : A + P}$$

$$\frac{\Gamma \vdash a : A + P \qquad \Gamma, x : A \vdash u : B \qquad \Gamma, P \vdash v : B}{\Gamma \vdash \texttt{try}\ t\ \texttt{as}\ x\ \texttt{in}\ u\ \texttt{or\_else}\ v : B}$$

$$\frac{\Gamma \vdash a : A + P \qquad \Gamma \vdash f : (A \Rightarrow B) + Q}{\Gamma \vdash \texttt{[}f\ \ a\texttt{]} : B + (P \vee Q)}$$

10.2.2 Objets

Dans la littérature sur le sujet (par exemple [14],[16],[13],[10],[45]) un objet est vu comme un enregistrement composé de valeurs et de méthodes, ces dernières pouvant être modifiées ou héritées par d'autres objets. En ayant en tête le concept d'héritage, le sous typage permet d'éviter la notion de récursivité en spécifiant simplement qu'une méthode est susceptible d'agir sur n'importe quel objet qui comporte des valeurs de même type que l'objet en question.

Définition 65 (Type d'objet) *Un type d'objet est un type de la forme suivante :*

$$\{ \quad V : Val \quad ; \quad M : Meth \quad \}$$

avec $Val = \{V_1 : T_1; \ldots; V_n : T_n\}$*, et* $Meth = \{M_1 : S_1; \ldots; M_m : S_m\}$*, Val et Meth satisfaisant les conditions suivantes : Pour tout* $i = 1, \ldots, n$*, il existe* S'_i *tel que :*

$$S_i = \forall m \Pi o : \{V : Val; M : m\}.S'_i(o, m)$$

On peut alors formaliser les notions d'héritage, de composition, etc ...dans le système ST.

10.2.3 Impératif

L'ajout de traits impératifs est également une perspective de recherche pour compléter notre langage. Cela pourrait être fait en utilisant des techniques de codage analogues à celles de J-C. Filliâtre [23], d'autant que nous disposons de toutes les notions d'enregistrement qu'il utilise.

10.2.4 A propos des constructeurs

Les constructeurs n'ont été vus en fait dans le cadre du langage de programmation que comme des constantes. Or, il est clair par la traduction qu'ils peuvent être vus comme des variables de type N (type des entiers de Church). A partir de là, il est possible d'envisager des types sommes ou des enregistrements avec des quantifications sur les constructeurs. On pourra par exemple prendre le type des entiers naturels paramétrés par des choix de constructeurs :

$$\texttt{type} \;\; Nat = \lambda Z : cons \lambda S : cons(S \neq Z) \twoheadrightarrow (\mu X(Z \mid S \;\; \texttt{of} \;\; X))$$

On pourrait aussi envisager des types enregistrements avec noms de champs abstraits :

$$Some_{cons} A Some_{cons} B\{A : T; B : S\}$$

Cela augmenterait encore la souplesse des types, permettant de construire par exemple des types de modules totalement abstraits instantiables avec plus de liberté (songer au type des groupes dont on pourrait alors changer le nom de l'élément neutre et des lois ...).

10.2.5 Évaluation

La stratégie d'évaluation en appel par nom n'est pas la plus satisfaisante au niveau efficacité de calcul. Cependant, l'appel par valeur n'est gérable qu'en présence d'arguments qui sont réellement des valeurs. On pourrait donc proposer une stratégie d'évaluation en "appel par type" qui consisterait à tagger les programmes avec des marques qui permettraient d'affirmer que ce sont des valeurs :

$$\frac{\Gamma \vdash t : A \qquad \vdash A \subset NORM}{\Gamma \vdash t_{NORM} : NORM}$$

Ceci permettrait ensuite d'effectuer de l'appel par valeur lorsque l'argument est taggé par $NORM$.
Il serait également souhaitable de donner un résultat sur l'évaluation en profondeur : nous conjecturons que le résultat de terminaison reste valable.

10.2.6 Implémentation

Au niveau de l'implémentation, notre langage est encore un prototype. Nous avons codé la traduction dans PhoX, mais cela s'avère trop lourd à manipuler. Nous avons mis une petite partie du langage du haut dans le système PhoX, mais une fois encore le système semble mal adapté : nous aimerions pouvoir développer un prouveur dédié. Pour l'évaluation, nous avons développé un parseur-évaluateur en Objective Caml, avec une interface QT. L'idée est de développer l'interface en y intégrant un prouveur, et en développant une méthodologie de développement dans l'idée de la méthode B [29] ou d'extended ML [1] pour créer une bibliothèque de programmes certifiés dans notre système. L'intérêt et la faisabilité de la chose repose cependant sur l'étude théorique du typage telle que vue ci-dessus.

10.3 Conclusion

Notre travail a éclairci quelques points :
- Au plan théorique, nous avons introduit un modèle abstrait de réalisabilité, donné une preuve de β-réduction de ST et montré l'expressivité du système (impossibilité d'obtenir la logique classique, possibilité d'exprimer la terminaison (résolubilité, normalisabilité)).
- Au plan pratique, nous avons étudié un encodage possible des types de données jusqu'au niveau des modules avec spécifications.

Nous avons également dégagé quelques pistes de recherche, espérant avoir contribué à une meilleure compréhension des objets sur lesquels se penche notre domaine de recherche.

Annexe A

Résumé des règles du système ST

Sortes :

$$\frac{}{\vdash \tau : *} \qquad \frac{}{\vdash o : *}$$

$$\frac{\vdash s_1 : * \qquad \vdash s_2 : *}{\vdash (s_1 \to s_2) : *}$$

Règles de bonne formation de contextes :

$$\frac{}{WF([])} \qquad \frac{WF(\Gamma) \quad \vdash s : * \quad x \notin V(\Gamma)}{WF(\Gamma, x : s)}$$

$$\frac{WF(\Gamma) \quad \Gamma \vdash P : o}{WF(\Gamma, P)} \qquad \frac{WF(\Gamma) \quad \Gamma \vdash A : \tau \quad x \notin V(\Gamma) \quad x \in \mathcal{V}}{WF(\Gamma, x : A)}$$

Règles de typage des termes :

$$\frac{WF(\Gamma) \quad x : s \in \Gamma}{\Gamma \vdash x : s} \qquad \frac{WF(\Gamma) \quad c : s \in Const}{\Gamma \vdash c : s}$$

$$\frac{\Gamma \vdash u : s_1 \to s_2 \quad \Gamma \vdash v : s_1 \quad \vdash s_1 \to s_2 : *}{\Gamma \vdash u(v) : s_2} \qquad \frac{\Gamma, x : s_1 \vdash t : s_2 \quad \vdash s_1 \to s_2 : *}{\Gamma \vdash \lambda x : s_1 . t : s_1 \to s_2}$$

β-règles :

$$\frac{\Gamma \vdash t : A \qquad A \succ_\beta A'}{\Gamma \vdash t : A'} \beta^\tau_{red} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A' : \tau \qquad A' \succ_\beta A}{\Gamma \vdash t : A'} \beta^\tau_{exp}$$

$$\frac{\Gamma \vdash P \qquad P \succ_\beta P'}{\Gamma \vdash P'} \beta^o_{red} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P' : o \qquad P' \succ_\beta P}{\Gamma \vdash P'} \beta^o_{exp}$$

Règles logiques :

$$\frac{WF(\Gamma) \qquad P \in \Gamma}{\Gamma \vdash P}\, Ax^o$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow^o Q} \Rightarrow^o_I \qquad \frac{\Gamma \vdash P \Rightarrow^o Q \qquad \Gamma \vdash P}{\Gamma \vdash Q} \Rightarrow^o_E$$

$$\frac{\Gamma, x : s \vdash Q}{\Gamma \vdash \forall^o_s x\, Q} \forall^o_I \qquad \frac{\Gamma \vdash \forall^o_s Q \qquad \Gamma \vdash t : s}{\Gamma \vdash Q(t)} \forall^o_E$$

Règles de typage :

$$\frac{WF(\Gamma) \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\, Ax^\tau$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\ \ x\ \ \texttt{->}\ \ t : A \Rightarrow^\tau B} \Rightarrow^\tau_I \qquad \frac{\Gamma \vdash u : A \Rightarrow^\tau B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\ u\ \ v\ ) : B} \Rightarrow^\tau_E$$

$$\frac{\Gamma, X : s \vdash t : A}{\Gamma \vdash t : \forall^\tau_s X.A} \forall^s_I \qquad \frac{\Gamma \vdash t : \forall^\tau_s A \qquad \Gamma \vdash v : s}{\Gamma \vdash t : A(v)} \forall^s_E$$

Règles mixtes :

$$\frac{\Gamma \vdash A \subset B \qquad \Gamma \vdash t : A}{\Gamma \vdash t : B} \subset$$

$$\frac{\Gamma, P \vdash t : A}{\Gamma \vdash t : P \twoheadrightarrow A} \twoheadrightarrow_I \qquad \frac{\Gamma \vdash t : P \twoheadrightarrow A \qquad \Gamma \vdash P}{\Gamma \vdash t : A} \twoheadrightarrow_E$$

$$\frac{\Gamma, x : A \vdash A \subset B}{\Gamma \vdash A \subset B}\, Renf \qquad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash A' \subset A}{\Gamma, x : A' \vdash t : B}\, Coup$$

$$\frac{\Gamma \vdash t : (P \twoheadrightarrow \bot^\tau) \Rightarrow \bot^\tau}{\Gamma \vdash P}\, RC \qquad \frac{\Gamma, y : s, x : F(y) \vdash t : A \qquad \Gamma \vdash A : \tau}{\Gamma, x : \cup^s y F(y) \vdash t : A}\, UG$$

Axiomes :

$$\frac{WF(\Gamma) \qquad P \in \mathcal{A}}{\Gamma \vdash \Gamma \vdash P}\, Axiome$$

1. Axiomes de base
   - Réflexivité : $\forall A(A \subset A)$
   - Transitivité : $\forall A, B, C((A \subset B) \Rightarrow (B \subset C) \Rightarrow (A \subset C))$
   - Borne inférieure (pour chaque sorte s) :
     (a) Minorant : $\forall^o_{(s\to\tau)} B \forall^o_s x(\forall^\tau_s B \subset B(x))$
     (b) Plus grand Minorant : $\forall^o_\tau A \forall^o_{(s\to\tau)} B(\forall^o_s x(A \subset B(x)) \Rightarrow (A \subset \forall^\tau_s B))$
   - Contra- et Co-variance de $\Rightarrow^\tau$:
     $\forall A, B, C, D((A \subset B) \Rightarrow (C \subset D) \Rightarrow ((B \Rightarrow^\tau C) \subset (A \Rightarrow^\tau D)))$
   - Flèche spéciale :
     (a) Gauche : $\forall P, \forall A(P \Rightarrow ((P \twoheadrightarrow A) \subset A))$
     (b) Droit : $\forall P, \forall A, B((P \Rightarrow (A \subset B)) \Rightarrow (A \subset (P \twoheadrightarrow B)))$
     (c) Permutation : $\forall P \forall A, B((P \twoheadrightarrow (A \Rightarrow B)) \subset (A \Rightarrow (P \twoheadrightarrow B)))$
   - Axiome de Mitchell :
     $$\forall A, B(\forall x(A(x) \Rightarrow B(x)) \subset (\forall x A(x) \Rightarrow \forall x B(x)))$$
2. Axiomes avec opérateurs
   - Le faux est vide : $\forall A, B(A \subset (\bot^\tau \Rightarrow B))$
   - Atomiticité : $\forall A(A \subset \cup X(X \upharpoonright (S(X) \wedge X \subset A)))$
   - Complément : $\forall A, B(A \subset B \cup B^c)$
   - Axiomes de l'application :
     (a) Continuité à droite : $\forall A, B(A[\cup x B(x)] \subset \cup x(A[B(x)]))$
     (b) Conservation des singletons : $\forall A, B(S(A) \Rightarrow S(B) \Rightarrow S(A[B]))$
     (c) Extensionnalité : $\forall A, B(S(B) \Rightarrow \forall X(A[X] \subset B[X]) \Rightarrow (A \subset B))$
3. Axiome de non trivialité :
   $$(\neg TRIV) \qquad \exists X, Y((X \neq Y) \wedge \mathcal{S}(X) \wedge \mathcal{S}(Y))$$
4. Axiome de description définie :
   $$(\Delta_s) \qquad \forall P(\exists^s P(x_1) \dots (x_n) \Rightarrow P(\Delta^s_1(P)) \dots (\Delta^s_n(P)))$$
5. Axiome d'antisymétrie :
   $$(AS) \qquad \forall A, B((A =_\subset B) \Rightarrow (A =_L B))$$

Annexe B

Grammaire du langage

B.1 Grammaire

Les mots clés sont écrits en style sans sérif.

Les variables *var* sont des identificateurs qui sont des chaînes de caractères formées de lettres minuscules ou de chiffres et qui commencent par une lettre minuscule.

Les labels *label* sont des identificateurs qui sont des chaînes de caractères formées de lettres minuscules ou de chiffres et qui commencent par une lettre minuscule.

Les constructeurs *constr* sont des identificateurs qui sont des chaînes de caractères formées de lettres majuscules ou de chiffres et qui commencent par une lettre majuscule. Les noms de fichiers $file$ sont des identificateurs qui sont des chaînes de caractères qui commencent par une lettre majuscule, dont la suite est formée de lettres minuscules, du symbole _ ou de chiffres. ϵ représente "rien du tout".

$$
\begin{array}{lcl}
prog & := & var \\
 & | & \texttt{fun}\ var \rightarrow prog \\
 & | & (\ prog\ prog\ ) \\
 & | & \mathsf{let}\ var = prog\ \mathsf{in}\ prog \\
 & | & \mathsf{rec}\ prog \\
 & | & (\ prog\ ,\ prog\ ) \\
 & | & \mathsf{fst}\ prog \\
 & | & \mathsf{snd}\ prog \\
 & | & <record> \\
 & | & prog.label \\
 & | & [record_{atom}] + prog \\
 & | & prog\ \mathsf{as}\ labellist \\
 & | & constr \\
 & | & constr[\ prog\ ] \\
 & | & \mathsf{cases}\ prog\ \mathsf{of}\ motifs \\
 & | & \mathsf{Struct}\ structure \\
 & | & file \\
 & & \\
record & := & \epsilon \\
 & | & ;\ record_{atom}\ ;\ record \\
 & & \\
record_{atom} & := & label = prog \\
 & & \\
labellist & := & \epsilon \\
 & | & labellist\ ,\ label \\
 & & \\
motifs & := & motif \\
 & | & motifs \quad motif \\
 & & \\
motif & := & constr \rightarrow prog \\
 & | & \_ \rightarrow prog \\
 & | & constr\ \ [var] \rightarrow prog \\
 & & \\
structure & := & open_{decl}\ ;\ let_{decl} \\
 & & \\
open_{decl} & := & \epsilon \\
 & | & open_{decl}\ ;\ \mathsf{open}\ prog\ \mathsf{as}\ labellist \\
 & & \\
let_{decl} & := & \mathsf{endStruct} \\
 & | & let_{atom}\ !\ \ let_{decl} \\
 & & \\
let_{atom} & := & \mathsf{let}\ var\ \mathsf{as}\ label = prog \\
 & | & \mathsf{let\ rec}\ var\ \mathsf{as}\ label = prog
\end{array}
$$

B.2 Règles de réduction

B.2.1 Valeurs

L'ensemble des valeurs est défini comme un sous-ensemble de l'ensemble des programmes :

$$
\begin{array}{lll}
val & := & \\
 & & \\
 & & \mathsf{fun}\ x \rightarrow p \\
 & | & constr \\
 & | & constr\ \ [val] \\
 & | & (\ prog\ ,\ prog\ ) \\
 & | & record \\
 & | & [label = prog] + prog
\end{array}
$$

Les valeurs seront notées v.

B.3 Règles d'évaluation

B.3.1 Relation "filtre"

On dit qu'un motif m filtre un programme u, et on appelle le résultat du filtrage $m \leftarrow u$ dans les cas suivants :
- $m = C \rightarrow v$ et $u = C : m \leftarrow u = v$
- $m = C[x] \rightarrow v(x)$ et $u = C[a] : m \leftarrow u = v(a)$
- $m = \_ \rightarrow v$ et $u = C$ ou $u = C[a] : m \leftarrow u = v$

B.3.2 Evaluation gauche

1. Réduction de tête λ :

$$\frac{}{(\mathsf{fun}\ x \rightarrow t\ \ u)u_1 \ldots u_n \succ t[x := u]u_1 \ldots u_n}$$

2. Récurseur :

$$\frac{}{(\mathsf{rec}\ \ \ u)u_1 \ldots u_n \succ (u(\mathsf{rec}\ \ \ u))u_1 \ldots u_n}$$

3. Première projection :

$$\frac{}{\mathsf{fst}\ (a,b)u_1 \ldots u_n \succ au_1 \ldots u_n}$$

4. Seconde projection :

$$\frac{}{\mathsf{snd}\ (a,b)u_1 \ldots u_n \succ bu_1 \ldots u_n}$$

5. Réduction de tête pour un programme qui n'est pas une paire (proj=fst ou snd) :

$$\frac{u \succ u'}{(\mathsf{proj}\ u)u_1 \dots u_n \succ (\mathsf{proj}\ u')u_1 \dots u_n}$$

6. Réduction de tête pour les cases :

$$\frac{m_i \text{ filtre } u}{(\mathsf{cases}\ u\ \mathsf{of}\ m_1|\dots|m_n)u_1 \dots u_n \succ (m_i \leftarrow u)u_1 \dots u_n}$$

7. Réduction de tête cases (argument qui ne commence pas par un constructeur) :

$$\frac{u \succ u'}{(\mathsf{cases}\ u\ \mathsf{of}\ pattern)u_1 \dots u_n \succ (\mathsf{cases}\ u'\ \mathsf{of}\ pattern)u_1 \dots u_n}$$

8. Projection d'enregistrement :

$$\frac{l_1, \dots, l_n \neq l}{< l_1 = u_1; \dots; l_n = un; l = u; R > .lu_1 \dots u_n \succ uu_1 \dots u_n}$$

9. Projection de champs ajouté (=) :

$$\frac{}{([l{=}u]{+}r).lu_1 \dots u_n \succ uu_1 \dots u_n}$$

10. Projection de champs ajouté (≠) :

$$\frac{}{([l'{=}u]{+}r).lu_1 \dots u_n \succ r.lu_1 \dots u_n}$$

11. Projection sur un programme qui n'est ni un enregistrement ni un ajout de champs :

$$\frac{u \succ u'}{u.lu_1 \dots u_n \succ u'.lu_1 \dots u_n}$$

12. Réduction sous un constructeur :

$$\frac{u \succ u'}{C[u] \succ C[u']}$$

B.3.3 Machine à pile

On définit une machine à piles pour gérer l'évaluation définie précédemment :

$Push_{app}$	$(u\ v)$	$*$	π	$\succ$	u	$*$	v, π
Pop_{fun}	$\mathsf{fun}\ x \to \mathsf{a}$	$*$	b, π	$\succ$	$a[x := b]$	$*$	π
Pop_{rec}	$\mathsf{rec}\ f$	$*$	π	$\succ$	$(f\ \mathsf{rec}\ f)$	$*$	π
$Push_{fst}$	$\mathsf{fst}\ u$	$*$	π	$\succ$	u	$*$	Fst, π
$Push_{snd}$	$\mathsf{snd}\ u$	$*$	π	$\succ$	u	$*$	Snd, π
Pop_{fst}	(a,b)	$*$	Fst, π	$\succ$	a	$*$	π
Pop_{snd}	(a,b)	$*$	Snd, π	$\succ$	b	$*$	π
$Push_{case}$	$\mathsf{case}\ u\ \mathsf{of}\ M$	$*$	π	$\succ$	M'	$*$	$\mathsf{fun}\ x \to x, u, \pi$
$Push_{Eq}$	$\mathsf{Eq}\ u$	$*$	C, π	$\succ$	u	$*$	Eq, C, π
Pop_{Eq}	$C[a]$	$*$	Eq, C, v, π	$\succ$	v	$*$	$C[v], \pi$
Pop_{NEq}	$D[a]$	$*$	$\mathsf{Eq}, C, v, w, \pi$	$\succ$	w	$*$	π
Pop_{fun2}	$\mathsf{fun2}\ x \to t$	$*$	$C[u], \pi$	$\succ$	$t[x := u]$	$*$	π
$Push_{proj}$	$u.l$	$*$	π	$\succ$	u	$*$	$., l, \pi$
Pop_{proj1}	$<\overline{l_1 = u_1}; l = u; \ldots >$	$*$	$., l, \pi$	$\succ$	u	$*$	π
Pop_{proj2}	$[l{=}u]$ + p	$*$	$., l, \pi$	$\succ$	u	$*$	π
Pop_{proj3}	$[l'{=}u]$ + p	$*$	$., l, \pi$	$\succ$	p	$*$	$., l, \pi$
$Push_{constr}$	$C[u]$	$*$	π	$\succ$	u	$*$	$C[].\pi$
Pop_{constr}	v	$*$	$C[].\pi$	$\succ$	$C[v]$	$*$	π

M' étant défini par :

$$
\begin{array}{lcl}
(\_ \to t)' & = & \mathsf{fun}\ f \to \mathsf{fun}\ c \to t \\
(C, x \to t)' & = & \mathsf{fun}\ f \to \mathsf{fun}\ c \to (((\mathsf{Eq}\ c\ \mathsf{C})\ (\mathsf{fun2}\ x \to t\ c)\ f) \\
(C \to t)' & = & \mathsf{fun}\ f \to \mathsf{fun}\ c \to (((\mathsf{Eq}\ c\ \mathsf{C})\ \mathsf{t'})\ f) \\
(m|M)' & = & \mathsf{fun}\ f \to \mathsf{fun}\ c \to (m\ (M\ f\ c)\ c)
\end{array}
$$

B.4 Evaluation en profondeur

L'évaluation en profondeur sert à évaluer sous les paires, et les enregistrements. On la définit par induction :

–

$$\frac{a \succ v}{a \succ_d v}$$

Si v n'est ni une paire ni un enregistrement.

–

$$\frac{a \succ_d a' \qquad b \succ_d b'}{(a,b) \succ_d (a',b')}$$

–

$$\frac{a_1 \succ_d a'_1 \ldots a_n \succ_d a'_n}{<l_1{=}a_1\,;\ldots;l_n{=}a_n) \succ_d <l_1{=}a'_1\,;\ldots;l_n{=}a'_n)>}$$

–

$$\frac{a \succ_d a' \qquad p \succ_d p'}{[l{=}a]{+}p \succ_d [l{=}a']{+}p'}$$

Annexe C

Extraits de fichiers Phox

Nous présentons ici quelques extraits des sorties générées par PhoX à partir de nos fichiers, afin que le lecteur voie la forme que prennent les énoncés prouvés formellement. Les fichiers concernant le système ST et les entiers comptent 4045 lignes et ceux concernant les types de données, 4854, soit au total environ 8900 lignes de code.

C.1 Axiomes, opérateurs et entiers

C.1.1 Axiomes et propriétés des opérateurs

Le fichier `st_ax.phx` contient les définitions des axiomes et opérateurs :

```
(* definition des sortes*)

Sort type defined
Sort term defined

(* definition des constantes *)
$@ : term -> term -> term
$fun : (term -> term) -> term
$|-> : type -> type -> type
$#/\ : ('a -> type) -> type
$subset : type -> type -> prop
$->> : prop -> type -> type
Inh : term -> type -> prop

(* quelques axiomes *)

arrow.intro =
   /\X,Y /\t (/\x (Inh x X -> Inh (t x) Y) -> Inh (fun x ==> t x) (X |-> Y))

arrow.elim = /\X,Y /\t,u (Inh t X -> Inh u (X |-> Y) -> Inh (u @ t) Y)

elim.Sub = /\A,B /\t (Inh t A -> (A subset B) -> Inh t B)

intro.FArrow = /\P /\A /\t ((P -> Inh t A) -> Inh t (P ->> A))

elim.FArrow = /\P /\A /\t (P -> Inh t (P ->> A) -> Inh t A)

trans.Sub = /\A,B,C ((A subset B) -> (B subset C) -> (A subset C))
```

```
left_Sub = /\B /\x (#/\x B x subset B x)

arrow.Sub =
   /\A,B,A',B' ((B subset A) -> (A' subset B') -> (A |-> A' subset B |-> B'))

left_Free = /\A /\P (P -> (P ->> A subset A))

Inversion.Dash = /\P /\A,B (P ->> A |-> B subset A |-> P ->> B)


(* definition de quelques operateurs *)

Empty = #/\X X : type

Rest = \F,G #/\K ((F subset G ->> K) ->> K) : type -> prop -> type

U2 = \A,B #/\K ((A subset K) ->> (B subset K) ->> K) : type -> type -> type

U = \F #/\K (/\x (F x subset K) ->> K) : ('a -> type) -> type

Image = \A,B #/\X ((A subset B |-> X) ->> X) : type -> type -> type

is_Empty = \A (A subset Empty) : type -> prop

Singl1 = \A (Non_empty A & /\B ((A subset U B) -> \/X (A subset B X)))
 : type -> prop

Inter = \A,B #/\K (/\P ((P subset A) -> (P subset B) -> (P subset K)) ->> K)
 : type -> type -> type

Cmpl = \A (U \X (Rest X (Inter X A subset Empty))) : type -> type


(* quelques axiomes avec opérateurs*)

absurd_less = /\A /\t (Inh t ((A ->> Empty) |-> Empty) -> A)

set_ax = /\A (A subset U \X (Rest X (Singl1 X & (X subset A))))

cmpl_ax = /\A,B (A subset U2 B (Cmpl B))

singl_cons = /\A,B (Singl1 A -> Singl1 B -> Singl1 (Image A B))
```

Le fichier `st_basicfacts.phx` contient les propriétés des opérateurs :

```
(* proprietes de l'application *)

prop_applic.1 = /\A,B,C ((Image A B subset C) <-> (A subset B |-> C))

(...)

(* proprietes du faux *)

prop_false.1 = /\P (((P -> Subfalse) -> Subfalse) -> P)

prop_false.2 = Subfalse -> False

(...)

prop_false.7 =
   /\x /\A,B,B' (Inh x A -> (B subset B') -> (A |-> B subset A |-> B'))
```

```
(* proprietes de la restriction *)

prop_restr.1 = /\A,B /\P ((P -> (A subset B)) -> (Rest A P subset B))

(...)

prop_restr.6 = /\A /\P (Non_empty (Rest A P) -> P)

(* propriete de l'union *)

prop_union = /\F /\A (#/\x:F  A subset U F |-> A)


(* singletons *)

(...)

prop_singl.3 =
/\A,B(#/\X( (Singl1 X ) ->> (X subset A) ->> (X |-> B)) subset (A |-> B))


(* intersection, complementaire, reunion  *)

prop_inter = /\A /\B eq_sub (Inter A (U B)) (U \x (Inter A (B x)))

prop_compl = /\A ((A subset Cmpl A) -> is_Empty A)

union2 = \A,B (U \X (Rest X (eq_sub X A or eq_sub X B))) : type -> type -> type

prop_union.2 = /\A,B,C eq_sub (Inter (U2 B C) A) (U2 (Inter B A) (Inter C A))


(* decideur et booleens *)

Decid = \P,Q #/\X ((P ->> X) |-> (Q ->> X) |-> X) : prop -> prop -> type
Bool = #/\X (X |-> X |-> X) : type
Btrue = #/\X (X |-> #/\Y (Y |-> X)) : type
Bfalse = #/\X (X |-> #/\Y (Y |-> Y)) : type

prop_decid.1 = /\P,Q (Decid P Q subset Bool)

(...)

prop_decid.5 = /\P ((P -> Subfalse) -> eq_sub Bfalse (Decid P (P -> Subfalse)))

(* Types inductifs *)

Mu = \F #/\X (/\A ((A subset X) -> (F A subset X)) ->> X)
 : (type -> type) -> type

Cresc = \F /\X,Y ((X subset Y) -> (F X subset F Y)) : (type -> type) -> prop
          /\Y (/\A ((A subset Y) -> (F A subset Y)) -> (X subset Y)))
(...)

prop_mu.3 = /\F /\X (/\A ((A subset X) -> (F A subset X)) -> (Mu F subset X))

(...)

prop_fix.2 = /\F:Cresc  eq_sub (F (Mu F)) (Mu F)
```

C.1.2 β-reduction et singletons

Le fichier `st_beta.phx` contient les faits relatifs à la preuve de la réduction du chapitre 4 (ayant défini les regles comme des axiomes, et non comme un prédicat inductif, on ne traite pas les inductions) :

```
(* definition de la lambda -abstraction de types *)

Lambda = \f #/\X (X |-> f X) : (type -> type) -> type

(* lambda-expansion et eta-reduction *)

beta_exp = /\A /\F (Image (Lambda F) A subset F A)

eta_red = /\A (A subset Lambda (Image A))

(* sur les singletons *)

Singl_union =
   \A (Non_empty A & /\X,Y ((A subset U2 X Y) -> (A subset X) or (A subset Y)))
 : type -> prop

Singl_subset =
   \A (Non_empty A & /\X ((X subset A) -> Non_empty X -> eq_sub A X))
 : type -> prop

non_vide_contient_singl = /\A:Non_empty  \/x:Singl1  (x subset A) : theorem

equiv_singletons =
   /\A
     ((Singl1 A -> Singl_union A) & (Singl_union A -> Singl_subset A) &
        (Singl_subset A -> Singl1 A))

autre_carac_singl =
   /\A:Non_empty  (/\x:Singl1  ((x subset A) -> (A subset x)) -> Singl1 A)

eta_red_singl = /\A:Singl1  (Lambda (Image A) subset A)

abstraction_preserve_singleton =
   /\A (Non_empty (Lambda A) -> /\X:Singl1  Singl1 (A X) -> Singl1 (Lambda A))
```

C.1.3 Expressivité

Le fichier `st_expr.phx` contient les preuves des enonces 56, 57, et 61 du chapitre 5 :

```
(* axiome de non-trivialite *)

Non_triv = /\X,Y (Singl1 X -> Singl1 Y -> (X subset Y)) -> Subfalse


(* les trois enonces *)

booleens_distincts = (Btrue subset Bfalse) -> Subfalse

exist_enonces_indecidables =
   /\P Non_empty (#/\X (Singl1 X ->> X |-> Decid (P X) (P X -> Subfalse))) ->
     Subfalse


vacuite_classique = #/\X (((X |-> Empty) |-> Empty) |-> X) subset Empty
```

Le fichier `st_normresol.phx` contient les preuves des énoncés attachés au théorème 62 et au théorème 71 du chapitre 5, avec quelques exemples. Noter qu'on y définit ici en fait déja le schéma de compréhension, et qu'on montre sa propriété en l'absence de l'axiome d'égalité pour les prédicats compatibles avec l'égalité $=_{\subset}$:

```
(* adequation et prédicat etre normalisable *)

adequ =
   \A,B ((A subset B |-> A) & (B |-> A subset A |-> B) & (A |-> B subset B))
 : type -> type -> prop

subnorm = \X /\A,B (adequ A B -> (X subset B)) : type -> prop


(* schema de comprehension *)

SC = \P (U \X (Rest X (Singl1 X & P X))) : (type -> prop) -> type

sub_compatible = \P /\X,Y (eq_sub X Y -> P X <-> P Y) : (type -> prop) -> prop

prop_SC_subcompatible =
   /\P:sub_compatible
     /\A ((A subset SC P) <-> /\X ((X subset A) -> Singl1 X -> P X))


(* Type des normalisables *)

Norm = SC subnorm


defnorm_ok = /\X ((X subset Norm) <-> subnorm X)

Nat = #/\X (X |-> (X |-> X) |-> X) : type

(* quelques exemples *)

nat_norm = Nat subset Norm
true_norm = Btrue subset Norm


(* resolubles *)

fonction_F = \A (U \X (Rest (X |-> A) (Non_empty X))) : type -> type

fonction_F' = \A (U2 (Lambda \X X) (fonction_F A)) : type -> type

Resol = Mu fonction_F' : type

resol_pt_fixe = eq_sub Resol (fonction_F' Resol)

(* exemples *)

Btrue_resol = Btrue subset Resol
Deux = Lambda \x (Lambda \f (Image f (Image f x))) : type

deux_resol = Deux subset Resol
```

C.2 λ-termes et types

On consacre deux fichiers à la syntaxe et aux preuves concernant les λ-termes dans les types : il s'agit des fichiers `st_singleton_terme.phx` et `st_quelques_termes.phx`

```
(* syntaxe pour l'application et l'abstraction *)

$@@ = Image : type -> type -> type
$FUN = Lambda : (type -> type) -> type

(* predicat "habite et est un singleton", qui correspond à "appartient" dans la traduction du langage *)

$Inh_singl = \x,X (Inh x X & Singl1 X) : term -> type -> prop

(* regles d'intros *)

inh_fun.intro =
   /\f /\F
       (/\x /\X:(Inh x)  Inh (f x) (F X) ->
          Inh (fun x ==> f x) (FUN X ===> F X))

inh_app.intro = /\x /\X /\y /\Y (Inh x X -> Inh y Y -> Inh (x @ y) (X @@ Y))

inh_inh_singl.intro = /\x /\X:(Inh_singl x)  Inh x X

inh_singl_applic.intro =
   /\x/\X /\y /\Y (Inh_singl x X -> Inh_singl y Y -> Inh_singl (x @ y) (X @@ Y))

inh_singl_fun.intro =
   /\f /\F
       (Inh (fun x ==> f x) (FUN X ===> F X) ->
          /\x /\X:(Inh_singl x)  Inh_singl (f x) (F X) ->
          Inh_singl (fun x ==> f x) (FUN X ===> F X))

(* un exemple *)

beta_typ = /\F /\A ((FUN X ===> F X) @@ A subset F A)

(* quelques termes standards et leurs traduits en types, préfixés par une majuscule *)

paire = \x,y fun c ==> (c @ x) @ y : term -> term -> term
Paire = \x,y FUN c ===> (c @@ x) @@ y : type -> type -> type
proj1 = fun c ==> c @ btrue : term
Proj1 = FUN c ===> c @@ Btrue : type
proj2 = fun c ==> c @ bfalse : term
Proj2 = FUN c ===> c @@ Bfalse : type

inh_paire =
   /\x /\X  /\y /\Y (Inh_singl x X -> Inh_singl y Y -> Inh_singl (paire x y)   (Paire X Y))

inh_proj1 = Inh_singl proj1 Proj1

inh_proj2 = Inh_singl proj2 Proj2

zero = fun x,f ==> x : term
Zero = FUN x,f ===> x : type

succ = fun n,x,f ==> f @ (n @ x) @ f : term
Succ = FUN n,x,f ===> f @@ (n @@ x) @@ f : type

inh_zero = Inh_singl zero Zero

inh_succ = Inh_singl succ Succ
```

C.3 Types de Données

C.3.1 Entiers

Le fichier `st_purnat.phx` contient la preuve du théorème 75 :

```
(* definitions des types et prédicats associés aux entiers *)

stnat     = \n #/\X (X Zero |-> #/\y:X  X (Succ @@ y) |-> X n) : type -> type

stnat_log = \n  /\X (X Zero  ->  /\y:X  X (Succ @@ y)  -> X n) : type -> prop

stnatplus = \n #/\X (X Zero |-> #/\y (stnat_log y ->> X y |-> X (Succ @@ y)) |-> X n) : type -> type

(* type de donnees strict *)

tdd_strict = \P /\x (Non_empty (P x) -> Singl1 x & eq_sub x (P x))
 : (type -> type) -> prop

(... lemmes ... )

stnat_strict_datatype = tdd_strict stnat
```

Les fichiers `zero_succ.phx` et `plus_mult.phx` contiennent les preuves des axiomes de Peano :

```
(* termes et habitants *)

Termpred = \n ((n @@ Paire Zero Zero) @@ FUN c ===> Paire (Proj2 @@ c) (Succ @@ Proj2 @@ c)) : type -> type

Pred = FUN n ===> Proj1 @@ Termpred n : type

(* enonces *)

factpred1 = /\n:stnat_log  (Pred @@ Succ @@ n subset n)

succ_injectif =   /\n,m:stnat_log  ((Image Succ n subset Image Succ m) -> (n subset m))

two_distincts = \/x,y (Singl1 x & Singl1 y & ((x subset y) -> Subfalse)) : prop

td = two_distincts

zero_neq_succ = /\n ((Zero subset Image Succ n) -> Subfalse)

non_nul_succ =   /\n:stnat_log   (((n subset Zero) -> False) -> \/m:stnat_log  (n subset Image Succ m))


(* Addition *)

Plus = \n,m ((m @@ n) @@ Succ) : type -> type -> type

plus_cas_zero = /\n (Plus n Zero subset n)
plus_cas_succ = /\n,m (Plus n (Succ @@ m) subset Succ @@ Plus n m)

Add = \n FUN x ===> Plus x n : type -> type

(* Multiplication *)

Mult = \n,m ((m @@ Zero) @@ Add n) : type -> type -> type

mult_cas_zero = /\n (Mult n Zero subset Zero)
mult_cas_succ = /\n,m (Mult n (Succ @@ m) subset Add n @@ Mult n m)
```

Enfin, le fichier `equal_nat.phx` contient la preuve du fait 81 :

```
(* lambda-termes *)

Eq_zero = FUN m ===> (m @@ Btrue) @@ FUN r ===> Bfalse : type

Eq_succ =
   FUN r,m ===> Proj1 @@ (m @@ Paire Bfalse Zero) @@
                FUN rec ===> Paire (r @@ Proj2 @@ rec) (Succ @@ Proj2 @@ rec) :  type

Eq_nat = FUN n ===> (n @@ Eq_zero) @@ Eq_succ : type

(* singleton *)

inh_eq_nat = Inh_singl eq_nat Eq_nat : theorem

(* sous-typage *)

sub_equal =  Eq_nat subset #/\x,y:stnat  Decid (eq_sub x y) (eq_sub x y -> Subfalse)

equal_decid = decidable eq_sub stnat : theorem

equal_true_false =
   /\x,y:stnat_log
     ((eq_sub x y -> ((Eq_nat @@ x) @@ y subset Btrue)) &
        ((eq_sub x y -> Subfalse) -> ((Eq_nat @@ x) @@ y subset Bfalse)))
```

C.3.2 Produits dépendants et schéma de compréhension

Le fichier `type_prog.phx` contient les définitions de Π^τ, Π^o et du schéma de compréhension $\{x : A; P(x)\}$, et leurs propriétés :

```
(* Produit dépendants et produit logique *)

$Pi = \A,B #/\x (Singl1 x ->> (x subset A) ->> x |-> B x) : type -> (type -> type) -> type

$FA_o = \A,B /\x:Singl1  ((x subset A) -> B x) : type -> (type -> prop) -> prop

(* predicat d'habitation pour les types *)

INH = \T,A (Singl1 T & (T subset A)) : type -> type -> prop

(* regles d'introduction et d'elimination : la regle d'intro comporte une condition de singletonicite
utilisant le predicat Inh_singl defini dans le fichier st_singleton_terme *)

PI_intro =   /\F /\A /\B
         (/\X (INH X A -> INH (F X) (B X)) -> \/f Inh_singl f (FUN X ===> F X)
          -> INH (FUN X ===> F X) (Pi x :: A :: B x))

PI_elim =   /\X,F,A /\B (INH F (Pi x :: A :: B x) -> INH X A -> INH (F @@ X) (B X))

(...)

FA_o_intro = /\A /\P (/\X (INH X A -> P X) -> FA_o x :: A :: P x)

FA_o_elim = /\X,A /\P (INH X A -> FA_o x :: A :: P x -> P X)

(* axiome de l'egalite et schema de comprehension  *)

equal_equal = /\X,Y (eq_sub X Y -> X = Y)

prop_SC = /\P /\A ((A subset SC P) <-> /\X ((X subset A) -> Singl1 X -> P X))
```

```
SC.intro = /\A /\P (FA_o x :: A :: P x -> (A subset SC P)) : theorem

SC.elim = /\A /\P ((A subset SC P) -> FA_o x :: A :: P x) : theorem

(* schema de comprehension du langage *)

$sc = \A,B SC \x (INH x A & (B x)): type -> (type -> prop) -> type

sc.intro =
/\x,A,P (INH x  A ->  (P x) -> INH x  sc r :: A :: (P r )): theorem

sc.elim =
/\A,P,t
/\K( ( INH t  A -> (P t) -> K) ->  INH t  sc x :: A :: (P x) -> K) : theorem


(* lien entre les produits *)

Pi_FA =   /\A /\B /\G
        ((G subset Pi x :: A :: B x) <-> FA_o x :: A :: (G subset x |-> B x))

Pi_arrow =   /\A,B ((Pi x :: A :: B subset A |-> B) & (A |-> B subset Pi x :: A :: B))

(* sous-typage des produits dependants *)

sub.PI_PI =   /\A  /\B  /\A' /\B'
          (/\x (INH x A' -> (B x subset B' x)) -> (A' subset A)
          -> (Pi x :: A :: B x subset Pi x :: A' :: B' x))
```

C.3.3 Produits et paires

Le fichier `st_tdd_prod.phx` contient les preuves des propriétés du produit (fait 82) :

```
(* definition du type A times B *)

$times = \A,B (U \x (U \y (Rest (Paire x y) (INH x A & INH y B)))) : type -> type -> type

(* regles d'introduction et d'elimination *)

prod.intro = /\X,Y,A,B (INH X A -> INH Y B -> INH (Paire X Y) (A times B))

prod.elim.left = /\A,B,X (INH X (A times B) -> INH (Proj1 @@ X) A) : theorem

prod.elim.right = /\A,B,X (INH X (A times B) -> INH (Proj2 @@ X) B) : theorem

prop_prod.surpair =   /\A,B FA_o x :: A times B :: x = Paire (Proj1 @@ x) (Proj2 @@ x)
```

C.3.4 Types sommes

Le fichier `st_tdd_sum_typsum.phx` contient la définition d'absence d'un constructeur dans un type ainsi que la définition inductive d'un type somme, ainsi que les règles d'introduction afférentes (qui sont évidentes : voir dans les fichiers) :

```
(* Absence d'un constructeur dans un type *)

Diff1 = \C,T FA_o X :: T :: (eq_sub (Proj1 @@ X) C -> Subfalse) : type -> type -> prop

Disjoint = \U0,V (Inter U0 V subset Empty) : type -> type -> prop
```

```
type_somme =   \A /\X
       (/\C,A   (stnat_log C ->                    X (C OF A)) ->
        /\C,A,T (X T -> Diff1 C T -> stnat_log C -> X (C OF A || T)) ->
                                                   X A) : type -> prop

type_somme.intro.of = /\C,A (stnat_log C -> type_somme (C OF A)) : theorem

type_somme.intro.of_union_T = /\C,T,A (type_somme T -> Diff1 C T -> stnat_log C -> type_somme (C OF A || T))

type_somme.intro.nil = /\C:stnat_log  type_somme (C NIL) : theorem

type_somme.intro.nil_union_T = /\C,T (type_somme T -> Diff1 C T -> stnat_log C -> type_somme (C NIL || T))

diff = \A,B (eq_sub A B -> Subfalse) : type -> type -> prop

Diff1.intro =   /\C1,C2 /\A /\B (stnat_log C1 -> stnat_log C2 -> diff C1 C2 -> Diff1 C1 (C2 OF B))

Diff1.intro.union = /\T,S,C (Diff1 C T -> Diff1 C S -> Diff1 C (T || S))

inh_typ_sum_nat = /\X,T (type_somme T -> INH X T -> stnat_log (Proj1 @@ X))

diff1_eq_sub =  /\X,T,C (Diff1 C T -> INH X T -> eq_sub (Proj1 @@ X) C -> Subfalse)
```

Le fichier `st_tdd_sum_intro.phx` contient les preuves des règles d'introduction de la somme (utilisées dans la preuve de correction) :

```
(* definition des termes et des  types: voir le fichier, il s'agit de
paires et de reunions *)

c_of.intro = /\C,A,X (stnat_log C -> INH X A -> INH (C [[ X ]]) (C OF A))

c_nil.intro = /\C:stnat_log  INH (C [[]]) (C NIL)

c_of_log.intro = /\C,A /\P (FA_o x :: A :: P (C [[ x ]]) -> stnat_log C -> FA_o y :: C OF A :: P y)

c_nil_log.intro = /\C /\A /\P (stnat_log C -> P (C [[]]) -> FA_o y :: C NIL :: P y) :

sum_or = /\A,B,X (INH X (A || B) -> INH X A or INH X B) : theorem
```

Le fichier `st_tdd_sum_elim.phx` contient les preuves des règles d'introduction de la somme (propositions 84 et 85) :

```
(* definition des termes, motifs et cases *)
$, =   \C,t,f,c
     ((((fun c ==> (eq_nat @ proj1 @ c) @ C) @ c) @
         (fun x ==> (fun y ==> t y) @ proj2 @ x) @ c) @ f)
 : term -> (term -> term) -> term -> term -> term

$---> = \C,t (C ,x --> t) : term -> term -> term -> term -> term

$other = \t,f,c t : 'a -> 'c -> 'b -> 'a

$| = \M1,M2,f,c (M1 (M2 f c) c) : ('d -> 'b -> 'a) -> ('c -> 'b -> 'd) -> 'c -> 'b -> 'a

$cases = \x,M (M (fun x ==> x) x) : 'b -> (term -> 'b -> 'a) -> 'a

$,, =
   \C,t,f,c
     ((((FUN c ===> (Eq_nat @@ Proj1 @@ c) @@ C) @@ c) @@
         (FUN x ===> (FUN Y ===> t Y) @@ Proj2 @@ x) @@ c) @@ f)
 : type -> (type -> type) -> type -> type -> type
```

```
$-----> = \C,t (C ,,X ----> t) : type -> type -> type -> type -> type

$Other = \t,f,c t : 'a -> 'c -> 'b -> 'a

$||| = \M1,M2,f,c (M1 (M2 f c) c)
 : ('d -> 'b -> 'a) -> ('c -> 'b -> 'd) -> 'c -> 'b -> 'a

$Cases = \x,M (M (FUN x ===> x) x) : 'b -> (type -> 'b -> 'a) -> 'a

(* predicat inductif "bon_motif" pour prouver la singletonicite singletonicite *)

bon_motif =  \M /\X
       (/\T (\/t Inh_singl t T -> X (Other --> T)) ->
       /\C /\T (stnat_log C -> \/t Inh_singl (fun x ==> t x) (FUN X0 ===> T X0)
                              -> X (C ,,x ----> T x)) ->
          /\M /\C /\T (X M -> stnat_log C -> \/t Inh_singl (fun x ==> t x) (FUN X0 ===> T X0) ->
                              X (C ,,x ----> T x ||| M))
       -> X M)
 : (type -> type -> type) -> prop

bon_motif_singleton =   /\M /\X (bon_motif (M X) -> Singl1 X -> Singl1 (Cases X Of M X))

(* preuves de la proposition *)

sub_motif_other =   /\C /\B /\A /\T /\X
               (type_somme A -> FA_o y :: A :: INH (T y) (B y) ->
                \/t Inh_singl t (T X) -> INH X A ->
                INH (Cases X Of Other --> T X) (B X))

sub_motif_simple_of =   /\C /\B /\A  /\T  /\X
               (stnat_log C -> FA_o y :: A :: INH (T y) (B (C [[ y ]])) ->
                \/t Inh_singl (fun x ==> t x) (FUN X ===> T X) ->
                INH X (C OF A) -> INH (Cases X Of C ,,x ----> T x) (B X))


sub_motif_simple_nil =   /\C /\B /\A /\T,X
             (stnat_log C -> INH T (B (C [[]])) -> \/t Inh_singl t T ->
              INH X (C NIL) -> INH (Cases X Of C -----> T) (B X))


sub_motif_compose_of =   /\C /\B  /\A /\T  /\U0  /\M  /\X
                  (stnat_log C -> Diff1 C U0 -> type_somme U0 ->
                   FA_o y :: A :: INH (T y) (B (C [[ y ]])) ->
                   \/t Inh_singl (fun x ==> t x) (FUN X ===> T X) ->
                   FA_o y :: U0 :: INH (Cases y Of M y) (B y) ->
                   bon_motif (M X) -> INH X (C OF A || U0) ->
                   INH (Cases X Of C ,,x ----> T x ||| M X) (B X))


sub_motif_compose_nil = /\C /\B  /\A  /\T,U0  /\M
              (stnat_log C -> Diff1 C U0 -> type_somme U0 ->
               INH T (B (C [[]])) -> \/t Inh_singl t T ->
               FA_o y :: U0 :: INH (Cases y Of M y) (B y) ->
               /\X (bon_motif (M X) -> INH X (C NIL || U0) ->
               INH (Cases X Of C -----> T ||| M X) (B X)))
```

C.3.5 Enregistrements

Le fichier `st_tdd_record.phx` contient les preuves des règles d'introduction de la somme (propositions 84 et 85) :

```
(* types *)
```

```
$;; = \T1,T2,X (T1 X || T2 X) : ('a -> type) -> ('a -> type) -> 'a -> type

$r_: = \C,A,X (C OF Pi y :: A :: X) : type -> type -> type -> type

${ = \T #/\X Pi y :: T X :: X : (type -> type) -> type

(* enregistrements *)

$R_= = \C,u (C ,,y ----> y @@ u) : type -> type -> type -> type -> type

$<< = \T FUN x ===> Cases x Of T : (type -> type -> type) -> type

$**=** = \C,a,r FUN x ===> Cases x Of C ,,y ----> y @@ a | Other --> r @@ x
 : type -> type -> type -> type

(* regle de formation et singletons *)

bon_record = bon_motif

record.singl = /\T:bon_record  \/t Inh_singl t << T >>

(* regles d'introduction *)

type_record = /\A,C,a (stnat_log C -> INH a A -> INH << C R_= a >> { C r_: A })
 : theorem

type_record_multiple =
   /\A,C,a
     /\U0
       /\T
         (stnat_log C -> INH a A -> INH << U0 >> { T } ->
          bon_motif U0 -> /\A Diff1 C (T A) -> /\A type_somme (T A) ->
            INH << C R_= a ;;; U0 >> { C r_: A ;; T })


type_record_addition =
   /\A,C,a,U0
     /\T
       (stnat_log C -> INH a A -> INH U0 { T } -> /\A Diff1 C (T A) ->
          /\A type_somme (T A) -> INH (C **=** a + U0) { C r_: A ;; T })


(* projections et elimination *)


$R.proj = \x,c (x @@ (c [[ FUN x ===> x ]])) : type -> type -> type


record.elim = /\X,C,A (stnat_log C -> INH X { C r_: A } -> INH (X R.proj C) A)


(* sous-typage *)

record.sous_type.l = /\A /\T /\C ({ C r_: A ;; T } subset { C r_: A })

record.sous_type.r = /\A /\T /\C ({ C r_: A ;; T } subset { T }) : theorem


(* quelques tests *)

test_recor.2.2 =
   /\x,X,y,Y
     (INH x X -> INH y Y ->
        INH << C1 R_= x ;;; C2 R_= y >> { C1 r_: X ;; C2 r_: Y }) :
```

```
petit_test_record.proj =
   /\x,X,y,Y
     (INH x X -> INH y Y -> INH (<< C1 R_= x ;;; C2 R_= y >> R.proj C1) X)
```

C.3.6 Types abstraits

Le fichier `st_tdd_abstract.phx` contient les preuves des règles des types abstraits (fait 89, restreint à une sorte, et règles de $Self$) :

```
(* Axiome du choix, "Some" et projection "ab" pour des sortes
quelconques *)

AC = /\Q (\/z Q z -> Q (Def Q)) : theorem

$Some = U : ('a -> type) -> type

$ab = \X,T (Def \z (X subset T z)) : type -> ('a -> type) -> 'a

inh.some.intro = /\X /\A /\T (INH X (T A) -> INH X (Some a T a)) : theorem

inh.some.elim = /\X /\T (INH X (Some x T x) -> INH X (T (X ab T))) : theorem

(* Axiome du choix, "Some" et projection "ab" pour la sorte des
types *)

Def_T : (type -> prop) -> type

AC_T = /\Q (\/z Q z -> Q (Def_T Q)) : theorem

$Some_T = U : (type -> type) -> type

$ab_T = \X,T (Def_T \z (X subset T z)) : type -> (type -> type) -> type

inh_T.some.intro = /\X,A /\T (INH X (T A) -> INH X (Some_T a T a)) : theorem

inh_T.some.elim = /\X /\T (INH X (Some_T x T x) -> INH X (T (X ab_T T)))

(* operateur "Self" *)

$Self = \A (SC \x (x subset A x)) : (type -> type) -> type

lemme_SC = /\x /\P (INH x (SC P) <-> Singl1 x & P x) : theorem

self.intro = /\x /\A (INH x (A x) -> INH x (Self x A x)) : theorem

self.elim = /\x /\A (INH x (Self x A x) -> INH x (A x)) : theorem
```

C.3.7 Types inductifs

Le fichier `st_tdd_induc.phx` contient les preuves des règles des types inductifs :

```
(* mu correspond à Mu (vu dans le fichier st_basicfacts.phx *)

$mu = \x (Mu x) : (type -> type) -> type

induc_princip = /\F  /\P
```

```
       (/\A (FA_o x :: A :: P x -> FA_o x :: F A :: P x) ->
          FA_o x :: mu X F X :: P x)

cresc_induc_princip =   /\F /\P
       (Cresc F ->
          /\A (FA_o x :: A :: P x -> (A subset mu X F X) -> FA_o x :: F A :: P x)
        -> FA_o x :: mu X F X :: P x)

(* termes et singletons *)

Delta = \f FUN X ===> f @@ X @@ X : type -> type
Psi = FUN F ===> Delta F @@ Delta F : type

Rec = $@@ Psi : type -> type

(* principe de recursion *)

recursion =   /\F,B   /\t
       (INH t (#/\A (Pi x :: A :: B x |-> Pi x :: F A :: B x)) ->
          INH (Rec t) (Pi x :: mu x F x :: B x))

cresc_recursion =  /\F,B  /\t
       (Cresc F ->
          INH t (#/\A (Pi x :: A :: B x |-> (A subset mu X F X) ->> Pi x :: F A :: B x))
        -> INH (Rec t) (Pi x :: mu x F x :: B x))

(* principe d'induction bien fondee pour n'importe quel type *)

bien_fonde =   \R,A /\P
       (FA_o x :: A :: (FA_o y :: A :: (R y x -> P y) -> P x) ->
          FA_o x :: A :: P x) : (type -> type -> prop) -> type -> prop

induction =   /\A /\R /\B /\t
           (bien_fonde R A ->
              INH t  (#/\x (INH x A ->> Pi y :: A :: (R y x ->> B y) |-> x |-> B x))
            -> INH (Rec t) (Pi x :: A :: B x))
```

C.3.8 Types co-inductifs

Le fichier `st_tdd_coinduc.phx` contient les preuves des règles des types co-inductifs :

```
(* definition de nu *)

$nu = \F (U \x (Rest x (/\A ((x subset A) -> (x subset F A))))) : (type -> type) -> type

(* regles de sous-typage *)

prop_nu.1 =   /\F /\X
  ((nu x F x subset X)   <->  /\Y (/\A ((Y subset A) -> (Y subset F A)) -> (Y subset X)))

prop_nu.2 = /\F /\X ((nu x F x subset X) -> (nu x F x subset F X))

prop_nu.3 =   /\F /\X (/\A ((X subset A) -> (X subset F A)) -> (X subset nu x F x))

prop_nu.4 = /\F (nu x F x subset F (nu x F x))

prop_nu.5 = /\F:Cresc  (nu x F x) = F (nu x F x)

prop_nu.6 = /\F:Cresc  /\I (F I = I -> (I subset nu X F X))

(* regles de coinduction logique et typee, et typage *)

coinduc.o =   /\F  /\P (Cresc F -> FA_o x :: F (nu x F x) :: P x -> FA_o x :: nu x F x :: P x)
```

```
coinduc.tau =  /\F /\T /\B (Cresc F -> (T subset Pi x :: F (nu x F x) :: B x) ->
            (T subset Pi x :: nu x F x :: B x))

lem_rec_coinduc.1 = /\F /\A (Psi subset #/\X ((A |-> X) |-> A |-> F X) |-> A |-> nu x F x)

lem.coinduc.2 =  /\F (Psi subset #/\X (X |-> F X) |-> nu x F x)

st_nu_cresc = /\F /\A (Cresc F -> (A subset F A) -> (A subset nu x F x))
```

C.4 Modules

C.4.1 Definition des structures et signatures et règles d'élimination

Le fichier `structures_signatures.phx` contient les définitions des structures et signatures, et les règles les plus immédiates, qui sont les règles d'élimination :

```
(* definition du let *)

$soit = \a,t (t a) : 'b -> ('b -> 'a) -> 'a

$let = \C,u,a soit x = u dans C R_= x ;;; a x
 : type -> type -> (type -> 'a -> type -> type) -> 'a -> type -> type

(* structures *)

endStruct = C1 R_= C1 : type -> type -> type

$Struct = $<< : (type -> type -> type) -> type

(* definitions des signatures *)

$val = \C,u,s,T,X (INH (X R.proj C) u & s (X R.proj C) T X)
 : type -> type -> (type -> (type -> type) -> type -> prop) ->  (type -> type) -> type -> prop

$FACT = \P,s,T,X (P & s T X) : prop -> ((type -> type) -> type -> prop) -> (type -> type) -> type -> prop

$TYPEDEF = \C,A,t,T,X (T C = A & t A T X)
 : type -> type -> (type -> (type -> type) -> type -> prop) ->  (type -> type) -> type -> prop

$TYPE = \C,t,T,X (t (T C) T X)
 : type -> (type -> (type -> type) -> type -> 'a) -> ( type -> type) -> type -> 'a

endSig = \T,X True : 'b -> 'a -> prop

$Sig = \c Some T SC (c T) : ('a -> type -> prop) -> type

(* regle d'elimination des  signatures *)

Sig.elim =   /\P /\X /\s ((s (X ab \T Self Y s T Y) X -> P) -> INH X (Sig ! s) -> P)

(* regles d'eliminations par composant des signatures *)

val.elim =   /\P  /\X /\T /\C,A /\s ((INH (X R.proj C) A -> s (X R.proj C) T X -> P) ->
                                  (val x as C :=: A ! s x) T X -> P)

FACT.elim = /\K,P /\s /\T /\X ((P -> s T X -> K) -> (FACT P ! s) T X -> K)
```

```
TYPE.elim =  /\K /\A /\X /\T /\C /\s ((s (T C) T X -> K) -> (TYPE t as C ! s t) T X -> K)

TYPEDEF.elim =   /\K  /\A,X  /\T /\C /\s ((s A T X -> K) -> (TYPEDEF t as C :=: A ! s t) T X -> K)

(* quelques tests *)

essai_sig.1 =   /\X,A     /\P
       (INH X (Sig ! TYPEDEF t as C2 :=: A ! val x as C1 :=: t ! endSig) ->  INH (X R.proj C1) A)

essai_sig.2 =   /\X,A     /\P
       (INH X  (Sig ! TYPEDEF t as C2 :=: A ! val x as C1 :=: A ! FACT P x  ! endSig)
         -> INH (X R.proj C1) A & P (X R.proj C1))
```

C.4.2 Definition des structures et signatures avec absences de constructeurs

Le fichier `bon_typ_struct.phx` contient les définitions inductives des structures et signatures avec absences de constructeurs :

```
(* insensibilité de la projection relativement à un champs distinct du premier *)

inequal_label =   /\m  /\C,D,a,A
       (bon_record m -> stnat_log C -> stnat_log D ->
        Diff1 C (D NIL) -> INH a A ->
          (<< C R_= a ;;; m >> R.proj D) = (<< m >> R.proj D))

(* absence d'un constructeur ou d'un nom dans une signature *)

pas_dans_sig =   \C,s /\X
       (/\C:stnat_log  X C endSig -> /\P /\s:(X C)  X C (FACT P ! s) ->
        /\A,D /\s (X C (s A) -> (eq_sub C D -> Subfalse) ->  X C (TYPEDEF t as D :=: A ! s t)) ->
        /\s /\D,C,u (stnat_log C -> stnat_log D -> (eq_sub C D -> Subfalse) ->
                                                  /\x X C (s x) -> X C (val x as D :=: u ! s x)) ->
         X C s) : type -> ((type -> type) -> type -> prop) -> prop

(* invariance des predicats *)

pas_dans_sig_equal =   /\s /\m  /\C /\T
          (bon_record m -> pas_dans_sig C s ->
             /\a,A (INH a A -> s T << m >> -> s T << C R_= a ;;; m >>))

(* invariance des fonctions *)

pas_dans_sig_typ_equal =  /\s  /\m,C  /\T /\A
          (pas_dans_sig C s -> s T m ->
             s \Y (Inter (eq_sub C Y ->> A) ((eq_sub C Y -> Subfalse) ->> T Y)) m)
```

C.4.3 Typage et sous-typage des modules

Le fichier `type_struct.phx` contient les définitions inductives des structures et signatures avec absences de constructeurs :

```
(* predicat inductif de typage des structures *)

Typ_struct =   \m,S   /\X
       (X endStruct endSig ->
          /\a,A /\m  /\s  /\C
                  (INH a A -> X (m a) (s a) -> stnat_log C ->
```

```
                    pas_dans_sig C (s a) ->
                    X (let x as C :=: a ! m x) (val x as C :=: A ! s x)) ->
          /\P /\m /\s (P -> X m s -> X m (FACT P ! s)) ->
          /\A,C /\m /\s
                (X m (s A) -> pas_dans_sig C (s A) ->
                   X m (TYPEDEF t as C :=: A ! s t)) ->
        X m S)
 : (type -> type -> type) -> ((type -> type) -> type -> prop) -> prop


(* typage des structures *)

Typ_inh = /\m /\S:(Typ_struct m)  INH (Struct ! m) (Sig ! S)
```

Le fichier `intro_modules.phx` contient les règles de typage proprement dite, et une forme générale pour la règle de sous-typage :

```
(* regles d'introduction *)

Typ_struct.end.intro = Typ_struct endStruct endSig : theorem

Typ_struct.let.intro =   /\a,A /\m /\s /\C
           (INH a A -> Typ_struct (m a) (s a) -> stnat_log C ->
              pas_dans_sig C (s a) ->
              Typ_struct (let x as C :=: a ! m x) (val x as C :=: A ! s x))


typ_struct.fact.intro =   /\P /\m /\s (P -> Typ_struct m s -> Typ_struct m (FACT P ! s))


typ_struct.typedef.intro =   /\A,C /\m  /\s
         (Typ_struct m (s A) -> pas_dans_sig C (s A) ->
            Typ_struct m (TYPEDEF t as C :=: A ! s t))

(* sous-typage *)

sous_type_signatures =
   /\s1,s2 (/\T /\X:(s1 T)  s2 T X -> (Sig ! s1 subset Sig ! s2))
```

Pour finir, le fichier `exemple_module.phx` contient un exemple de typage de module avec sous-typage :

```
(* exemple de typage *)

exemple1 =
   /\a,A,f
     (INH a A -> INH f (A |-> A |-> A) -> FA_o x :: A :: f @@ x @@ a = x ->
        INH
(Struct !
let x as C2 :=: a !
let y as C3 :=: f !
endStruct)

(Sig !
TYPE E as C1 !
val aa as C2 :=: E !
val ff as C3 :=: E |-> E |-> E !
FACT FA_o x :: E :: ff @@ x @@ aa = x !
endSig))
```

```
(* exemple de signature *)

groupe =
   \E

   Sig !
    val e0 as e :=: E !
    val plus as C2 :=: Pi x :: E :: Pi y :: E :: E !
    val op as C3 :=: Pi x :: E :: E !
    FACT   FA_o x :: E :: (plus @@ x) @@ e0 = x !
    FACT   FA_o x :: E :: (plus @@ x) @@ op @@ x = e0 !
   endSig

: type -> type
```

C.5 Exemples dans le langage

Nous présentons ici quelques exemples traités dans le langage "supérieur" :

C.5.1 Syntaxe

Le fichier grammaire.phx contient la grammaire du langage, et le fichier types.phx, dont voici quelques extraits, des règles de typage :

```
(*  regles de base *)

Pi_t.intro =   /\f /\a /\b (/\x ;; a  f x ;; b x -> fun x ==> f x ;; Pi_t x :: a :: b x)

Pi_t.elim = /\A /\B /\t,u (t ;; A -> u ;; Pi_t x :: A :: B x -> u @ t ;; B u)

Pi_o.intro = /\a /\p (/\x ;; a  p x -> Pi_o x :: a :: p x)

Pi_o.elim = /\A /\P /\t ;; A  (Pi_o x :: A :: P x -> P t)

(* schema de comprehension *)
$SC : type -> (prog -> prop) -> type

SC.intro = /\x /\A /\P (x ;; A -> P x -> x ;; SC r :: A :: P r)

SC.elim = /\A /\P /\t /\K ((t ;; A -> P t -> K) -> t ;; SC x :: A :: P x -> K)

(* sous-typage *)

Pi.Sub =   /\A,B,A',B'  (A' <: A -> /\x ;; A'  B' <: B -> Pi_t x :: A :: B <: Pi_t x :: A' :: B')

(* produit *)

prod.intro = /\A,B /\p,q (p ;; A -> q ;; B -> p , q ;; A X B) : theorem

(* cases *)

cases.of.motif =   /\p /\f /\B /\C0 /\M /\T,A
               (p ;; C0 OF A || T -> C0 notin T ->
                  /\x ;; A
                    (f x ;; B (C0 [ x ]) -> /\y ;; T  (cases y of M y) ;; B y
                     -> (cases p of C0 [[x ]] --> f x | M p) ;; B p)) :

(* types inductifs *)
```

```
$t : (type -> type) -> type

Alg = \F (/\x typsum F x & /\x,y (x <: y -> F x <: F y)) : (type -> type) -> prop

induc =   /\F /\P
       (/\A (Pi_o x :: A :: P x -> A <: (tx F x) -> Pi_o x :: F A :: P x) ->
          Pi_o x :: tx F x :: P x)

rec_mu =   /\F /\B /\t
         (/\f  /\A
              (f ;; Pi_t x :: A :: B x -> A <: (tx F x) -> t f ;; Pi_t x :: F A :: B x) ->
            rec fun f ==> t f ;; Pi_t x :: tx F x :: B x)
```

C.5.2 Exemples

Le fichier `booleens.phx` contient les spécifications des fonctions “not” et “ifthenelse” telles que vues dans la thèse :

```
(* booleens a la systeme F et specifications de not et ifthenelse *)

true = fun x ==> fun y ==> x : prog

false = fun x ==> fun y ==> y : prog

bool_f = #/\A Pi_t x :: A :: Pi_t y :: A :: A : type

Bool = SC x :: bool_f :: (x = true or x = false) : type

not = fun b ==> (b @ false) @ true : prog

not.spec =  not ;;
     Pi_t b :: Bool ::
       SC r :: Bool :: ((b = true -> r = false) & (b = false -> r = true))

$if = \b,x,y ((b @ x) @ y) : prog -> prog -> prog -> prog

ifthenelse.spec =   /\b,x,y  /\A
       (b ;; Bool -> x ;; A -> y ;; A ->
          (if b then x else y) ;;
            SC r :: A :: ((b = true -> r = x) & (b = false -> r = y)))
```

Le fichier `naturals.phx` contient quelques typages et specifications de fonctions usuelles sur les entiers :

```
nat = tx Z NIL || S OF x : type

(* addition *)

plus =
   rec
     fun plus0 ==>
       fun n ==>
         fun m ==> cases n of Z --> m | S [[y ]] --> S [ (plus0 @ y) @ m ]
 : prog


plus.type = plus ;; Pi_t n :: nat :: Pi_t m :: nat :: nat : theorem

plus.comm = Pi_o n :: nat :: Pi_o m :: nat :: ((plus @ n) @ m = (plus @ m) @ n)
```

```
(* comparaison *)

leq =
   rec
     fun leq0 ==>
       fun n ==>
         fun m ==>
           cases n of
             Z --> true |
             S [[x ]] -->
               cases m of Z --> false | S [[y ]] --> (leq0 @ x) @ y :


leq.type = leq ;; Pi_t n :: nat :: Pi_t m :: nat :: Bool
```

Le fichier `listes.phx` contient quelques typages et specifications de fonctions usuelles sur les listes :

```
(* type des listes *)

liste = \A ţx N NIL || C OF A X x : type -> type

tail = fun l ==> cases l of N --> N [] | C [[u ]] --> snd u : prog

tail.type = tail ;; #/\A Pi_t y :: liste A :: liste A

hd = fun l ==> cases l of C [[u ]] --> fst u : prog

(* la prise de tête est une fonction partielle *)

head.spec = hd ;; #/\A Pi_t y :: SC r :: liste A :: ~ r = N [] :: A


(* totalite du tri par insertion parametre par une fonction d'ordre *)

insert =
   fun less ==>
     rec
       fun ins ==>
         fun l ==>
           fun x ==>
             cases l of
               N --> C [ x , N [] ] |
                 C [[y ]] --> if (less @ x) @ fst y then
                 C [ x , l ] else C [ (fst y) , (ins @ snd y) @ x ] :
prog

sort =
   fun less ==>
     rec
       fun sor ==>
         fun l ==>
           cases l of
             N --> N [] | C [[y ]] --> ((insert @ less) @ sor @ snd y) @ fst y
 : prog

total_insert =
   insert ;;
     #/\A
       Pi_t less :: Pi_t x :: A :: Pi_t y :: A :: Bool ::
         Pi_t l :: liste A :: Pi_t x :: A :: liste A
```

```
total_sort =
   sort ;;
     #/\A
       Pi_t less :: Pi_t x :: A :: Pi_t y :: A :: Bool ::
         Pi_t l :: liste A :: liste A
```

Index

Bibliographie

[1] J.R. Abrial : *The B-Book, Assigning programs to meanings.* Cambridge University Press, 1996.

[2] David Aspinall : Subtyping with singleton types. *In Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.

[3] Henk Barendregt : *The Lambda Calculus.* North Holland, 1981.

[4] Henk Barendregt et Freek Wiedjik : The challenge of computer mathematics. soumis à Royal Society, 2005.

[5] Sylvain Baro : *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML.* Thèse de doctorat, Université Paris 7, 2003.

[6] Stefano Berardi : Pruning simply-typed λ-terms. Rapport technique, University of Torino, 1997.

[7] Chantal Berline : Cours de dea.

[8] Yves Bertini : *La notion d'indéfini en lambda calcul.* Thèse de doctorat, Université de Savoie, 2005.

[9] Luca Boerio : *Optimizing programs extracted from proofs.* Thèse de doctorat, University of Torino, 1995.

[10] Gérard Boudol : The recursive record semantics of objects revisited. Rapport technique 4199, INRIA, Juin 2001.

[11] Luca Cardelli et Xavier Leroy : Abstract types and the dot notation. Rapport technique 56, Digital Equipment Corporation SRC, March 1990.

[12] Luca Cardelli, Simone Martini, John C. Mitchell et Andre Scedrov : An extension of system f with subtyping. *Information and Computation*, 109:4–56, 1994.

[13] Luca Cardelli et John C. Mitchell : Operations on records. *Theoretical aspects of Object-Oriented Programming*, pages 295–350, 1994.

[14] Emmanuel Chailloux, Pascal Manoury et Bruno Pagano : *Développement d'applications avec Objective Caml.* O'Reilly, 2000.

[15] Thomas C.Hales : A computer verification of the Kepler Conjecture. *In Proceedings of the ICM*, volume 3, pages 795–804, Beijing, 2002.

[16] William R. Cook, Walter L. Hill et Peter S. Canning : Inheritance is not subtyping. *JJJJJ*, page PPPP, AAAA.

[17] René Cori et Daniel Lascar : *Logique Mathématique (I et II)*. Masson, 1993.

[18] Judicaël Courant : A module calculus for pure type systems. *In Typed Lambda Calculi and Applications 97*, pages 112–128. LNCS-Springer-Verlag, 1997.

[19] René David, Karim Nour et Christophe Raffalli : *Introduction à la logique, Théorie de la démonstration*. Dunod, 2001.

[20] Gilles Dowek : *La logique*. 1995.

[21] Derek Dreyer : Moscow ml's higher-order modules are unsound. TYPES electronic forum, 2002.

[22] Derek Dreyer, Karl Crary et Robert Harper : A type system for higher-order modules. *In Thirtieth ACM SIGPLAN Symposium on Principles of Programming Languages POPL'03*, 2003.

[23] Jean-Christophe Filliâtre : *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, 1999.

[24] Jean-Yves Girard, Yves Lafont et Paul Taylor : *Proofs and Types*. Cambridge University Press, 1989.

[25] Robert Harper et Mark Lillibridge : A type theoretic approach to higher-order modules with sharing. *In 21st symposium Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[26] Arend Heyting : *Mathematische Grundlagenforschung, Intuitionismus, Beweistheorie*. Springer-Verlag, 1934.

[27] C. A. R. Hoare : An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, 1983.

[28] W. A. Howard : The Formulae-As-Types Notion Of Construction. *In* J. P. Seldin et J. R. Hindley, éditeurs : *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.

[29] S. Kahrs, D. Sannella et A. Tarlecki : The definition of extended ml : a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[30] Andrey Nikolaevich Kolmogorov : Zur Deutung der intuitionistischen Logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

[31] Jean-Louis Krivine : *Théorie Axiomatique des Ensembles*. Presses Universitaires de France, 1972.

[32] Jean-Louis Krivine : *Lambda calcul, Types et Modèles*. Masson, 1990.

[33] Jean-Louis Krivine : Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.

[34] René Lalement : *Logique Réduction Résolution*. Masson, 1990.

[35] Xavier Leroy : Manifest types, modules, and separate compilation. *In 21st symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[36] Xavier Leroy : Systèmes de modules - notes de cours de l'ejcp, 2003.

[37] Alexandre MIQUEL : *Le calcul des constructions implicites, syntaxe et sémantique.* Thèse de doctorat, Université Paris VII, 2001.

[38] John C. MITCHELL et Gordon D. PLOTKIN : Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[39] Catherine PARENT : *Synthèse de preuves de programmes dans le calcul des constructions inductives.* Thèse de doctorat, Ecole normale supérieure de Lyon, 1995.

[40] Michel PARIGOT : $\lambda\mu$ calculus : an algorithmic interpretation of classical natural deduction. *Springer Lecture Notes in Computer Science*, 624:190–201.

[41] Michel PARIGOT : Recursive programming with proofs. *Theorical Computer Science*, 94:335–356, 1992.

[42] Catherine PAULIN-MOHRING : *Extraction de Programmes dans le Calcul des Constructions.* Thèse de doctorat, Université Paris VII, 1989.

[43] Christine PAULIN-MOHRING, Benjamin WERNER, Bruno BARRAS, Hugo HERBELIN, Jean-Christophe FILLIÂTRE et Claude MACHÉ : Cours 2-7 du master de recherche en informatique 2004-2005.

[44] Lancelot PECQUET : Cours de logique de licence, univ. paris xii, 2003.

[45] Benjamin C. PIERCE et David N. TURNER : Simple type-theoretic foundations for object-oriented programming. *J. Functiopnnal Programming*, 1993.

[46] François POTTIER : Notes du cours de DEA "Typage et Programmation", 2002.

[47] Christophe RAFFALLI : The PhoX proof assistant (version 0.84pre.041022). http ://www.lama.univ-savoie.fr/ raffalli/phox.html.

[48] Christophe RAFFALLI : *L'arithmétique fonctionnelle du second ordre avec points fixes.* Thèse de doctorat, Université Paris VII, 1994.

[49] Christophe RAFFALLI : System ST, beta-reduction and completeness. *LICS proceedings*, 2003.

[50] Christophe RAFFALLI : System ST, toward a type system for extraction and proof of programs. *Archive for Pure and Applied Logic*, 122, 2003.

[51] Christophe RAFFALLI et Frédéric RUYER : Realizability of the axiom of choice in HOL : an analysis of Krivine's work.

[52] Claudio RUSSO : *Types for modules.* Thèse de doctorat, Edinburgh University, 1998.

[53] Dana SCOTT.