Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting

Guillaume Genestier

Thèse de doctorat
NNT : 2020UPASG045

université PARIS-SACLAY

Terminaison en présence de types dépendants et encodage par réécriture d'une théorie des types extensionelle avec polymorphisme d'univers

Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche : Université Paris-Saclay, Inria, Inria Saclay-Île-de-France, 91120, Palaiseau, France
Référent : École Normale Supérieure Paris-Saclay

Thèse présentée et soutenue à Paris-Saclay, le 10 décembre 2020, par

Guillaume GENESTIER

Composition du Jury

Évelyne CONTEJEAN Directrice de recherche, CNRS	Présidente
Thierry COQUAND Professor, University of Gothenburg	Rapporteur & Examinateur
Ralph MATTHES Chargé de recherche, CNRS	Rapporteur & Examinateur
Delia KESNER Professeure, Université de Paris	Examinatrice
Cynthia KOP Assistant Professor, Radboud University Nijmegen	Examinatrice
Aart MIDDELDORP Professor, University of Innsbruck	Examinateur

Direction de la thèse

Frédéric BLANQUI Chargé de Recherche, Inria Saclay	Directeur de thèse
Olivier HERMANT Professeur, École des Mines de Paris	Co-Directeur de thèse

Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting

Guillaume Genestier

September 2020

Contents

Remerciements

These acknowledgements are in French. If your name is cited in this short text, I would like to deliver you a heartfelt thank.

If you think your name should be in this text, then I probably forgot it, this is not personal, I apologise and also deliver you a heartfelt thank.

Je souhaite tout d'abord remercier Thierry Coquand et Ralph Matthes pour avoir accepté d'être les rapporteurs de cette thèse, qui n'est probablement pas le manuscript le plus accueillant pour un lecteur que vous puissiez imaginer. De plus, les nombreux échanges que nous avons eus ont grandement contribué à améliorer ce document.

Je suis également très honoré, qu'Évelyne Contejean, Delia Kesner, Cynthia Kop et Aart Middeldorp aient accepté de lire cette thèse et de faire partie du jury. Cela me touche que ces chercheurs, dont le travail a grandement influencé le mien et dont j'ai lu les articles à de multiples reprises au cours de ces trois années, se soit intéressé de près à mon travail.

Naturellement, je voudrais remercier Frédéric Blanqui et Olivier Hermant, pour m'avoir guidé pendant trois ans.

Frédéric, ton immense connaissance du domaine force l'admiration, et je dois admettre avoir toujours été heureux de ressortir de ton bureau avec de multiples références à découvrir. Mais, plus important que cela, ta capacité à ne jamais oublier l'essentiel et à fixer un cap m'a grandement profité. De plus, bien que nous n'ayons pas toujours été d'accord, la liberté que j'ai eu de ne pas suivre les chemins que tu me conseillais, tout en continuant de bénéficier de ton aide précieuse, a permis au travail que voici d'aboutir.

Olivier, tu as également toujours été là, prêt à passer des heures pour me faire expliquer les embryons d'idées que j'avais. De plus, tu m'as régulièrement permis de dépasser mes blocages pour produire quelque chose de positif. En particulier, il est très clair que sans ton secours (et tes dizaines d'heures de relecture), j'aurais mis plusieurs mois de plus à produire un texte d'une qualité moindre.

Ensuite, je veux naturellement remercier l'ensemble de Deducteam. J'ai été toujours ravi de faire partie de cette équipe, non seulement parce que le projet qu'elle porte m'enthousiasme, mais aussi pour les multiples discussions passionnantes que j'ai pu avoir avec ces membres. Gilles Dowek a toujours été disponible, que ce soit pour parler de problèmes scientifiques, de la vie de l'équipe ou de mon avenir. Les autres membres permanents de l'équipe, Bruno Barras, Valentin Blot, Guillaume Burel, Catherine Dubois et Jean-Pierre Jouannaud furent également toujours disponible pour répondre à mes questions ou expliquer leur travaux. Thida Iem, Emmanuelle Perrot et Adeline Lochet m'ont également toujours aidé, malgré ma faible appétence pour tout ce qui a trait aux tâches administratives. C'est grâce à elles que j'ai pu effectuer tous les déplacements que j'ai fait.

Mais une équipe de recherche est aussi constituée de membres éphémères. Je pense notamment aux différents post-docs passés dans l'équipe : Rodolphe Lepigre, Frank Slama, Michael Färber, Rehan Malak, Pierre Vial et Étienne Miquey. Même si j'ai eu plus d'interactions avec les

premiers cités qu'avec ceux arrivés dans l'équipe plus tard (les raisons à cela sont diverses), J'ai beaucoup apprécié chacune de ces rencontres. Je pense également aux divers stagiaires passés dans l'équipe, je vais malheureusement oublier des noms, mais je peux notamment citer Walid Moustaoui, Aristomenis Papadopulos, Jui-Hsuan Wu et Tristan Delort.

Je voulais également remercier les membres du LSV, Stéphane Demri qui dirigeait le LSV à l'époque et m'a accueilli, Serge Haddad, toujours prompt à se méler à nos discussions de pauses café (il faut dire que son bureau est idéalement placé pour cela), Hubert Comon et David Baelde, pour qui ce fut un véritable plaisir de faire les TDs de leur admirable cours de logique (Gilles Dowek doit également être cité à ce propos), Jean Goubault-Larrecq, dont la culture informatique semble sans fond, ou encore Hugues Moretto-Viry et Johnny Pinson, qui contrairement à moi savent se servir d'un ordinateur, même lorsqu'il n'est pas disposé à nous obéir. Je devrais citer encore de nombreux membres du laboratoire, j'espère que personne ne me tiendra rancune de cette liste pour le moins lacunaire.

Le LSV compte aussi de nombreux doctorants, avec qui ce fut toujours un grand plaisir d'aller boire des bières et de goûter (plus ou moins régulièrement). Là encore, il faudrait en citer une multitude, j'espère que personne ne s'offusquera de son absence dans cette liste. Adrien Koutsos, Charlie Jacomme et Mathieu Hilaire, qui nous ont fièrement représenté tous les mardis matin au conseil de direction du laboratoire, Simon Halfon, qui nous a reçu pour un repas de Noël mémorable (et un retour en voiture, non moins mémorable), Igor Khmelnitsky, qui nous a reçu pour une soirée jeux de société, qui fut l'occasion pour moi d'apprendre le mot "cauliflower", Mathilde Boltenhagen et Juraj Kolčák, qui ont oeuvré pour que le quatrième ne soit pas trop souvent oublié, et pour que l'on sorte boire des bières tous ensemble, Aliaume Lopez, toujours motivé, notamment pour jouer au jeu des mots, et enfin Jawher Jerray, qui a partagé un bureau avec des membres de Deducteam et a supporté notre tendance à nous regrouper longuement et bruyamment devant des tableaux blancs remplis de symboles cabalistiques.

Mais j'avais également un second laboratoire, le CRI des Mines, le peu de gens travaillant sur les mêmes thématiques que moi, combinés à la distance (Fontainebleau c'est loin) m'ont fait m'y rendre moins régulièrement. Malgré cela, je m'y suis toujours senti à la maison. Je voulais donc remercier François Irigoin, Corinne Ancourt, Fabien Coelho, Laurent Daverio, Emilio Gallego Arias, Pierre Jouvelot, Claire Medrala et Claude Tadonki pour leur accueil.

J'ai également passé un mois et demi à Göteborg, pour travailler avec Andreas Abel et Jesper Cockx. Je les remercie très chaleureusement pour leur invitation et leur accueil. Que ce soit pour découvrir la vie suédoise, ou pour démêler le code d'Agda, ils ont toujours été disponibles.

Je voulais également remercier Cécile Balkanski et Hélène Maynard, qui m'ont confié l'encadrement d'un groupe de TD pour leur cours d'initiation à la programmation. La confiance qu'elles m'ont fait me touche. Chantal Keller m'a permis d'avoir cette opportunité d'enseigner à l'IUT d'Orsay, je l'en remercie.

Tant que je parle d'enseignement, je voulais remercier la totalité des élèves que j'ai eu, aussi bien à l'ENS qu'à l'IUT. J'ai toujours été heureux d'aller en cours et les quelques heures par semaine que je passais en votre présence ont toujours fait partie des plus enthousiasmantes de ma semaine. Je vous souhaite à tous une brillante réussite.

À cette occasion, je voulais remercier tous les professeurs que j'ai rencontré au cours de ma scolarité et qui m'ont mené jusqu'à cette aboutissement de ma formation aujourd'hui avec cette thèse.

Enfin, je voulais remercier ceux qui ont été mes complices pendant ces trois années, et dont je peux dire qu'ils sont maintenant des amis, les autres doctorants de Deducteam. Je pense tout particulièrement aux trois qui ont partagé mon bureau pendant ces trois ans et avec qui j'ai fait le voyage en totalité : François Thiré, Gaspard Férey et Yacine El Haddad.

Je me souviens que François m'impressionnait lorsque je suis arrivé dans l'équipe en stage,

avec sa connaissance complète de Dedukti et son écosystème, et ses affirmations péremptoires pas toujours faciles à décrypter. Il s'avère qu'il est à l'origine de l'immense majorité des discussions passionnantes que j'ai eu sur les objectifs et évolutions du système que nous développons. Mais il ne faudrait pas en oublier nos multiples discussions cinémas, ainsi que nos quelques séances dans les salles obscures, pour voir des films souvent originaux.

Gaspard, toujours motivé pour bidouiller le code de Dedukti, quitte à faire des "sagouineries" (mot que j'ai réussi à faire entrer dans son vocabulaire, à force de qualifier ainsi ses expérimentations), fut également le meilleur compagnon de voyage (au sens propre comme au sens figuré) possible. Nos discussions, sur la complexité des institutions ou l'énergie de son chaton, ont fait passé en un éclair nos trajets de RER et fait s'éterniser nos pauses cafés. Je me languis d'assister à sa soutenance de thèse.

Yacine est moins bavard que nous trois, mais je sais qu'il m'aurait tout de même repris pour rappeler qu'il a commencé sa thèse trois mois après Gaspard et moi. Talentueux à de multiples jeux : les échecs (mon niveau ne m'a permis de juger sur pièce), le baby foot ou la coinche (dans ces deux cas-là, il est sans doute moins doué qu'aux échecs, mais il m'a battu à plate couture), toujours curieux et fin analyste des différences entre la France et l'Algérie, nos discussions sur l'origine des différentes fêtes ou l'organisation des études dans ces deux pays me manqueront. Je lui souhaite beaucoup de courage dans la fin de la rédaction de sa thèse, en cours au moment où j'écris ces lignes.

Deux nouveaux compagnons de voyage ont rejoint l'équipe au milieu de la traversée pour nous trois, il s'agit d'Émilie Grienenberger et Gabriel Hondet. Et même si j'ai du mal à comprendre la haine qu'a Gabriel pour la "Pop", leur arrivée fut non seulement l'occasion d'assister à des concerts où je ne serai jamais allé sinon, mais surtout la rencontre de partenaires de pause café incroyables, toujours prompt à se moquer de ma maladresse, mais surtout d'amis.

J'ai malheureusement moins connu Frédéric Gilbert, Guillaume Bury et Amélie Ledein, qui ont soit quitté l'équipe trop tôt, soit rejoint celle-ci trop tard, pour que nous ayons le temps de développer une relation privilégiée. Cependant, j'ai beaucoup apprécié les quelques discussions que j'ai eu avec chacun des trois.

Pour finir, je voulais remercier mes amis "Les Bolos.se.s" et surtout les "Vieux de la Vieille" (Adrien, Anna, Claire, Loïc, Clément, Iphigénie, Frédéric, Nicolas (Je sais bien que tu vas râler parce que j'ai écrit le prénom en entier), Séverine, Rémy, Thomas, Anouk, Jean-Baptiste et Laëtitia) ainsi que ma famille, particulièrement Geoffroy et Philippe (que j'appelle au quotidien "Joe" et "Papa") qui m'ont aidé à me changer les idées au quotidien et ont supporté ma fréquente mauvaise humeur pendant la rédaction de ce document.

Chapter 1

Introduction (en français)

Cette thèse d'informatique porte sur les démonstrations formelles.

1.1 La Logique : mathématiques ou informatique ?

Avant même de détailler plus précisément quels aspects des démonstrations formelles sont abordés dans ce texte, le lecteur néophyte peut se demander pourquoi l'étude des démonstrations formelles relève de l'informatique et non des mathématiques.

Il faut alors commencer par observer qu'historiquement, les pionniers de l'informatique que sont Turing, Church et Gödel, se sont posés la question de ce qui était "mécaniquement calculable", avant même l'existence d'ordinateurs à proprement parler. À sa naissance, l'informatique n'est donc pas la science qui s'intéresse au fonctionnement des ordinateurs, mais la science qui cherche à comprendre ce que veut dire "calculable par un procédé mécanique". De multiples modèles sont inventés pour expliciter un calcul mécanique. Citons par exemple, les fameuses "machines" de Turing, ou le λ-calcul, introduit par Church.

1.1.1 Un petit détour par la déduction naturelle

Jusque là, il semble assez naturel que la question de savoir ce qu'il est possible, ou impossible, de calculer avec une machine, soit une question qui fasse partie de l'informatique. Cela n'explique toujours pas pourquoi la question de la formalisation des démonstrations y est également associée. Pour répondre à cette interrogation, commençons par nous demander ce que veulent dire les énoncés mathématiques et plus précisément, quel est le sens associé aux connecteurs logiques : que signifie "A et B", "A ou B", "si A, alors B" quand A et B sont eux-mêmes des énoncés ?

Une première réponse, relativement fréquente, car enseignée dans les cours de mathématiques du secondaire, est de dire qu'un connecteur est défini par la fonction qui aux "valeurs de vérité" de A et B associe la "valeur de vérité" du nouvel énoncé. Ces fonctions sont souvent représentées par des tableaux, souvent appelés "tables de vérité", tels que celui-ci :

A	B	A et B	A ou B	si A alors B
Faux	Faux	Faux	Faux	Vrai
Faux	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Vrai	Faux
Vrai	Vrai	Vrai	Vrai	Vrai

Cependant, cette définition des connecteurs logiques n'est que peu satisfaisante, puisqu'elle ne dit absolument pas ce qu'est une preuve. Par conséquent, une autre démarche[1] fût adoptée par Jaśkowski, qui définit un système de preuves. Indépendamment, Gentzen introduit quelques années après un système similaire, qu'il baptise "déduction naturelle". Comme son nom l'indique, cette approche vise à simuler la démarche adoptée par un mathématicien lorsqu'il raisonne. Les connecteurs logiques sont donc définis par la façon dont ils sont manipulés, par des règles, nommées "règles d'inférence". Ces manipulations sont de deux types:

- on peut soit utiliser une hypothèse qui contient ce connecteur, on va alors chercher à casser l'hypothèse, on utilise alors les "règles d'élimination" du connecteur;
- soit chercher à obtenir une conclusion qui contient ce connecteur, on va alors chercher à assembler des hypothèses, on utilise pour cela les "règles d'introduction" du connecteur.

Par exemple, si une hypothèse dit que "A et B", alors, je peux en déduire "A". Cette utilisation d'un "et" présent dans une hypothèse correspond à la règle d'élimination du "et" dans la déduction naturelle : $\frac{A \text{ et } B}{A}$. La règle $\frac{A \text{ et } B}{B}$ est aussi une règle d'élimination du "et" dans la déduction naturelle.

De façon similaire, si d'un côté je sais "A", de l'autre je sais "B", je peux en conclure "A et B". Ce raisonnement correspond à la règle d'introduction du "et" dans la déduction naturelle : $\frac{A \qquad B}{A \text{ et } B}$.

Cependant le cas de l'implication nécessite de complexifier légèrement le système. En effet, utiliser une implication est relativement simple, avec les hypothèses "si A alors B" et "A", on peut en conclure "B", ce qui donne donc la règle d'élimination de l'implication $\frac{\text{si } A \text{ alors } B \qquad A}{B}$. Mais pour ce qui est de l'introduire, le formalisme tel qu'il est actuellement présenté est trop limité. En effet, pour prouver "si A alors B", on commence par admettre A, puis l'on raisonne, et lorsque notre raisonnement aboutit à B, on peut conclure "si A alors B". Il faut donc, tout au long de la preuve, se souvenir que l'on a admis A, pour pouvoir introduire l'implication finale lorsque le moment sera venu. Pour cela, on utilise le signe $\vdash$, appelé "thèse" ou "taquet", pour séparer les propositions admises de la conclusion. On peut alors écrire la règle d'introduction de l'implication : $\frac{A \vdash B}{\vdash \text{si } A \text{ alors } B}$

Cet ajout d'un contexte de propositions admises nous permet d'introduire la règle la plus simple, aux fondements de toutes les démonstrations, qui dit, sans autre hypothèse, "si j'admets A, alors j'ai A", ce qui donne, sous la forme de règle d'inférence la règle dite "axiome" : $\frac{}{A \vdash A}$.

Mais, on pourrait raisonnablement objecter que cette digression sur la démarche de définition et représentation des preuves n'explique toujours pas le lien qui peut exister entre une preuve représentée dans ces formalismes et la question des fonctions "mécaniquement calculables". Faisons donc un pas de plus vers la réponse à cette question et remarquons qu'un arbre de preuve fait mention un nombre important de fois des mêmes propositions. On pourrait essayer d'être plus synthétique en notant simplement la suite des règles utilisées.

Commençons par observer ce que cela donnerait sur un exemple[2] :

[1] Naturellement, ces deux démarches sont liées, et il est par exemple possible de passer des tables de vérité aux règles de la déduction naturelle, comme présenté dans [GH17].

[2] Bien que nous ayons écrit les énoncés en français jusqu'à ce point, nous nous permettrons d'utiliser les notations $A \wedge B$ pour désigner "A et B", $A \vee B$ pour "A ou B" et $A \Rightarrow B$ pour "si A, alors B", afin de limiter la taille de l'arbre présenté et de simplifier la lecture des suites d'implications, une phrase telle que "si (si A et B, alors C), alors si A, alors si B alors C" étant particulièrement difficile à comprendre lorsqu'elle est écrite ainsi.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A\wedge B)\Rightarrow C;A;B\vdash (A\wedge B)\Rightarrow C}\,\text{ax} \quad \dfrac{\dfrac{}{(A\wedge B)\Rightarrow C;A;B\vdash A}\,\text{ax} \quad \dfrac{}{(A\wedge B)\Rightarrow C;A;B\vdash B}\,\text{ax}}{(A\wedge B)\Rightarrow C;A;B\vdash A\wedge B}\,\wedge_I}{(A\wedge B)\Rightarrow C;A;B\vdash C}\,\Rightarrow_E}{(A\wedge B)\Rightarrow C;A\vdash B\Rightarrow C}\,\Rightarrow_I}{(A\wedge B)\Rightarrow C\vdash A\Rightarrow B\Rightarrow C}\,\Rightarrow_I}{\vdash ((A\wedge B)\Rightarrow C)\Rightarrow A\Rightarrow B\Rightarrow C}\,\Rightarrow_I$$

Cette preuve serait donc résumée par la suite de règle : $\Rightarrow_I (\Rightarrow_I (\Rightarrow_I (\Rightarrow_E (ax, (\wedge_I(ax, ax))))))$.

Cependant, une preuve de la formule $B \Rightarrow ((A\wedge B) \Rightarrow C) \Rightarrow A \Rightarrow C$ est constituée exactement de la même suite de règles. On voudrait donc savoir plus précisément comment on peut relier les propositions qui apparaissent dans les axiomes et celles qui sont introduites lors de l'utilisation de la règle $\Rightarrow_I$.

On pourrait alors vouloir nommer les hypothèses, afin de s'y référer lors de l'utilisation des règles, comme cela[3] :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash (A\wedge B)\Rightarrow C}\,\text{ax}_x \quad \dfrac{\dfrac{}{\Gamma\vdash A}\,\text{ax}_y \quad \dfrac{}{\Gamma\vdash B}\,\text{ax}_z}{\Gamma\vdash A\wedge B}\,\wedge_I}{x:(A\wedge B)\Rightarrow C;y:A;z:B\vdash C}\,\Rightarrow_E}{x:(A\wedge B)\Rightarrow C;y:A\vdash B\Rightarrow C}\,\Rightarrow_{Iz}}{x:(A\wedge B)\Rightarrow C\vdash A\Rightarrow B\Rightarrow C}\,\Rightarrow_{Iy}}{\vdash ((A\wedge B)\Rightarrow C)\Rightarrow A\Rightarrow B\Rightarrow C}\,\Rightarrow_{Ix}$$

Le résumé de cette preuve devient alors $\Rightarrow_{Ix} (\Rightarrow_{Iy} (\Rightarrow_{Iz} (\Rightarrow_E (ax_x, (\wedge_I(ax_y, ax_z))))))$. Il faut noter que maintenant, pour prouver $B \Rightarrow ((A \wedge B) \Rightarrow C) \Rightarrow A \Rightarrow C$, la suite de règles est $\Rightarrow_{Ix} (\Rightarrow_{Iy} (\Rightarrow_{Iz} (\Rightarrow_E (ax_y, (\wedge_I(ax_z, ax_x))))))$, les indices permettent donc de distinguer les preuves de ces deux énoncés.

1.1.2 λ-calcul et preuves : Le typage

Après cette très brève introduction à la théorie de la démonstration, il est temps de remarquer que toutes les règles d'inférence présentées jusqu'à présent pourraient aussi être utilisées pour typer un programme.

Typer un programme, c'est se demander de quelle nature sont les entrées ou les sorties d'un programme. Par exemple, la fonction $f : x \mapsto \ln(x)$ est une fonction qui à un réel strictement positif associe un réel. En mathématiques, on note souvent cela en indiquant les ensembles de départ et d'arrivée de la fonction. Ici $f : \mathbb{R}_+^* \to \mathbb{R}$.

En général, la première rencontre avec la notion de typage vient avec la notion d'homogénéité en physique. Si m désigne ma masse et ν ma fréquence cardiaque, cela n'a pas de sens de se demander ce que vaut $m + \nu$, dans le sens où cette grandeur ne mesure rien[4]. Cela est dû au fait que le symbole "+" attend deux arguments de même nature, et retourne un terme de cette même nature.

De la même façon, écrire sin(vrai) n'a pas de sens, car la fonction sinus attend un nombre en entrée et renvoie un nombre. En revanche, cette information nous permet d'affirmer, que

[3] Pour des questions de place, on note Γ le contexte $x : (A \wedge B) \Rightarrow C; y : A; z : B$

[4] Il existe sans aucun doute, dans la très riche littérature médicale ou paramédicale, un indicateur de la bonne (ou mauvaise) santé d'un patient qui utilise cette somme. Quelque soit la pertinence de cet indicateur hypothétique, il n'en demeure pas moins que cette somme n'a pas de réalité physique.

puisque 3 est un nombre, sin(3) est aussi un nombre. Plus généralement, si f est une fonction de A vers B et que x appartient à A, alors $f(x)$ appartient à B. Dans l'esprit des règles d'inférences introduites dans la section précédente pour la démonstration formelle, on pourrait introduire une règle d'inférence pour le typage de la forme : $\dfrac{f \in A \to B \qquad x \in A}{f(x) \in B}$.

Pour le produit cartésien, on peut avoir exactement la même démarche et constater que si a appartient à A et b à B, alors la paire (a, b) appartient à $A \times B$. Ce qui donne la règle : $\dfrac{a \in A \qquad b \in B}{(a, b) \in A \times B}$.

On remarque alors assez bien que les règles d'utilisation de la flèche et d'élimination de l'implication sont analogues, de même que les règles d'introduction du produit cartésien et de la conjonction. Cette analogie ne se limite pas à ces deux règles, ce qui nous amène donc à considérer les propositions comme des types, et à maintenant avoir des règles d'inférence qui contiennent non seulement un contexte et une proposition, comme ce qui a été présenté à la section précédente, mais aussi un terme, comme cela a été fait pour le typage.

On a alors les règles suivantes[5] :

$$\frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B} \Rightarrow_e \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \wedge B} \wedge_i$$

Pour les règles d'introduction de l'implication et d'élimination de la conjonction, il va falloir inventer de nouveaux symboles, pour créer une implication à partir d'une variable dans le contexte, ou pour indiquer laquelle des deux propositions de la conjonction l'on veut sélectionner. Comme l'implication correspond à la flèche fonctionnelle, nous cherchons donc à construire une fonction. Pour cela, nous allons reprendre la notation du λ-calcul, et écrire $\lambda x.t$ pour introduire l'hypothèse correspondant à la variable x. La conjonction, quant à elle, est un produit cartésien, elle s'élimine donc avec des projections, que nous noterons π_i. Ces règles sont donc :

$$\frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B} \Rightarrow_i \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_1 t : A} \wedge_{e1} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_2 t : B} \wedge_{e2}$$

1.1.3 Calcul et preuves : la correspondance de Curry-Howard

Nous avons donc constaté que typer un programme ou faire une démonstration formelle correspond à la même construction. Il n'en reste pas moins qu'un ordinateur vise avant tout à effectuer des calculs. Le λ-calcul étant un modèle du comportement d'un ordinateur, il contient un mécanisme de calcul : la β-réduction, qui revient à dire qu'appliquer une fonction à un argument, c'est avant tout remplacer une variable par l'argument dans la définition de la fonction. Ainsi, lorsque l'on définit $f : x \mapsto x^2 + 2 - x$, puis que l'on cherche à calculer $f(5)$, on commence par remplacer x par 5 dans le "corps" de f, donnant $f(5) = 5^2 + 2 - 5$.

Remarquons que si l'on écrit cela sous forme d'arbre, on a :

$$\frac{\dfrac{x : A \vdash t : B}{\vdash \lambda x.t : A \Rightarrow B} \Rightarrow_i \qquad \vdash u : A}{\vdash (\lambda x.t)\, u : B} \Rightarrow_e$$

pour typer le terme à calculer.

Un tel arbre s'appelle en logique une "coupure", qui revient à dire que si un lemme a été prouvé, il peut ensuite être utilisé comme hypothèse dans les preuves des théorèmes qui suivent.

[5] On garde les notations de la logique $\Rightarrow$ et $\wedge$ plutôt que celle traditionnelle en mathématiques $\to$ et $\times$.

Intuitivement, on voudrait dire que la coupure est uniquement là pour des questions de commodité[6], mais qu'elle ne change pas l'ensemble des énoncés prouvables, puisqu'il suffit de recopier intégralement la preuve du lemme à chaque fois qu'on l'utilise, pour se passer de cette règle. En pratique, faire cette transformation, c'est exactement dire que partout là où l'hypothèse associée à la variable x est utilisée, on la remplace par la preuve u.

Là encore, tout correspond parfaitement. En expliquant comment éliminer les coupures dans une preuve, nous avons décrit la substitution qui correspond à β-réduire la conclusion de cet arbre de preuve.

Ainsi, l'étude des preuves formelles et des programmes informatiques se rejoignent dans l'étude des systèmes de typage pour le λ-calcul. Cette identité entre systèmes de types (pour les programmes) et formalismes de déduction (pour les preuves) est couramment nommée la *correspondance de Curry-Howard*[7].

1.1.4 Plusieurs logiques

Jusqu'à présent, nous avons parlé de "la" logique, comme s'il s'agissait d'un objet unique et bien identifié. Ce n'est pas si simple. Tout d'abord, remarquons que dans la section précédente, nous n'avons parlé que de la déduction naturelle propositionnelle, c'est-à-dire qu'il n'y avait pas de notion de quantification, et que tous les types "atomiques" étaient simplement constitué d'une proposition. Mais même pour exprimer un énoncé aussi simple que la commutativité de l'addition, qui est un résultat enseigné très tôt dans un cursus mathématique (à l'âge de 6 ans en France), il est déjà nécessaire de pouvoir quantifier universellement des énoncés et d'avoir des prédicats. Ainsi, la commutativité de l'addition s'écrit $\forall x \in \mathbb{N}, \forall y \in \mathbb{N}, x + y = y + x$. De nombreuses logiques se distinguent les unes des autres justement par les situations où l'on est autorisé à quantifier pour former un énoncé syntaxiquement valide de la logique.

- Ainsi, la logique d'ordre supérieure autorise à quantifier sur des types fonctionnels et sur des prédicats, ce qui n'est pas le cas de la logique du premier ordre. Par exemple, le principe de récurrence sur les entiers s'écrit $\forall P, P(0) \Rightarrow (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n, P(n)$. Il nécessite de quantifier sur le prédicat P qui dépend d'un entier. Par conséquent, on ne peut pas écrire cette formule dans la logique du premier ordre[8].

- Certaines logiques incluent du polymorphisme, c'est-à-dire la possibilité de quantifier universellement sur l'ensemble des types pour construire d'autres types. C'est particulièrement utile en programmation, où l'on ne souhaite pas définir un type des listes pour chaque type des éléments qu'elle peut contenir. Ainsi, la fonction `map`[9] a pour type, en OCaml, `('a -> 'b) -> 'a list -> 'b list`, où les symboles `'a` et `'b` peuvent être instanciés par n'importe quel type.

- Des logiques autorisent les types dépendants. Dans l'énoncé de la commutativité de l'addition, la proposition $x + y = y + x$ dépendait des variables quantifiées précédemment. Comme la correspondance de Curry-Howard tend à nous faire confondre propositions et types de données, on peut vouloir également avoir des types qui dépendent des variables quantifiées précédemment. Cela nous permet par exemple de déclarer le type des vecteurs de taille n, où n désigne un entier.

[6] En l'absence de lemmes, aucun texte mathématique n'est lisible par un humain en un temps raisonnable.

[7] Certains y ajoutent les noms de Lambek ou de Bruijn.

[8] L'axiomatique de Peano contourne cette difficulté en faisant du principe de récurrence un "schéma d'axiomes", c'est-à-dire en disant "il y a une infinité d'axiomes de la forme $P(0) \Rightarrow (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n, P(n)$, un pour chaque P possible".

[9] Fonction qui applique une fonction à tous les éléments d'une liste.

Naturellement, il existe aussi des différences entre les logiques qui ne se réduisent pas à la question de savoir si une quantification est ou non autorisée. Nous allons justement évoquer l'un de ces enrichissements dans la section suivante qui portera sur le vérificateur de types Dedukti.

1.1.5 Vrai ou prouvable ?

Avant cela, il faut éliminer une tentation que le lecteur pourrait avoir : celle de penser que pour créer la logique la plus "puissante", il suffit d'inclure toutes les fonctionnalités que l'on connaît dans une même logique.

En un sens, on ne pourrait pas donner tort au lecteur qui ferait cette proposition, une telle logique permettrait effectivement de prouver énormément de choses. Un peu trop même. En fait, une telle logique serait "incohérente", c'est-à-dire qu'elle permettrait de tout prouver, même ce qui "est faux".

Pour être précis, il y a maintenant deux notions qui cohabitent:

- la syntaxe, qui désigne le système de démonstration et est associée à la notion de proposition "prouvable",
- la sémantique, qui désigne le modèle concret des axiomes et est associée à la notion de proposition "valide".

Une logique est dite "cohérente" si le système de démonstration est "correct", c'est-à-dire qu'elle admet des modèles non-triviaux et qu'il n'est possible de prouver que des énoncés qui sont vrais dans tous les modèles.

Les systèmes de types incohérents sont très intéressants, en particulier en programmation. Cependant, pour un assistant de preuves destiné à augmenter la confiance que l'on a dans la validité de résultats mathématiques, ils sont beaucoup moins désirables. Si Thomas Hales avait expliqué avoir réussi à obtenir une preuve de la conjecture de Kepler [HAB+15, Hal05] dans un système dans lequel il est possible de tout prouver, même ce qui est faux, il y a fort à parier que la communauté mathématique ne se serait pas intéressée à son résultat.

1.2 Le Projet Dedukti

Maintenant que des rudiments de théorie de la démonstration ont été énoncés, afin d'expliquer le lien entre calcul et preuves et pourquoi l'étude de la logique fait partie de l'informatique, il est temps de situer plus précisément le contexte dans lequel cette thèse a été effectuée.

L'équipe au sein de laquelle j'ai effectuée mes recherches développe Dedukti, un vérificateur de preuves pour une logique appelée le $\lambda\Pi$-calcul modulo réécriture.

1.2.1 Pourquoi le $\lambda\Pi$-calcul modulo réécriture ?

Nous l'avons vu, il existe de multiples logiques, il est donc naturel de se demander pourquoi c'est celle-ci qui a été choisie pour Dedukti.

Après avoir évoqué de nombreuses variantes, le lecteur pourrait se sentir soulagé de savoir que l'on va dorénavant se restreindre à l'étude d'une seule d'entre elles.

Mais la situation est un peu plus complexe que cela. En effet, comme il existe de multiples logiques, il existe de multiples assistants de preuves, implémentant chacun une logique différente. Mais cela signifie que chaque résultat, aussi universel soit-il, doit être prouvé dans tous les systèmes de façon indépendante. Cette situation étant très frustrante, l'objectif de l'équipe est

de créer LOGIPEDIA [Log], une encyclopédie de preuves utilisables dans de multiples systèmes, et ce quelque soit l'assistant de preuves dans lequel la preuve a été produite à l'origine.

Pour faire cela, toutes les preuves sont dans un premier temps traduites vers un unique cadre logique : le $\lambda\Pi$-calcul modulo réécriture et son implémentation DEDUKTI.

1.2.2 Sur les cadres logiques

Ici une question émerge naturellement. Nous avons vu dans la Section 1.1.5, qu'il n'était pas possible d'avoir une logique à la fois cohérente et plus forte que toutes les autres. Or, maintenant je prétends que l'on va traduire des preuves dans le $\lambda\Pi$-calcul modulo réécriture, et ce quelque soit la logique d'origine dans laquelle elles ont été produites.

La conclusion naturelle que l'on pourrait tirer de cela est que le $\lambda\Pi$-calcul modulo réécriture est une logique incohérente. Fort heureusement, ce n'est pas le cas en général. La subtilité se trouve dans la possibilité d'encoder dans le $\lambda\Pi$-calcul modulo réécriture d'autres logiques, et ainsi de transformer une preuve du théorème T dans la logique $\mathcal{L}$ en une preuve dans $\lambda\Pi$ modulo réécriture de l'énoncé "T est prouvable dans $\mathcal{L}$".

Les logiques particulièrement adaptées pour y encoder d'autres logiques sont appelées des "cadres logiques". Le premier système logique à avoir été introduit dans le but d'y définir de multiples théories est la "logique des prédicats" [HA28], dans laquelle on peut définir aussi bien la géométrie euclidienne (axiomes de Hilbert), l'arithmétique (axiomatique de Peano) ou la théorie des ensembles (théorie de Zermelo-Fraenkel).

Même si disposer d'un cadre unique, et donc de définitions uniformisées des connecteurs ($\wedge$, $\vee$, $\Rightarrow\dots$), des quantificateurs ($\forall$, $\exists\dots$) et des notions de modèles et preuves, constitue une avancée majeure, en permettant notamment de disposer de théorèmes de correction et de complétude généraux, la logique des prédicats présente quelques défauts, qui incitent à chercher à encoder les preuves dans un autre cadre logique.

Tout d'abord, en logique des prédicats, il n'est possible de lier une variable autrement qu'en utilisant les quantificateurs $\exists$ et $\forall$. Par exemple, il n'est pas possible de définir un symbole $\mapsto$ qui lierait une variable dans son argument. Pour surmonter ce problème, plusieurs cadres logiques ont été proposés, comme λ-PROLOG [Mil91] ou le cadre logique d'Édimbourg [HHP93], souvent abrégé en LF (*Logical Framework*) et qui est aussi appelé $\lambda\Pi$-calcul.

Un autre défaut majeur de la logique des prédicats est que ce qui relève du calcul n'est pas séparé des étapes de déduction, ainsi dans l'arithmétique de Peano, obtenir l'égalité 2+3=5 nécessite de nombreuses étapes de preuves, alors qu'un simple calcul, à la portée d'un enfant de 6 ans, permet d'obtenir le résultat. La déduction modulo théorie [DHK03] permet d'introduire des étapes de calcul au sein des preuves.

1.2.3 Qu'est-ce que le $\lambda\Pi$-calcul modulo réécriture ?

Vous l'aurez compris, le $\lambda\Pi$-calcul modulo réécriture est la combinaison du cadre logique d'Édimbourg et de la déduction modulo réécriture.

Le cadre logique d'Édimbourg, ou $\lambda\Pi$-calcul, est un système relativement simple. Il s'agit d'un λ-calcul dont le système de types ne présente qu'une seule des fonctionnalités évoquées Section 1.1.4, les types dépendants.

Comme indiqué précédemment, le mécanisme des types dépendants permet de lier une variable dans un type, pour construire, non pas une proposition, mais un nouveau type[10]. En

[10]Certes, la correspondance de Curry-Howard tend à identifier types et propositions, cependant, il semblerait incongru de dire que `List n` ou $\mathcal{M}_n(\mathbb{R})$ sont valides pour dire qu'il s'agit de types non-vides. Nous nous contenterons de cette approche intuitive de ce qu'est une proposition (essentiellement un type qui correspond à un

mathématiques, cette construction est souvent utilisée, par exemple pour définir la fonction $I : n \in \mathbb{N} \to \mathcal{M}_n(\mathbb{R})$ qui à n associe la matrice identité de taille n, il est nécessaire de faire dépendre l'ensemble d'arrivée de la fonction (les matrices carrées de taille n) de l'argument passé à cette fonction (la variable n). Cette construction ressemble fortement à une quantification universelle. En effet, lorsque j'affirme $\forall n, \exists x, x \leqslant n < 2x$, cela signifie que quelque soit le n que l'on me donne, je suis capable de fournir une preuve de $\exists x, x \leqslant n < 2x$. De même, la fonction I, quelque soit le n qu'on lui fournit, retourne un élément habitant le type $\mathcal{M}_n(\mathbb{R})$.

La déduction modulo théorie, quant à elle, consiste à ajouter aux axiomes, la possibilité de définir des règles de calcul, c'est-à-dire des égalités, avec la possibilité de remplacer un membre de l'égalite par l'autre dans la proposition à prouver (ou le type à habiter). Ainsi, si je dispose de la règle de calcul correspondant à l'égalité $2 + 3 = 5$, je peux l'utiliser pour dire que montrer que $[1; 2; 3; 4; 5]$ appartient à `Vec (2+3)`, revient à montrer que $[1; 2; 3; 4; 5]$ appartient à `Vec 5`.

Dans le $\lambda\Pi$-calcul modulo réécriture, les égalités utilisées ne sont pas de n'importe quel type, ce sont des règles de réécriture, ce qui signifie, qu'il s'agit d'égalité orientée. Il y a une notion de membre gauche et de membre droit, et le calcul a une direction, ce qui correspond à ce que l'on fait usuellement : 5 est le résultat de $2 + 3$ ou de $2 \int_2^3 x \, \mathrm{d}x$ et non l'inverse.

1.2.4 Dedukti est un langage de programmation

Cette opportunité offerte à l'utilisateur de déclarer les règles de réécriture de son choix, lui permet de demander à l'ordinateur de faire absolument n'importe quel calcul (effectuable par une machine de Turing) entre deux étapes de raisonnement.

Ceci convient peut-être pour le système de types d'un langage de programmation[11], mais ce n'est pas ce que l'on attend d'un vérificateur de preuves. En effet, fournir une preuve d'un résultat, signifie fournir suffisamment d'informations pour que le résultat puisse "facilement" être vérifié. Sinon, dire "c'est vrai" à propos de n'importe quel résultat prouvable, serait considéré comme une preuve valide, puisqu'il est possible pour l'interlocuteur de construire de son côté la preuve du résultat, mais ce n'est pas très intéressant, puisqu'il ne serait pas possible de discriminer "facilement" les preuves justes et celles fausses. La définition de "facilement" est naturellement sujette à discussion, mais dans notre contexte d'informatique, la définition la moins contraignante de "facile" que l'on puisse prendre est "calculable". C'est-à-dire que l'on souhaite un algorithme qui étant donné une preuve nous dit si elle est valide ou non, et ce en temps fini, sans limite de temps autre[12].

Pour avoir cette décidabilité du typage (donc s'assurer que les preuves transportent suffisamment d'informations), il est nécessaire que tous les calculs aboutissent à un unique résultat et qu'effectuer le calcul ne modifie pas le type des objets.

1.3 Contenu de la thèse

Maintenant que l'on a discuté longuement de la raison pour laquelle la logique était considérée comme une discipline de l'informatique, en introduisant la notion de preuves formelles et la correspondance qu'elles ont avec un modèle de calcul, le λ-calcul, puis que nous avons vu quel

énoncé ayant une valeur de vérité), et nous ne chercherons jamais plus avant à distinguer les types qui "sont des propositions" de ceux qui "n'en sont pas".

[11] C'est déjà fort discutable. Existe-t-il des situations où l'on est prêt à passer un temps infini pour vérifier le type d'un programme, qui lui s'exécute pourtant très rapidement ?

[12] Certains sont plus exigeants et considèrent que le temps de vérification d'une preuve doit être polynomial, voire linéaire en sa taille.

était le vérificateur de preuves étudié dans cette thèse, il est temps de discuter plus en détails de son contenu et des résultats qu'elle contient.

1.3.1 Prémices sur le λ-calcul et la réécriture en théorie des types

Dans le chapitre 3, le λ-calcul pur est présenté en détails, non pas parce qu'une introduction supplémentaire était requise, il est existe déjà de multiples très bien faites, mais surtout pour expliciter la syntaxe choisie. En effet, il en existe plusieurs variantes, qui diffèrent par la priorité et le parenthésage des opérations, ce qui risquerait d'introduire une ambiguïté dans la suite du texte. Ce chapitre est aussi l'occasion d'expliciter certaines notations liées à la substitution et aux variables liées et libres et de définir l'ensemble des termes comme étant un quotient relativement à l'α-équivalence.

Ensuite, ce chapitre introduit les systèmes de types purs (PTS) et présente plusieurs variantes équivalentes des règles de typage, avec des preuves que ces différentes variantes permettent de typer les mêmes termes. Ce typage du λ-calcul dans les PTS est à la base du système de preuves utilisé dans cette thèse, et les diverses présentations équivalentes sont utilisées ultérieurement, pour permettre de faire des inductions sur l'arbre de preuves de la façon la plus pratique en fonction du résultat démontré.

Ce chapitre est suivi d'un court chapitre (le Chapter 4) qui définit précisément la forme des règles de réécriture qui seront utilisées, puis explique comment celles-ci sont intégrées à la règle de conversion pour enrichir le système de typage.

La notion de règles de réécriture d'ordre supérieur ne désigne pas la même chose pour tout le monde, et il était important, même si elles ne sont pas originales, de détailler les définitions que nous employons afin d'éviter toute ambiguïté.

1.3.2 Le $\lambda\Pi$-calcul modulo réécriture

Après deux chapitres très généraux, le chapitre 5 est consacré au $\lambda\Pi$-calcul modulo réécriture plus spécifiquement. Il s'agit certes d'un cas particulier de "Rewriting Type System", qui a été défini au chapitre précédent, cela n'exclut pas certaines spécificités qui sont justement discutées dans ce chapitre.

Tout d'abord, ce chapitre explicite un certain nombre de propriétés du calcul qui sont abondamment utilisées dans les chapitres suivants. La suite de ce chapitre contient une démonstration que la logique à laquelle cette thèse est consacrée est cohérente. Bien que cette preuve ne soit pas particulièrement originale par les techniques utilisées et "fasse partie du folklore", je ne l'ai trouvée dans aucune source antérieure.

Il se termine par une présentation et une discussion de l'encodage introduit par Cousineau et Dowek [CD07] des systèmes de types purs fonctionnels, cet encodage constituant à la fois un premier exemple détaillé de la façon dont on encode une logique dans le $\lambda\Pi$-calcul modulo réécriture, et un fondement sur lequel sont construits les enrichissements présentés dans les trois derniers chapitres de la thèse, qui porte sur la traduction d'un fragment de la logique sous-jacente à l'assistant de preuves Agda vers le $\lambda\Pi$-calcul modulo réécriture.

1.3.3 Terminaison de la réécriture

Le chapitre 6 est le premier a être consacré à un résultat nouveau. Il s'agit d'un critère de terminaison de la réécriture d'ordre supérieur en présence de types dépendants. Comme expliqué précédemment, la terminaison des systèmes de réécritures définis par l'utilisateur dans Dedukti est une propriété importante, puisqu'elle contribue à la décidabilité de la vérification

du typage, condition nécessaire pour que le mot "preuve" recouvre son sens usuel et désigne une suite d'arguments *vérifiables par tous* conduisant logiquement à la conclusion souhaitée.

Le critère présenté (Theorem 6.9.1) utilise une extension de la notion de paires de dépendances, introduite par Arts et Giesl [AG00] au cas de l'ordre supérieur avec types dépendants. Plusieurs travaux antérieurs introduisaient déjà une notion de paires de dépendances pour la réécriture d'ordre supérieur [Bla06, KS07, KvR12, FK19], mais tous ces travaux se restreignent au cas simplement typé.

Pour démontrer que lorsque le système de réécriture vérifie le critère proposé, tous les termes typés sont terminants, j'adapte la technique des candidats de réductibilité[13] de Tait et Girard [Tai67, GLT88]. Il s'agit de donner une interprétation purement syntaxique aux types. Ensuite, deux propriétés sont à démontrer :

- Les interprétations ne contiennent que des termes terminants. Cette propriété découle de la façon dont sont construites les interprétations.
- Tout terme typable est dans l'interprétation de son type. C'est ce résultat qui occupe la majeure partie du chapitre, puisqu'il est obtenu par raffinements successifs du critère.

Pour être très précis, le résultat principal obtenu est le Theorem 6.6.12, qui spécifie que la terminaison d'une relation appelée "relation d'appel" implique la terminaison de la réécriture. Une telle implication pouvant sembler peu utilisable[14], le théorème final (Theorem 6.9.1) est une instanciation avec un critère simple pour prouver la terminaison de la nouvelle relation : le "principe de changement de taille" de Lee, Jones et Ben-Amram [LJBA01], mettant ainsi en lumière l'utilisabilité du théorème 6.6.12.

Le chapitre suivant (le 7) contient une description du prouveur de terminaison SizeChange Tool que j'ai développé et qui utilise une extension du critère présenté précédemment. Ce chapitre contient des exemples d'utilisation du critère, ainsi que des discussions concernant ses forces et faiblesses, ainsi que des pistes d'amélioration de celui-ci.

1.3.4 Encoder une théorie des types riche dans Dedukti

À partir du chapitre 8, on entre dans la dernière partie de la thèse, qui, n'est plus consacrée à l'étude de la réécriture, mais à son utilisation afin d'encoder dans le $\lambda\Pi$-calcul modulo réécriture des fonctionnalités courantes des systèmes de types sous-jacents aux assistants de preuves les plus populaires, et en particulier présents dans le système Agda [NAD+05], dont la traduction vers Dedukti fût la motivation principale de cette partie du travail.

Le chapitre 8 est consacré à une extension de la conversion appelée η-conversion. En théorie des ensembles, une fonction est simplement l'ensemble des couples antécédent/image, ainsi deux fonctions renvoyant le même résultat pour toute entrée sont considérées comme égales, il n'y a pas de référence au processus utilisé pour effectuer le calcul. Cette égalité purement fondée sur les résultats obtenus est appelée "égalité extensionnelle". En théorie des types, l'égalité entre les fonctions est souvent plus liée au processus de calcul, ainsi, une implémentation du tri à bulles et du tri pivot ne sont pas considérées comme égales, puisqu'elles "ne font pas la même chose", et ce bien qu'elles arrivent au même résultat lorsqu'elles sont appliquées à des entrées bien typées. Cette égalité est donc dite "intensionnelle". L'η-conversion est une introduction d'une dose minime d'extensionnalité dans l'égalité intensionnelle, en considérant que la fonction f est égale à la fonction qui à x associe $f(x)$, notée formellement $f =_\eta \lambda x.f\,x$. Ce chapitre explique

[13] Parfois appelés "relations logiques".

[14] Le théorème disant "Pour montrer qu'une relation est bien fondée, il suffit de montrer qu'une autre relation est bien fondée" donne l'impression que l'on s'apprête à tourner en rond.

donc comment encoder cela dans le $\lambda\Pi$-calcul modulo réécriture et contient une preuve de la correction de l'encodage en question, dans le sens où l'encodage préserve bien le typage.

Le chapitre 9 est consacré au polymorphisme d'univers. Le paradoxe de Russell dit que l'existence d'un ensemble de tous les ensembles ne se contenant pas eux-mêmes serait contradictoire. Il n'est donc pas possible de définir $\{x \mid x \notin x\}$ en théorie des ensembles. La solution choisie par les théoriciens des ensembles pour empêcher la déclaration de cet ensemble est de forcer les ensembles définis par compréhension[15] à être des sous-ensembles d'un ensemble déjà défini (compréhension restreinte), et de déclarer qu'il n'existe pas d'ensemble de tous les ensembles[16]. De façon analogue, en théorie des types, il serait contradictoire d'avoir un type de tous les types. Cependant, quantifier sur tous les types permet de déclarer des fonctions polymorphes, ce qui est très pratique pour programmer. La solution qui a alors été trouvée est de rajouter un système de "niveaux" : on ne quantifie pas sur tous les types possibles, mais uniquement sur tous ceux de niveau 0, alors que le type de tous les types de niveau 0, de son côté est de niveau 1, et ainsi de suite. Cependant, de même que l'on trouvait pratique d'avoir du polymorphisme pour ne pas répéter des définitions identiques pour chaque type, on ne souhaite pas répéter des définitions identiques pour chaque niveau. Une notion de polymorphisme d'univers a donc été introduite.

Dans le chapitre 9, j'introduis donc un système de types incluant du polymorphisme d'univers, appelé "polymorphisme d'univers uniforme", qui présente la bonne propriété que toutes les fonctions qui pourraient être appliquées à des niveaux d'univers non-instanciés sont des fonctions totales, garantissant ainsi l'absence d'erreur lors de l'application de ces fonctions sur des niveaux concrets. J'explique ensuite comment encoder cela dans le $\lambda\Pi$-calcul modulo réécriture et prouve que, de nouveau, l'encodage est correct dans le sens où il préserve le typage.

Enfin, l'η-conversion et le polymorpisme d'univers étant présent dans l'assistant de preuves Agda, j'utilise les encodages présentés dans les chapitres précédents dans un outil que j'ai développé lors d'un séjour dans l'équipe de développement d'Agda, à l'université de Chalmers (Göteborg, Suède), et durant lequel Jesper Cockx m'a grandement aidé à appréhender la large base de code, afin de traduire un fragment de la librairie standard d'Agda [DDA20]. Je présente donc dans le chapitre 10 la façon dont sont traduits certains éléments d'Agda, puis discute des résultats obtenus par ce traducteur et des améliorations potentielles de celui-ci pour couvrir une part plus importante de la très riche logique implémentée dans Agda.

[15] De la forme les éléments ayant la propriété P.

[16] Cette proposition n'est pas à proprement parler un axiome de la théorie des types, mais une conséquence directe de l'axiome "de fondation".

Chapter 2

Introduction (in English)

This thesis is a computer science thesis on formal proofs.

2.1 Logic: Mathematics or Computer Science

Before even detailing more precisely what aspects of formal proofs are discussed in this text, the neophyte reader may wonder why the study of formal demonstrations is Computer Science. We must start by observing that, historically, the pioneers of computing, that are Turing, Church and Gödel, raised the question of what is "mechanically computable", even before the existence of computers properly speaking. At its birth, Computer Science was not the science that is interested in how computers work, but the science that was trying to understand what "computable by a mechanical process" means. Multiple models were invented to make a mechanical computation explicit. Let us quote for example, the famous Turing's "machines", or the λ-calculus, introduced by Church.

2.1.1 A Little Detour through Natural Deduction

Until then, it seems quite natural that the question of what is possible, or impossible, to compute with a machine, is part of computer science. That still does not explain why the question of the formalization of proofs is also associated with it. To answer this question, let us start by asking ourselves what mathematical statements mean, and, more specifically, what is the meaning associated with logical connectives: what do "A and B", "A or B", "if A, then B" mean when A and B are themselves statements? A first answer, relatively frequent, because taught in mathematics lessons of secondary school, is to say that a connective is defined by the function which combines the "truth values" of A and B to output the "truth value" of the new statement. These functions are often represented by tables, often called "truth table", like this one:

A	B	A and B	A or B	if A, then B
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	True

However, this definition of the logical connectives is not very satisfactory, since it does not

tell a word of what a proof is. Hence, another approach[1] was adopted by Jaśkowski, who defined a proof system. Independently, Gentzen introduced a few years later a similar system, that he called "Natural Deduction". As indicated by its name, this approach tends to simulate the procedure followed by the mathematician during its reasoning. So, logical connectives are now defined by the way they are handled, by rules named "inference rules". Those handlings are of two kinds:

- one can use an hypothesis which contains the connective one is interested in. The aim is then to break the hypothesis. For this, the "elimination rules" are used;
- one can also target a conclusion which contains the connectives one is interested in. The aim is then to use hypotheses. For this, the "introduction rules" are used.

For instance, if an hypothesis states "A and B", one can then deduce "A". This use of an "and" occurring in an hypothesis is an elimination rule of the conjunction in natural deduction: $\dfrac{A \text{ and } B}{A}$. The rule $\dfrac{A \text{ and } B}{B}$ is the other rule to eliminate a conjunction in natural deduction.

Similarly, if on the one hand I know "A", and on the other hand, I also have the information that "B", I can conclude that "A and B". This reasoning matches the introduction rule of the conjunction in natural deduction: $\dfrac{A \qquad B}{A \text{ and } B}$.

However, the case of implication requires to complexify a little the system. Indeed, to use an implication is relatively simple, with the hypotheses "if A, then B" and "A", one can conclude "B", giving birth to the elimination rule for the implication $\dfrac{\text{if } A\text{, then } B \qquad A}{B}$. But, regarding introduction, the currently presented formalism is limited. Indeed, to prove "if A, then B", one starts by admitting A, and then reasons until reaching B, one then can conclude that "if A, then B". So one has to remember, all along the proof of "B", that "A" was admitted, in order to be able to introduce the implication once the time has come. For this, one uses the symbol $\vdash$, named "turnstile", to separate the admitted propositions and the conclusion. One can then write the introduction rule of the implication: $\dfrac{A \vdash B}{\vdash \text{if } A\text{, then } B}$.

This addition of a context of admitted propositions allows us to introduce the simplest rule, at the top of every proof, which states, without further hypotheses, that "if I admit A, then I have A". Written as an inference rule, it is the rule called "axiom": $\dfrac{}{A \vdash A}$.

But one could reasonably object that this digression on the approaches to define and depict proofs still does not explain what could be the link between a proof, even represented in this formalism, and "mechanically computable" functions. Let us take a step closer to the answer, and note that a proof tree mentions very often the same propositions. Hence, one could try to be more synthetic in its representation, by writing only the sequence of rules used.

Let us try this method on an example[2]:

[1] Naturally, those two approaches are related, and it is for instance possible to construct natural deduction rules from truth tables, as presented in [GH17].

[2] Even if the statements were written in English until now, we will allow ourselves to use the notation $A \wedge B$ to denote "A and B", $A \vee B$ for "A or B" and $A \Rightarrow B$ for "if A, then B", in order to limit the size of the presented proof trees and to simplify the reading of sequences of implications. A sentence like "if (if A and B, then C), then if A, then if B, then C" is peculiarly hard to read, when written this way.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A\wedge B)\Rightarrow C;A;B\vdash (A\wedge B)\Rightarrow C}\,\text{ax} \quad \dfrac{\dfrac{}{(A\wedge B)\Rightarrow C;A;B\vdash A}\,\text{ax} \quad \dfrac{}{(A\wedge B)\Rightarrow C;A;B\vdash B}\,\text{ax}}{(A\wedge B)\Rightarrow C;A;B\vdash A\wedge B}\wedge_I}{(A\wedge B)\Rightarrow C;A;B\vdash C}\Rightarrow_E}{(A\wedge B)\Rightarrow C;A\vdash B\Rightarrow C}\Rightarrow_I}{(A\wedge B)\Rightarrow C\vdash A\Rightarrow B\Rightarrow C}\Rightarrow_I}{\vdash ((A\wedge B)\Rightarrow C)\Rightarrow A\Rightarrow B\Rightarrow C}\Rightarrow_I$$

This proof would then be summed up by the sequence of rules $\Rightarrow_I (\Rightarrow_I (\Rightarrow_I (\Rightarrow_E (ax, (\wedge_I(ax, ax))))))$.

However, a proof of the formula $B \Rightarrow ((A \wedge B) \Rightarrow C) \Rightarrow A \Rightarrow C$ contains exactly the same sequence of rules. One could then expect to have more precise information on how the propositions appearing in the axioms are related to the one introduced by the rule $\Rightarrow_I$.

One could then name the hypotheses, to reference them in the rules, like this[3] :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma\vdash (A\wedge B)\Rightarrow C}\,\text{ax}_x \quad \dfrac{\dfrac{}{\Gamma\vdash A}\,\text{ax}_y \quad \dfrac{}{\Gamma\vdash B}\,\text{ax}_z}{\Gamma\vdash A\wedge B}\wedge_I}{x:(A\wedge B)\Rightarrow C;y:A;z:B\vdash C}\Rightarrow_E}{x:(A\wedge B)\Rightarrow C;y:A\vdash B\Rightarrow C}\Rightarrow_{Iz}}{x:(A\wedge B)\Rightarrow C\vdash A\Rightarrow B\Rightarrow C}\Rightarrow_{Iy}}{\vdash ((A\wedge B)\Rightarrow C)\Rightarrow A\Rightarrow B\Rightarrow C}\Rightarrow_{Ix}$$

The summary of this proof is now $\Rightarrow_{Ix} (\Rightarrow_{Iy} (\Rightarrow_{Iz} (\Rightarrow_E (ax_x, (\wedge_I(ax_y, ax_z))))))$. One must note, that to prove $B \Rightarrow ((A\wedge B) \Rightarrow C) \Rightarrow A \Rightarrow C$, the sequence of rules is $\Rightarrow_{Ix} (\Rightarrow_{Iy} (\Rightarrow_{Iz} (\Rightarrow_E (ax_y, (\wedge_I(ax_z, ax_x))))))$, the indices allow now to distinguish the proofs of those two statements.

2.1.2 λ-calculus and Proofs: Typing

After this very short introduction to proof theory, it is time to note that all the inference rules presented could also but used to type a program. Typing a program comes down to wonder what is the nature of valid inputs and outputs of a program. For instance, the function $f : x \mapsto \ln(x)$ is a function which associates a real number to any strictly positive real number. In mathematics, this is denoted when indicating the domain and range of the function. Here $f : \mathbb{R}_+^* \to \mathbb{R}$.

In most of the cases, the first time the concept of typing is met by students is in physics, with the notion of homogeneity. If m is a mass and ν a cardiac frequency, it does not make sense to wonder what is $m + \nu$, since this quantity does not measure anything[4]. This is because the symbol $+$ is expecting two arguments of the same nature, and output a term of this nature.

Similarly, writing sin(true) does not have any meaning, since the sine function is expecting a number. However, knowing that sine is expecting a number and outputs a number ensures us that since 3 is a number, sin(3) is also a number. More generally, if f is a function from A to B and x belongs to A, then $f(x)$ belongs to B. In the spirit of inference rules introduced in the previous section for formal proofs, on could imagine having inference rules for typing, of the form: $\dfrac{f \in A \to B \qquad x \in A}{f(x) \in B}$.

[3] For size reason, we denote by Γ the context $x : (A \wedge B) \Rightarrow C; y : A; z : B$

[4] It may exist, in the very rich medical or paramedical literature, an indicator of the good (or bad) health of a patient which uses this sum. No matter how accurate this indicator is, this sum does not have any physical reality.

For cartesian product, the same approach can be adopted, and one can note that if a belongs to A and b belongs to B, then the ordered pair (a, b) belongs to $A \times B$, leading to the rule: $\dfrac{a \in A \qquad b \in B}{(a, b) \in A \times B}$.

Here, the attentive reader can notice that the rule to use the functional arrow and the one to eliminate an implication are analogous, as well as the one of introduction of cartesian product and conjunction. This analogy goes further than those two rules, leading to consider propositions of the logic as types of a programming language. So now, we are expecting inference rules to contain not only a context and a proposition, like in the previous section, but also a term, just like for typing.

The rules are now[5] :

$$\frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B} \Rightarrow_e \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \wedge B} \wedge_i$$

For the rules introducing implication and eliminating conjunction, one has to create new symbols, to construct a implication from a variable of the context, and to indicate which proposition of the conjunction to select. Since implication is analogous to functional arrow, one is trying to construct a function. Hence, the notation of the λ-calculus is reused, and one will write $\lambda x.t$ to introduce the hypothesis corresponding to the variable x. On the other hand, since conjunction is a cartesian product, it eliminates with projections, that will be denoted by π_i. Hence those rules are:

$$\frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B} \Rightarrow_i \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_1 t : A} \wedge_{e1} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_2 t : B} \wedge_{e2}$$

2.1.3 Calculus and Proofs: Curry-Howard Correspondence

We have noted that typing a program and proving formally a proposition are two sides of the same construction. However a computer is above all designed to compute. Since the λ-calculus is a model of the behaviour of a computer, it features a computation mechanism, namely the β-reduction, which just states that applying a function to an argument starts by the substitution of a variable of the definition of the function by the argument. For instance, when one defines $f : x \mapsto x^2 + 2 - x$, and then tries to compute $f(5)$, the first step is to replace x by 5 is the "body" of f, producing $f(5) = 5^2 + 2 - 5$.

Let us note that if one writes this down as a tree, the typing of the term to reduce is :

$$\frac{\dfrac{x : A \vdash t : B}{\vdash \lambda x.t : A \Rightarrow B} \Rightarrow_i \qquad \vdash u : B}{\vdash (\lambda x.t)\, u : B} \Rightarrow_e$$

Such a tree is called a "cut" in logic. It states that if a lemma has been proven, it can then be used as an hypothesis in the proofs of the theorems which follow.

Intuitively, one would like to say that cuts are used in proofs only for convenience[6], but that it does not change the set of provable propositions, since one just has to copy the proof of a lemma each time it is used to avoid cuts in a proof. In fact the transformation we just described is stating that each time the hypothesis associated to the variable x is used, it can be replaced by the proof u.

[5] The logical notation $\Rightarrow$ and $\wedge$ are kept rather than $\rightarrow$ et $\times$, which are more traditional in mathematics.

[6] Without lemmas, no mathematical textbook are readable by a human being in reasonable time.

Once again, everything matches perfectly. While explaining how to eliminate the cuts in a proof, we described the substitution associated to the β-reduction of the conclusion of this proof tree.

Hence, the study of formal proofs and of computer programs are both the study of λ-calculus type systems. This identity between type systems (for programs) and deduction formalisms (for proofs) is frequently named the *Curry-Howard correspondence*[7].

2.1.4 Several Logics

Until now, we discussed "the" logic, as if it were a unique and well-identified object. Things are more complicated. First of all, we must note that in the previous section, we only interested ourselves in propositional natural deduction, meaning that there were no quantifiers and all the "atomic" types were simply made of a proposition, as an unspecified, given, string. But, even to express a statement as simple as commutativity of addition, which is a result taught very early in courses of mathematics (at the age of 6 in France), it is already necessary to universally quantify the statements. Commutativity of addition is written $\forall x \in \mathbb{N}, \forall y \in \mathbb{N}, x + y = y + x$. Numerous logics distinguish each other precisely by the situations in which one is allowed to quantify to construct a syntactically valid statement of the logic.

- For instance, higher-order logic allows to quantify on types of functions and on predicates, this is not the case of the first-order logic. For instance, the induction principle on natural numbers is $\forall P, P(0) \Rightarrow (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n, P(n)$. It requires to quantify on the predicate P which depends of a natural number. Hence, one cannot write this formula in first-order logic[8].

- Some logics include polymorphism, which is the ability to quantify universally on the set of types to construct new types. It is especially useful for programming languages, when one does not want to define a type of lists for each types of elements it can contain. For instance, the `map` function has in OCaml the type `('a -> 'b) -> 'a list -> 'b list`, where `'a` and `'b` can be instantiated by any types.

- Some logics allow dependent types. In the statement of the commutativity of the addition, the proposition $x + y = y + x$ depends of variables previously introduced by a quantifier. Since Curry-Howard correspondence tends to erase the distinction between propositions and datatypes, one can also have type which depends of variables introduced by previous quantifications. This allows, for instance, to declare a type of vectors of size n, with n a natural number.

Naturally, some logics combine several of those features and some differences between logics do not reduce to the ability to quantify or not on a certain type at a certain position. We will precisely discuss one of those enrichment in the next section, which will be about Dedukti.

2.1.5 True or Provable

Before this, we must eliminate a temptation that the reader might have, which is to imagine that to create the "strongest" logic, one just has to aggregate all features in one logic.

[7] Some people add the names of Lambek or de Bruijn.

[8] Peano's arithmetic avoids this difficulty by transforming the induction principle into an axiom scheme, meaning that it states "there is an infinity of axioms of the shape $P(0) \Rightarrow (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n, P(n)$, one for each possible P".

In a sense, the reader who makes this suggestion is right, such a logic would allow to prove a lot of things. A little too much. In fact, this logic would be "inconsistent", meaning that it would allow to prove anything, even what "is false".

To be precise, there are now two notions which cohabit:

- the syntax, which nominates the proof system and is associated to the notion of "provable" proposition,
- the semantic, which nominates the concrete models of the axioms and is associated to the notion of "valid" proposition.

A logic is said "consistent" if the proof system is "correct", meaning that it has non-degenerated models and only allows to prove propositions which are true in every model.

Inconsistent type systems are very interesting, especially in programming. However, for a proof assistant, aiming at increasing the trust that one has in the validity of mathematical results, they are clearly less desirable. If Thomas Hales had explained that he obtained a proof of Kepler's conjecture [HAB+15, Hal05] in a system in which it is possible to prove anything, no matter how correct it is, then it is a safe bet to say that the mathematical community would not have had any interest in his result.

2.2 The Dedukti Project

Now that elements of proof theory have been stated, in order to explain the link between computation and proofs and why the study of logic is considered as part of computer science, it is time to detail more the context in which this thesis has been done.

The team in which I worked develops Dedukti, a proof checker for a logic called the $\lambda\Pi$-calculus modulo rewriting.

2.2.1 Why the $\lambda\Pi$-Calculus Modulo Rewriting?

After mentioning several variants of logics, the reader might be relieved to know that we will now focus on the study of one of them.

But the global picture is a bit more complex. Indeed, as there are several logics, there are several proof checkers, each implementing a different logic. But this implies that every result, no matter how universal it is, must be proved in all the systems independently. This situation is frustrating, and one of the aim of the team is to construct Logipedia [Log], an encyclopedia of proofs usable in multiple proof checkers, and this regardless of the system in which the proof was originally produced.

To achieve this, all the proofs are first translated to a unique logical framework, the $\lambda\Pi$-calculus modulo rewriting and its implementation Dedukti.

2.2.2 On Logical Frameworks

Here a question raises naturally. I explained in Section 2.1.5, that it is not possible to have a consistent logic stronger than any other. And now, I am claiming that one will encode all proofs into the $\lambda\Pi$-calculus modulo rewriting, regardless of the logic in which they were demonstrated first.

The natural conclusion that could be drawn from this is that the $\lambda\Pi$-calculus modulo rewriting is inconsistent. Happily, it is not always the case. The subtlety relies on the ability to encode

other logics in the λΠ-calculus modulo rewriting, transforming then a proof of theorem T in logic $\mathcal{L}$ in a proof in λΠ modulo rewriting of the proposition "T is provable in $\mathcal{L}$".

Logics well-suited for encoding of other logics are called "logical frameworks". The first logical system introduced in order to define several theories in it is the "predicate logic" [HA28], in which one can define euclidean geometry (Hilbert's axioms), arithmetic (Peano's axioms) or set theory (Zermelo-Fraenkel theory).

Even if having a unique framework, with unified definitions of connectives ($\vee$, $\wedge$, $\Rightarrow\dots$), of quantifiers ($\forall$, $\exists\dots$) and of the notions of models and proofs, is a major breakthrough, allowing to have general theorems of correctness and completeness, predicate logic suffers some defaults, encouraging to try to encode proofs in another logical framework.

First of all, in predicate logic, it is not possible to bind a variable, other than by using the quantifiers $\exists$ and $\forall$. For instance, it is not possible to define a symbol $\mapsto$ which would bind a variable in its argument. To overcome this issue, several logical frameworks were introduced, like λ-Prolog [Mil91] or the Edinburgh Logical Framework [HHP93], often abbreviated as LF, which is also called the λΠ-calculus.

Another major drawback of predicate logic is that there are no distinction between the computation steps and the reasoning steps. For instance, in Peano's arithmetic, getting the equality $2+3=5$ requires numerous proof steps, whereas a computation allows to get the result. Deduction modulo theory [DHK03] allows to introduce computation steps in proofs.

2.2.3 What is the λΠ-Calculus Modulo Rewriting?

The reader may have already guessed that λΠ-calculus modulo rewriting is the combination of the Edinburgh Logical Framework and of deduction modulo rewriting.

The Edinburgh Logical Framework, or λΠ-calculus, is a quite simple system. It is a λ-calculus which features only one of the enrichments introduced Section 2.1.4, dependent types.

As indicated previously, the mechanism of dependent types allows to bind a variable in a type, to create, not a proposition, but a new type[9]. In mathematics, this construction is often used, for instance to define the function $I : n \in \mathbb{N} \to \mathcal{M}_n(\mathbb{R})$ which associates to n the identity matrix of size n, it is necessary to have the codomain (square matrices of size n) to depend of the argument given to the function (the variable n). This construction looks strongly like a universal quantification. Indeed, when we state that $\forall n, \exists x, x \leqslant n < 2x$, it means that no matter the n we are given, we are able to produce a proof of $\exists x, x \leqslant n < 2x$. Similarly, the function I returns an element of the type $\mathcal{M}_n(\mathbb{R})$ for any n it is given.

On the other hand, deduction modulo theory adds to the axioms the ability to define computation rules, which are equalities with the possibility to replace one side of the equality by the other in the proposition to prove (or the type to inhabit). So if I have the computation rule associated to $2+3=5$, I can use it to state that proving $[1;2;3;4;5]$ belongs to `Vec (2+3)` amounts to prove that $[1;2;3;4;5]$ belongs to `Vec 5`.

In the λΠ-calculus modulo rewriting, the used equalities are not of any kind, they are rewriting rules, meaning that they have a favorite direction. There is a notion of left- and right-hand side, and the computation has a direction, as usually: 5 is the result of 2+3 or of $2\int_2^3 x\,\mathrm{d}x$ and not the opposite.

[9] Even if Curry-Howard correspondence tends to identify types and propositions, it would be awkward to state that `Vec n` or $\mathcal{M}_n(\mathbb{R})$ are valid, to say that those types are not empty. We will only consider this intuitive notion of proposition (essentially a type which corresponds to a statement having a truth value), and we will not in this work go further in distinguishing the types which "are propositions" and the ones which "are not".

2.2.4 Dedukti is a Programming Language

This ability given to the user to declare the rewriting rules she wants, allows her to ask the computer to do absolutely any computation (performable by a Turing machine) between two steps of reasoning.

This could be acceptable for the type system of a programming language[10], but it is not what is expected of a proof checker. Indeed, give a proof of a result, means provide enough information for the result to be "easily" checked. Otherwise, simply stating "this is true" would be an acceptable proof of any provable statement, since it is theoretically possible for the interlocutor to build herself independently a proof of the result. But such a proof is not very interesting, since it would not be possible to "easily" discriminate between right and wrong proofs. The meaning of "easily" is naturally debatable, but in our context of Computer Science, the less restrictive definition of "easy" one could take is "computable". This means that we want an algorithm telling us if a proof is correct or not in finite time, without other restrictions[11].

To have decidable typing (hence ensure that the proofs carry enough information), it is necessary that all computations end with a result and that computing does not modify the type of the objects.

2.3 Content of the Thesis

Now that the reason why logic is considered as a computer science topic has been discussed at length, introducing the notion of formal proofs and the correspondence they have with a computation model, the λ-calculus, and that the proof checker studied and its underlying logic have been presented, it is time to discuss more specifically of the content of this thesis and the results it contains.

2.3.1 Premises on the λ-calculus and on Rewriting in Type Theory

In Chapter 3, the pure λ-calculus is presented in details, not because a new introduction to it was required, there are already several very good presentations of it, but mainly to make explicit the chosen syntax. Indeed, there are several variants, differing mainly by the priority of the operations and the parenthesising, and this might have introduce some ambiguities in the rest of the text. This chapter also gives the opportunity to explicit some notations related to substitution and bound or free variables and to define the set of terms as a quotient relatively to α-equivalence.

Then, this chapter introduces the Pure Type Systems (PTS) and present several equivalent variants of their typing rules, with proofs that those variants all type the same terms. This typing of λ-calculus in PTS's is the basis of the type system used throughout this thesis, and the various equivalent presentations are used to do inductions on the proof tree in the most practical way depending on the demonstrated result.

This chapter is followed by a short one (Chapter 4) which defines precisely the shape of the rewriting rules we will use, and explain how they will be integrated in the conversion rule to enrich the type system.

The notion of higher-order rewriting rule does not designate the same thing for everyone, and it is important, even if they are not original, to detail all the definitions we are using, to avoid

[10] This assertion is already highly debatable. Is there any situation where one is ready to wait an unbounded time to type check a program, which executes fastly?

[11] Some people are more demanding and consider that the checking time of a proof must be polynomial, or even linear in the size of the proof certificate.

any ambiguity.

2.3.2 λΠ-Calculus Modulo Rewriting

After two very general chapters, Chapter 5 is dedicated to the λΠ-calculus modulo rewriting. Even if it a special case of "Rewriting Type Systems" defined in the previous chapter, it has several specificities that deserve to be discussed.

The beginning of the chapter explicits various properties of this system which are widely used in the following chapters. It then contains a proof of the consistency of the logic this thesis is dedicated to. Although this proof is not particularly original by the techniques used, and is probably "folklore", I have not found it in any previous source.

It ends with a presentation and discussion of the encoding of functional Pure Type Systems dues to Cousineau and Dowek [CD07]. This encoding is both a first detailed example of how to encode a logic in the λΠ-calculus modulo rewriting and a foundation on which will be built the enrichments presented in the last three chapters of the thesis, which concerns the translation of a fragment of the logic underlying the proof assistant Agda to the λΠ-calculus modulo rewriting.

2.3.3 Termination of Rewriting

Chapter 6 is the first one dedicated to a new result. It introduces a termination criterion for higher-order rewriting with dependent types. As explained earlier, termination of rewriting systems declared by the user in Dedukti, combined with β, is a crucial property, since it participates to the decidability of type checking, a necessary condition for the word "proof" to recover its usual meaning and designate a sequence of *checkable by anyone* arguments leading logically to the expected conclusion.

The presented criterion (Theorem 6.9.1) uses an extension of the notion of dependency pairs, introduced by Arts and Giesl [AG00], to the higher-order case with dependent types.

To prove that whenever a rewriting system verifies the criterion, all the well-typed terms are terminating, I adapted the reducibility candidates[12] techniques of Tait and Girard [Tai67, GLT88]. They are a syntactical interpretation of types. Then, two properties are to prove:

- The interpretations only contain terminating terms. This property is a direct consequence of the way the interpretations are defined.
- Every typable term is in the interpretation of its type. This result takes the main part of the chapter, since it is obtained by successive refinements of the criterion.

To be very precise, the main result is Theorem 6.6.12, which states that the termination of a relation called "call relation" implies the termination of rewriting. Such an implication might seem unusable[13], the final theorem (Theorem 6.9.1) is an instance of it with a simple criterion to prove the well-foundedness of the new relation: the "Size-Change Principle" of Lee, Jones and Ben-Amram [LJBA01], highlighting the usability of Theorem 6.6.12.

The next chapter (Chapter 7) contains a description of the termination prover SizeChange Tool, that I developed and which uses an extension of the previously presented criterion. This chapter contains examples of use of the criterion and discussions regarding its strengths, weaknesses, and potential enhancements.

[12] Also called "logical relations".

[13] The theorem stating "To show that a relation is well-founded, one just has to show that another relation is well-founded" gives the impression that we are about to go in circles.

2.3.4 Encoding a Rich Type Theory in Dedukti

From chapter 8, one enters the last part of this thesis, which is not anymore dedicated to the study of rewriting, but to its use in order to encode frequent features of the type theories underlying proof assistants in the $\lambda\Pi$-calculus modulo rewriting. More precisely, one will be interested in features offered by the system Agda [NAD+05], whose translation to Dedukti was the main motivation of this work.

Chapter 8 is devoted to an extension of conversion called η-conversion. In set theory, a function is just a set of preimage/image ordered pairs, hence two functions outputting the same result for every input are considered equal. There are no references to the computation process. This equality purely based on the result of the computation is called "extensional equality". In type theory, equality between functions is more often related to the computation process, and an implementation of bubble sort and one of quick sort are not considered equal, since they do not "do the same computations", although they reach the same result when applied to well-typed arguments. An equality discriminating them is said "intentional". η-conversion is an introduction of a minimal amount of extensionality in the intentional equality, by stating that the function f is equal to the function which associates to x the result $f(x)$, formally stated $f =_\eta \lambda x.f\,x$. So this chapter explains how to encode this feature in the $\lambda\Pi$-calculus modulo rewriting and provides a proof of the soundness of the proposed encoding, in the sense that this encoding preserves well-typedness.

Chapter 9 is dedicated to universe polymorphism. Russell's paradox states that the existence of a set of all sets which does not belong to themselves would be inconsistent. Hence, it is not possible to define $\{x \mid x \notin x\}$ in set theory. The solution chosen by set theoreticians to prevent the declaration of this set is to enforce sets defined by comprehension[14] to be subsets of previously defined sets (restricted comprehension), and to declare that there are no sets of all sets[15]. Analogously, in type theory, it would be inconsistent to have a type of all types. However, quantifying on all types allows to declare polymorphic functions, a very useful feature for programming. The solution found was to add a system of "levels": one does not quantify on all possible types, but only on those of level 0, whereas the type of all the types of level 0 is of level 1, and so on. However, just like we liked to have polymorphism to avoid duplicating identical definitions, one for each type, we do not want to duplicate identical definitions for each level. Hence a notion of universe polymorphism was introduced.

In chapter 9, I introduce a type system featuring universe polymorphism, called "uniform universe polymorphism", which offers the property that every function which can be applied to non-instantiated universe levels are total, preventing errors occurring when those functions are applied to concrete instances of levels. I then explain how this system can be encoded in the $\lambda\Pi$-calculus modulo rewriting and provide again a proof that the encoding is sound, in the sense that it preserves well-typedness.

Finally, since η-conversion and universe polymorphism are featured by the proof assistant Agda, I use the encodings presented in the previous chapters in a tool I developed in the Agda's development team, in Chalmers University (Gothenburg, Sweden), where Jesper Cockx greatly helped me to understand the big code base, in order to translate a part of the Agda standard library [DDA20]. In chapter 10, I explain how are translated some Agda terms, and then discuss the results obtained by the translator and possible enhancements of it to cover a wider fraction of the very rich logic implemented by Agda.

[14] Of the shape "elements verifying property P".

[15] This proposition is not an axiom of Set Theory, but a direct consequence of the axiom of "foundation".

Chapter 3

Pure and Typed λ-Calculus

The λ-calculus is a formal system introduced by Church [Chu40]. It was thought as a formalism to define and characterise recursive functions.

The only ingredients of the λ-calculus are functions and applications. Despite the apparent simplicity of a language with so few features, the study of the λ-calculus is the subject of an extensive literature, both the pure λ-calculus seen as a computation model, and the typed ones, which are viewed as the foundations of all functional programming languages, and especially the ones of proof assistants.

Even if the presentation I have adopted is not always standard, this chapter does not contain original results. However, it is required to devote a chapter to this subject, since the numerous books, articles, surveys and course notes available on the subject do not agree on the syntax employed for the λ-calculus and especially the parenthesising conventions, and this chapter should remove all ambiguities on the way to parse terms.

All the results presented in this chapter can be found in any good reference on the λ-calculus, like the book of Krivine [Kri93] or the course notes of Selinger [Sel08]. If the reader is especially interested in the pure λ-calculus, all the results proved in the first sections of this chapter, and much more, can be found in the comprehensive [Bar81]. The same author presents the Pure Type Systems in another reference [Bar92]. Finally, since this thesis advocates for the use of formal provers, it must be noted that Barras proved numerous properties on the Pure Type Systems in the proof Assistant Coq [Bar07], as described in his PhD thesis [Bar99].

3.1 Syntax of λ-Calculus

Even if one could imagine avoiding them [Sch24, Cur30], our language will contain variables.

In order to declare them, let $\mathcal{X}$ be a denumerable set of names. In all this work, we will assume that $\mathcal{X}$ does not interact with the rest of the syntax, meaning that a variable name cannot be confused with a more complex term. For instance $x\,x$ is not a valid name, nor are $\lambda\,(x : A)\,.x$ and $\vdash_P x : A$.

An easy way to ensure this property, could be to define the set of names to be any finite non-empty sequence of latin letters, greek letters and arabic numerals which does not start by a λ. However, since properties, in general, do not depend on the names we have chosen, we will allow ourselves to change the set $\mathcal{X}$, according to our needs. Especially, in some definitions, $\bot$, $+$ or $\times$ will be perfectly valid names, whereas in some others, those symbols will not be valid names.

To declare the identity function on natural numbers, rather than writing $f = x \mapsto x$, like in most mathematical branches, one denotes it by $f = \lambda\,(x : \mathbb{N})\,.x$.

Definition 3.1.1 (Abstract terms of λ-calculus)**.** *The* abstract terms *are defined by the grammar T, with:*

$$T ::= \mathcal{X} \;\mid\; \lambda(\mathcal{X}, T, T) \;\mid\; \Pi(\mathcal{X}, T, T) \;\mid\; @(T, T)$$

Even if, in this section, we are only considering the syntax, let us say a word about the purpose of those different constructions. The λ and the @ are used in terms to denote the definition of functions and the application respectively, whereas Π is used to declare a type of functions.

This definition of terms is very close to the way one could declare such an inductive type in a functional programming language, however, it is not lightweight at all and is not really human-readable. Hence, we will provide a new definition for terms, much more readable.

Definition 3.1.2 (Named terms of λ-calculus)**.** *The* terms *are defined by the grammar Λ, with:*

$$\begin{aligned} \Lambda &::= R \;\mid\; L^+\,R \\ L &::= \mathcal{X} \;\mid\; (\Lambda) \\ R &::= L \;\mid\; \lambda\,(\mathcal{X} : \Lambda)\,.\Lambda \;\mid\; (\mathcal{X} : \Lambda) \Rightarrow \Lambda \end{aligned}$$

The purpose of L and R is simply to declare our parenthesising conventions. Since several such conventions exist, let us say a word about the syntax we have chosen. Two general principles lead the reading of a term respecting our convention: binders have maximal range and application is left associative. Maximality of the range of binders means that $(\lambda\,(x : A)\,.t\,u)$ is interpreted as $\lambda(x, A, @(t, u))$. Left-associativity of the application means that $(t)\,(u)\,v$ is interpreted as $@(@(t, u), v)$.

One can note here that the application, is not denoted by @ anymore, but simply by the concatenation of terms, and the dependent arrow uses the symbol $\Rightarrow$ rather than the Π. The notation $\Pi(x : A).B$ is sometimes used to denote the dependent arrow, which is also called *product*, we will not use this notation, but write it $(x : A) \Rightarrow B$.

Proposition 3.1.3 (Non-ambiguity)**.** *The grammar is not ambiguous.*

Indeed, this grammar is SLR(1) [ALSU86].

Definition 3.1.4 (Parsing function)**.** *We define the function* parse : $(M \in \{\Lambda, L, R, L^+\}) \to M \to T$*, by:*

$$\begin{aligned}
\mathrm{parse}(\Lambda, r) &= \mathrm{parse}(R, r) && \text{if } r \in R \\
\mathrm{parse}(\Lambda, lr) &= @(\mathrm{parse}(L^+, l), \mathrm{parse}(R, r)) && \text{if } r \in R \text{ and } l \in L^* \setminus \{\varepsilon\} \\
\mathrm{parse}(L, x) &= x && \text{if } x \in \mathcal{X} \\
\mathrm{parse}(L, (l)) &= \mathrm{parse}(\Lambda, l) && \\
\mathrm{parse}(R, l) &= \mathrm{parse}(L, l) && \text{if } l \in L \\
\mathrm{parse}(R, \lambda\,(x : A)\,.t) &= \lambda(x, \mathrm{parse}(\Lambda, A), \mathrm{parse}(\Lambda, t)) && \\
\mathrm{parse}(R, (x : A) \Rightarrow t) &= \Pi(x, \mathrm{parse}(\Lambda, A), \mathrm{parse}(\Lambda, t)) && \\
\mathrm{parse}(L^+, l) &= \mathrm{parse}(L, l) && \text{if } l \in L \\
\mathrm{parse}(L^+, ml) &= @(\mathrm{parse}(L^+, m), \mathrm{parse}(L, l)) && \text{if } m \in L^+ \text{ and } l \in L
\end{aligned}$$

Proposition 3.1.5 (Surjectivity)**.** parse(Λ) *is surjective and* print : $T \to \Lambda$ *defined by:*

$$\begin{aligned} \text{print}(x) &= x && \text{if } x \in \mathcal{X} \\ \text{print}(\lambda(x, A, t)) &= \lambda(x : \text{print}(A)).\text{print}(t) \\ \text{print}(\Pi(x, A, B)) &= (x : \text{print}(A)) \Rightarrow \text{print}(B) \\ \text{print}(@(t, u)) &= (\text{print}(t))\ (\text{print}(u)) \end{aligned}$$

is one of its right-inverse.

Proof. By induction on $t \in T$, we show that $\text{parse}(\Lambda, \text{print}(t)) = t$.

($\mathcal{X}$) Let $x \in \mathcal{X}$. $\text{parse}(\Lambda, \text{print}(x)) = \text{parse}(\Lambda, x) = \text{parse}(R, x) = \text{parse}(L, x) = x$.

(λ) Let $x \in \mathcal{X}$ and a, b in T. By induction hypothesis, we consider a, b such that $\text{parse}(\Lambda, \text{print}(a)) = a$ and $\text{parse}(\Lambda, \text{print}(b)) = b$. Then

$$\begin{aligned} \text{parse}(\Lambda, \text{print}(\lambda(x, a, b))) &= \text{parse}(\Lambda, \lambda(x : \text{print}(a)).\text{print}(b)) \\ &= \text{parse}(R, \lambda(x : \text{print}(a)).\text{print}(b)) \\ &= \lambda(x, \text{parse}(\Lambda, \text{print}(a)), \text{parse}(\Lambda, \text{print}(b))) \\ &= \lambda(x, a, b) \text{ by induction hypothesis.} \end{aligned}$$

(Π) This case is analogous to the one of (λ).

(@) Let $a, b \in T$. Then

$$\begin{aligned} \text{parse}(\Lambda, \text{print}(@(a, b))) &= \text{parse}(\Lambda, (\text{print}(a))\ (\text{print}(b))) \\ &= @(\text{parse}(L^+, (\text{print}(a))), \text{parse}(R, (\text{print}(b)))) \\ &= @(\text{parse}(L, (\text{print}(a))), \text{parse}(L, (\text{print}(b)))) \\ &= @(\text{parse}(\Lambda, \text{print}(a)), \text{parse}(\Lambda, \text{print}(b))) \\ &= @(a, b) \text{ by induction hypothesis.} \end{aligned}$$

□

Until this point, we have been persnickety regarding the syntax of λ-calculus and especially bracketing, prompting us to distinguish between abstract and named terms. This (over-)rigorous approach to syntax was made mandatory by the diversity of conventions prevailing in the community. For instance, some authors do not have a usage of parentheses as liberal as ours, enforcing applications to be between parentheses, whereas some others reserve them to the left-most applicand. Those two conventions lead to different interpretations of the examples given after Definition 3.1.2.

Convention 3.1.6. *From this point, we will allow ourselves to confuse the named terms and the abstract syntax. For instance, "by induction on Λ" should be read as "by induction on T" and t u does not designate the concatenation of terms but any term whose parsing leads to* $@(t, u)$.

Definition 3.1.7 (Size of a λ-term)**.** *The function* size : $\Lambda \to \mathbb{N}$ *is defined by:*

$$\begin{aligned} \text{size}(x) &= 1 && \text{if } x \in \mathcal{X} \\ \text{size}(\lambda\,(x : A)\,.t) &= \text{size}(A) + \text{size}(t) + 1 \\ \text{size}((x : A) \Rightarrow B) &= \text{size}(A) + \text{size}(B) + 1 \\ \text{size}(t\,u) &= \text{size}(t) + \text{size}(u) + 1 \end{aligned}$$

This is a typical example of Convention 3.1.6, since in practice our definition of size is by case analysis of the four constructions of T. Especially, $t\,u$ designates the application of t to u, no matter what the syntactic concatenation of t and u is.

3.2 Irrelevance of Names

3.2.1 Free and Bound Variables

In mathematics, the variable x is purely local in the expressions $\sum_{x=0}^{n} \frac{1}{x!}$, $\int_0^1 \sin(nx)\mathrm{d}x$ and $\exists x \in \mathbb{N}.x < n$. The variable x is then called *bound* and the operator which binds it is called a *binder*.

Unlike x, n which also appears in all those expressions occurs only once per expression. It is an indication that n is not used as a shortcut to say "the variable I introduced earlier in this term", but is more global. It is what we call a *free variable*. One has to be cautious, occurring only once is not what defines a *free variable*, it happens that free variables occur several times in the same term.

In λ-calculus, there are two constructions which bind variables: $\lambda\,(x : A)\,.t$ binds x in t and $(x : A) \Rightarrow B$ binds x in B. One must note that in both cases, x is not bound in A.

More formally:

Definition 3.2.1 (Free variables)**.** *The function* $\mathrm{FV} : \Lambda \to \mathcal{P}(\mathcal{X})$ *is defined by:*

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} && \text{if } x \in \mathcal{X} \\ \mathrm{FV}(\lambda\,(x : A)\,.t) &= \mathrm{FV}(A) \cup (\mathrm{FV}(t) \setminus \{x\}) \\ \mathrm{FV}((x : A) \Rightarrow B) &= \mathrm{FV}(A) \cup (\mathrm{FV}(B) \setminus \{x\}) \\ \mathrm{FV}(t\,u) &= \mathrm{FV}(t) \cup \mathrm{FV}(u) \end{aligned}$$

The function FV outputs a set of names, for practical reasons. However, each occurrence of a variable is either bound or free. Indeed, if the same name occurs several times in a term, some occurrences can be bound, whereas the others are free. For instance in $x\ (\lambda\,(x : A)\,.x)$, the first occurrence of x is free, whereas the third is bound by the second occurrence of x, the one in $\lambda(x : A)$.

One must also note that being free or bound depends on the context, and is not stable by the sub-term operation. Indeed if an occurrence of x is free in t, it is not anymore in $\lambda\,(x : A)\,.t$.

3.2.2 De Bruijn Indices

Convention 3.2.2. *In all this section, we assume that* $\mathbb{N} \cap \mathcal{X} = \emptyset$.

Since λ and $\Rightarrow$ are binders, the name one has chosen for the bound variable "does not exist" outside of the term, hence changing it should not impact the "meaning" of the term. To formalize this idea, De Bruijn introduced a version of the λ-calculus with nameless bound variables. Those are simply replaced by a number stating how many binders must be gone through to find the one binding this variable.

Definition 3.2.3 (λ-terms with De Bruijn indices)**.** *The* terms with De Bruijn indices *are defined by the grammar:*

$$DB ::= x \in \mathcal{X} \ \mid\ n \in \mathbb{N} \ \mid\ \lambda(DB, DB) \ \mid\ @(DB, DB) \ \mid\ \Pi(DB, DB)$$

Here the "abstract" terms with De Bruijn indices are defined. One could naturally define a more usable syntax, closer to Definition 3.1.2. However, since we do not intend to use the λ-calculus with De Bruijn indices to really write terms, we do not introduce such a user syntax for it.

One can spot directly that this defines a nameless λ-calculus, since the main difference with the definition of named λ-calculus (Definition 3.1.2) is the absence of variables in the binders (λ and Π, used respectively to construct functions and function types).

Definition 3.2.4 (De Bruijnizers)**.** *We define the function* $\uparrow\colon \mathcal{X} \to (\mathcal{X} \to (\mathcal{X} \uplus \mathbb{N})) \to \mathcal{X} \to (\mathcal{X} \uplus \mathbb{N})$ *by:*

$$\uparrow(x,\sigma) = \begin{cases} x \mapsto 0 & \\ y \mapsto \sigma(y) & \text{if } y \neq x \text{ and } \sigma(y) \in \mathcal{X} \\ y \mapsto \sigma(y)+1 & \text{if } y \neq x \text{ and } \sigma(y) \in \mathbb{N} \end{cases}$$

We then define $toDB : (\mathcal{X} \to (\mathcal{X} \uplus \mathbb{N})) \to \Lambda \to DB$ *by:*

$$\begin{aligned} toDB(\sigma, x) &= \sigma(x) && \text{if } x \in \mathcal{X} \\ toDB(\sigma, \lambda\,(x : A)\,.t) &= \lambda(toDB(\sigma, A), toDB(\uparrow(x,\sigma), t)) \\ toDB(\sigma, (x : A) \Rightarrow B) &= \Pi(toDB(\sigma, A), toDB(\uparrow(x,\sigma), B)) \\ toDB(\sigma, t\,u) &= @(toDB(\sigma, t)\, toDB(\sigma, u)) \end{aligned}$$

For instance the term $(\lambda\,(x : A)\,.x\ (\lambda\,(y : B)\,.y\,x))\ (\lambda\,(y : C)\,.y\,x)$ is translated (using $toDB(\mathrm{id})$) to:

$$@(\lambda(A, @(0, \lambda(B, @(0,1)))), \lambda(C, @(0,x)))$$

3.2.3 α-equivalence

Definition 3.2.5 (α-equivalent terms)**.** $t, u \in \Lambda$ *are said* α-equivalent *if* $toDB(\mathrm{id}, t) = toDB(\mathrm{id}, u)$.
We denote by $\equiv_\alpha$ *this equivalence relation.*

This might seem to be a pedantic definition, just for saying that one can rename bound variables. However, as one will see in the next section, doing this more explicitly is even heavier.

One can check that, for instance, $\lambda\,(x : A)\,.x \equiv_\alpha \lambda\,(y : A)\,.y$ and $\lambda\,(x : A)\,.y \equiv_\alpha \lambda\,(z : A)\,.y$

3.2.4 Barendregt's Convention

In a class of α-equivalent terms, some have well-separated names. This property of well-separation is called *Barendregt's convention.* Those terms are interesting for practical purposes, since they will allow us to avoid to complexify all the definitions, just to handle terms for which the names were badly chosen.

Definition 3.2.6 (Term in Barendregt's convention)**.** inBC *is a predicate on* Λ *and* inBCwith *one on* $\Lambda \times \mathcal{P}(\mathcal{X})$*, defined by:*

$$\frac{}{x \text{ inBCwith } M}\ x \in M \qquad \frac{B \text{ inBCwith } M \cup \{x\} \quad A \text{ inBCwith } M}{(x : A) \Rightarrow B \text{ inBCwith } M}\ x \notin M$$

$$\frac{t \text{ inBCwith } M \quad u \text{ inBCwith } M}{t\,u \text{ inBCwith } M} \qquad \frac{t \text{ inBCwith } M \cup \{x\} \quad A \text{ inBCwith } M}{\lambda\,(x : A)\,.t \text{ inBCwith } M}\ x \notin M$$

$$\frac{t \text{ inBCwith } \mathrm{FV}(t)}{t \text{ inBC}}$$

t inBCwith M is defined as an intermediary relation to gather in M all the variables which can appear free in t.

Some authors are more restrictive than us and do not allow any two binders to bind the same variable name, even if the scope of those two binders do not overlap. Since being so restrictive is not necessary for our use and make the definition of the relation much heavier, we have chosen this less standard lightweight version of Barendregt's convention.

With our definition, $(\lambda(x : A).x\,x)\ (\lambda(x : A).x\,x)$ is in Barendregt's convention, since the variable x is bound by two distinct abstractions, but they are not one under the other. On the other hand, terms like $\lambda(x : A).\lambda(x : B).x$ and $x\ (\lambda(x : A).x)$ do not respect Barendregt's convention.

Definition 3.2.7 (Barendregtizer)**.** *Since $\mathcal{X}$ is infinite, let us consider an injection $f : \mathbb{N} \to \mathcal{X}$.*

Let us first define $\_\,[\_ \mapsto \_] : (\mathcal{X} \to \mathcal{X}) \to \mathcal{X} \to \mathcal{X} \to \mathcal{X} \to \mathcal{X}$ by

$$\sigma\,[x \mapsto z] = \begin{cases} x \mapsto z \\ y \mapsto \sigma(y) \quad \text{if } y \neq x \end{cases}$$

Then $\mathrm{Baren} : \mathcal{P}_F(\mathcal{X}) \to (\mathcal{X} \to \mathcal{X}) \to \Lambda \to \Lambda$ *is the function defined by:*

$$\begin{aligned}
\mathrm{Baren}(M, \sigma, x) &= \sigma(x) \\
\mathrm{Baren}(M, \sigma, t\,u) &= \mathrm{Baren}(M, \sigma, t)\,\mathrm{Baren}(M, \sigma, u) \\
\mathrm{Baren}(M, \sigma, \lambda(x : A).t) &= \lambda(z : \mathrm{Baren}(M, \sigma, A)).\mathrm{Baren}(M \cup \{z\}, \sigma\,[x \mapsto z], t) \\
\mathrm{Baren}(M, \sigma, (x : A) \Rightarrow t) &= (z : \mathrm{Baren}(M, \sigma, A)) \Rightarrow \mathrm{Baren}(M \cup \{z\}, \sigma\,[x \mapsto z], t)
\end{aligned}$$

where in both cases, $z = f(\min\{j \in \mathbb{N} \mid f(j) \notin M\})$.

In this definition, f is simply introduced to select a new fresh name, denoted by z.

A term like $x\ (\lambda(x : A).x)$ is transformed into $x\ (\lambda(y : A).y)$, the binder of x and its bound occurrence are replaced by a new name, to respect Barendregt's convention. Similarly, $\lambda(x : A).\lambda(x : B).x$ becomes $\lambda(x : A).\lambda(y : B).y$.

Lemma 3.2.8. *For all $t \in \Lambda$, all $M \in \mathcal{P}_F(\mathcal{X})$ and all $\sigma : \mathcal{X} \to \mathcal{X}$, such that $\sigma|_{\mathrm{FV}(t)}$ is injective and $\sigma[\mathrm{FV}(t)] \subseteq M$, one has* $\mathrm{Baren}(M, \sigma, t)$ inBCwith M.

Proof. We prove this by induction on t.

- If $t = x$, $\mathrm{Baren}(M, \sigma, x) = \sigma(x) \in \mathcal{X}$, and all variable names are in Barendregt's convention.
- if $t = u\,v$, $\mathrm{Baren}(M, \sigma, u\,v) = \mathrm{Baren}(M, \sigma, u)\,\mathrm{Baren}(M, \sigma, v)$, and by induction hypothesis $\mathrm{Baren}(M, \sigma, u)$ inBCwith M and $\mathrm{Baren}(M, \sigma, v)$ inBCwith M. One can conclude using Definition 3.2.6.
- If $t = \lambda(x : A).u$, $\mathrm{Baren}(M, \sigma, \lambda(x : A).u) = \lambda(z : \mathrm{Baren}(M, \sigma, A)).\mathrm{Baren}(M \cup \{z\}, \sigma\,[x \mapsto z], u)$, where $z = f(\min\{j \in \mathbb{N} \mid f(j) \notin M\})$.

  By induction hypothesis, $\mathrm{Baren}(M, \sigma, A)$ inBCwith M. We know that $\mathrm{FV}(u) \subseteq \mathrm{FV}(t) \cup \{x\}$. By construction $z \notin M$. Hence, since $\sigma[\mathrm{FV}(t)] \subseteq M$, $\sigma\,[x \mapsto z]\,|_{\mathrm{FV}(t) \cup \{x\}}$ is injective and $\sigma\,[x \mapsto z]\,[\mathrm{FV}(t) \cup \{x\}] = \sigma[\mathrm{FV}(t)] \cup \{z\} \subseteq M \cup \{z\}$. We can apply the induction hypothesis to u and deduce that $\mathrm{Baren}(M \cup \{z\}, \sigma\,[x \mapsto z], u)$ inBCwith $M \cup \{z\}$. One can conclude using Definition 3.2.6.
- If $t = (x : A) \Rightarrow B$, the proof is identical to the one for the abstraction case.

□

Lemma 3.2.9. *For all $t \in \Lambda$, all $M \in \mathcal{P}_F(\mathcal{X})$ and all $\sigma : \mathcal{X} \to (\mathcal{X} \uplus \mathbb{N})$, $\sigma' : \mathcal{X} \to (\mathcal{X} \uplus \mathbb{N})$ and $\tau : \mathcal{X} \to \mathcal{X}$ such that $\tau[\mathrm{FV}(t)] \subseteq M$ and $(\sigma \circ \tau)|_{\mathrm{FV}(t)} = \sigma'|_{\mathrm{FV}(t)}$, one has $toDB(\sigma, \mathrm{Baren}(M, \tau, t)) = toDB(\sigma', t)$.*

Proof. By induction on t.

- If $t = x$, $toDB(\sigma, \mathrm{Baren}(M, \tau, x)) = toDB(\sigma, \tau(x)) = \sigma(\tau(x)) = \sigma'(x) = toDB(\sigma', x)$.
- If $t = u\,v$, then

$$\begin{aligned} toDB(\sigma, \mathrm{Baren}(M, \tau, u\,v)) &= toDB(\sigma, \mathrm{Baren}(M, \tau, u)\,\mathrm{Baren}(M, \tau, v)) \\ &= @(toDB(\sigma, \mathrm{Baren}(M, \tau, u)), toDB(\sigma, \mathrm{Baren}(M, \tau, v))) \\ &= @(toDB(\sigma', u), toDB(\sigma', v)) \qquad \text{by induction hypothesis} \\ &= toDB(\sigma', u\,v) \end{aligned}$$

- If $t = \lambda\,(x : A)\,.u$, then

$$\begin{aligned} toDB(\sigma,&\mathrm{Baren}(M, \tau, \lambda\,(x : A)\,.u)) = toDB(\sigma, \lambda\,(z : \mathrm{Baren}(M, \tau, A))\,.\mathrm{Baren}(M \cup \{z\}, \tau\,[x \mapsto z], u)) \\ &= \lambda(toDB(\sigma, \mathrm{Baren}(M, \tau, A)), toDB(\uparrow(z, \sigma), \mathrm{Baren}(M \cup \{z\}, \tau\,[x \mapsto z], u))) \end{aligned}$$

  and

$$toDB(\sigma', \lambda\,(x : A)\,.u) = \lambda(toDB(\sigma', A), toDB(\uparrow(x, \sigma'), u))$$

  By induction hypothesis, $toDB(\sigma', A) = toDB(\sigma, \mathrm{Baren}(M, \tau, A))$.

  $\mathrm{FV}(u) \subseteq \mathrm{FV}(t) \cup \{x\}$, hence $\tau\,[x \mapsto z]\,[\mathrm{FV}(u)] \subseteq \tau[\mathrm{FV}(t)] \cup \{z\} \subseteq M \cup \{z\}$.

  For x, we have $(\uparrow(z, \sigma) \circ \tau\,[x \mapsto z])(x) = \uparrow(z, \sigma)(z) = 0 = \uparrow(x, \sigma')(x)$. And if y is a free variable of u with $y \neq x$, then $(\uparrow(z, \sigma) \circ \tau\,[x \mapsto z])(y) = \uparrow(z, \sigma)(\tau(y))$. By definition of z, $\tau(y) \neq z$. Hence $\uparrow(z, \sigma)(\tau(y)) = \sigma(\tau(y)) = \sigma'(y) = \uparrow(x, \sigma')(y)$ if $\sigma'(y) = \sigma(\tau(y)) \in \mathcal{X}$ and $\uparrow(z, \sigma)(\tau(y)) = \sigma(\tau(y)) + 1 = \uparrow(x, \sigma')(y)$ if $\sigma'(y) = \sigma(\tau(y)) \in \mathbb{N}$.

  Hence, the induction hypothesis applies, and $toDB(\uparrow(z, \sigma), \mathrm{Baren}(M \cup \{z\}, \tau\,[x \mapsto z], u)) = toDB(\uparrow(x, \sigma'), u)$.

  One can then conclude $toDB(\sigma', \lambda\,(x : A)\,.u) = toDB(\sigma, \mathrm{Baren}(M, \tau, \lambda\,(x : A)\,.u))$.

- The case $t = (x : A) \Rightarrow B$, is identical to the abstraction one. □

With those two lemmas, one can prove that Barendregt's convention is universal in the sense that any term can be put in Barendregt's convention just by changing "local names".

Theorem 3.2.10. *Given any term t and any finite set M such that $\mathrm{FV}(t) \subseteq M$, $\mathrm{Baren}(M, \mathrm{id}, t)$ is a term such that such that $\mathrm{Baren}(M, \mathrm{id}, t)$ inBCwith M and $t \equiv_\alpha \mathrm{Baren}(M, \mathrm{id}, t)$.*

Proof. By the two previous lemmas. □

Convention 3.2.11. *From this point, we consider terms up to α-conversion. This means that Λ will stand for $\Lambda/_{\equiv_\alpha}$. However, most of the definitions given are not independent of the representative of the $\equiv_\alpha$ class, but are independent of the representative in Barendregt's convention. Hence, $t \in \Lambda$ should read "Let $A \in \Lambda/_{\equiv_\alpha}$ and $t \in A$ with t inBC".*

Most (if not all) of the proofs that definitions are independent of the chosen representative in Barendregt's convention are omitted.

3.3 Computation in λ-calculus

3.3.1 Substitutions

Definition 3.3.1 (Domain)**.** *We define* $\mathrm{dom} : (\mathcal{X} \to \Lambda) \to \mathcal{P}(\mathcal{X})$ *by* $\mathrm{dom}(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$.

The attentive reader might be upset by this definition, since x lives in $\mathcal{X}$ and $\sigma(x)$ in Λ. However, by definition of Λ, we have $\mathcal{X} \subset \Lambda$, hence it makes sense to talk about the equality (or disequality) between elements of those sets, by considering that both live in Λ. We have done the same operation of "implicit casting" as when we see -1 as a complex number in the famous identity $e^{i\pi} = -1$, which undoubtedly makes sense, since $\mathbb{Z} \subset \mathbb{C}$.

Definition 3.3.2 (Substitution)**.** *A function* $\sigma : \mathcal{X} \to \Lambda$ *is called a* substitution*. Given such a function, we define its extension as* $\widetilde{\sigma} : \Lambda \to \Lambda$ *by:*

$$\widetilde{\sigma}(x) = \sigma(x) \qquad \text{if } x \in \mathcal{X}$$

$$\widetilde{\sigma}(\lambda\,(x : A)\,.t) = \lambda\,(z : \widetilde{\sigma}(A))\,.\widetilde{\sigma\,[x \mapsto z]}(t) \text{ with } z \in \mathcal{X} \setminus \left(\bigcup_{y \in \mathrm{FV}(t) \setminus \{x\}} \mathrm{FV}(\sigma(y)) \right)$$

$$\widetilde{\sigma}((x : A) \Rightarrow B) = (z : \widetilde{\sigma}(A)) \Rightarrow \widetilde{\sigma\,[x \mapsto z]}(B) \text{ with } z \in \mathcal{X} \setminus \left(\bigcup_{y \in \mathrm{FV}(B) \setminus \{x\}} \mathrm{FV}(\sigma(y)) \right)$$

$$\widetilde{\sigma}(t\,u) = \widetilde{\sigma}(t)\,\widetilde{\sigma}(u)$$

Where $\_\,[\_ \mapsto \_] : (\mathcal{X} \to \Lambda) \to \mathcal{X} \to \Lambda \to \mathcal{X} \to \Lambda$ *is defined, as in Definition 3.2.7, by:*

$$\sigma\,[x \mapsto t]\,(x) = t$$
$$\sigma\,[x \mapsto t]\,(y) = \sigma(y) \qquad \text{if } y \neq x$$

In fact, the outputs of the $\widetilde{\sigma}$'s are not deterministic, since z is not uniquely defined. However, the choice of z is exactly the choice of a representative for the $\equiv_\alpha$ class. It would even have been possible to define $t \equiv_\alpha u$ by "u is a possible choice for $\widetilde{\mathrm{id}}(t)$". This definition presents the advantage of not requiring to define the λ-calculus with De Bruijn indices, but the proof that this relation is really an equivalence relation would have been much more involved, that is why we have chosen the longer but easier-going path of Section 3.2.3. Furthermore, defining $\widetilde{\sigma}$ when some images by σ contain binders, without pre-defining $\equiv_\alpha$, would lead to the definition of a relation which is not independent of the representative.

Notation 3.3.3. *Given* $t \in \Lambda$ *and* σ *a substitution, we denote by* $t\sigma$ *the term* $\widetilde{\sigma}(t)$.

Notation 3.3.4. *Given* $x_1, \ldots, x_n$ *distinct variable names and* $t_1, \ldots, t_n$ *terms,* $\left[t_1/_{x_1}, \ldots, t_n/_{x_n}\right]$ *is the substitution defined by:*

$$\left[t_1/_{x_1}, \ldots, t_n/_{x_n}\right] = \mathrm{Id}\,[x_1 \mapsto t_1] \ldots [x_n \mapsto t_n]$$

Hence

$$\left[t_1/_{x_1}, \ldots, t_n/_{x_n}\right](x_i) = t_i \text{ if } 1 \leqslant i \leqslant n \qquad \left[t_1/_{x_1}, \ldots, t_n/_{x_n}\right](y) = y \text{ if } y \notin \{x_1, \ldots, x_n\}$$

Lemma 3.3.5 (Commutation of substitutions)**.** *For all $x, y \in \mathcal{X}$ with $x \neq y$ and $t, a, b \in \Lambda$, such that $x \notin \mathrm{FV}(b)$, then $(t\,[a/x])\left[b/y\right] = \left(t\left[b/y\right]\right)\left[a\left[b/y\right]/x\right]$.*

Proof. By induction on t.

- If $t = x$, then $(t\,[a/x])\left[b/y\right] = a\left[b/y\right]$ and, since $x \neq y$, $\left(x\left[b/y\right]\right)\left[a\left[b/y\right]/x\right] = x\left[a\left[b/y\right]/x\right] = a\left[b/y\right]$
- If $t = y$, then $(y\,[a/x])\left[b/y\right] = y\left[b/y\right] = b$, since $x \neq y$ and a substitution does not change a term if the domain of the substitution contains no free variables of the term. And $\left(y\left[b/y\right]\right)\left[a\left[b/y\right]/x\right] = b\left[a\left[b/y\right]/x\right] = b$, since $x \notin \mathrm{FV}(b)$.
- If $t = z \in \mathcal{V}$ with $z \notin \{x, y\}$, then $(z\,[a/x])\left[b/y\right] = z\left[b/y\right] = z$ and $\left(z\left[b/y\right]\right)\left[a\left[b/y\right]/x\right] = z\left[a\left[b/y\right]/x\right] = z$.
- If $t = \lambda\,(z : A)\,.u$, by Theorem 3.2.10, one can consider that $z \notin \{x, y\} \cup \mathrm{FV}(a) \cup \mathrm{FV}(b)$.
  Then $((\lambda\,(z : A)\,.u)\,[a/x])\left[b/y\right] = \left(\lambda\,(z : A\,[a/x])\,.\widetilde{[a/x]\,[z \mapsto z]}(u)\right)\left[b/y\right]$, since $z \in \mathcal{X} \setminus \left(\bigcup_{y \in \mathrm{FV}(u)\setminus\{z\}} \mathrm{FV}\left([a/x]\,(y)\right)\right)$.
  One can note that $\widetilde{[a/x]\,[z \mapsto z]}$ is equal to $\widetilde{[a/x]}$, hence $((\lambda\,(z : A)\,.u)\,[a/x])\left[b/y\right] = (\lambda\,(z : A\,[a/x])\,.u\,[a/x])\left[b/y\right] = \lambda\left(z : (A\,[a/x])\left[b/y\right]\right).\widetilde{\left[b/y\right]\,[z \mapsto z]}(u\,[a/x])$, since $z \in \mathcal{X}\setminus\left(\bigcup_{k \in \mathrm{FV}(u)\setminus\{z\}\cup\mathrm{FV}(a)} \mathrm{FV}\left(\left[b/y\right](k)\right)\right)$.
  Finally, $((\lambda\,(z : A)\,.u)\,[a/x])\left[b/y\right] = \lambda\left(z : (A\,[a/x])\left[b/y\right]\right).(u\,[a/x])\left[b/y\right]$.
  For the same reason, $\left((\lambda\,(z : A)\,.u)\left[b/y\right]\right)\left[a\left[b/y\right]/x\right] = \left(\lambda\left(z : A\left[b/y\right]\right).\widetilde{\left[b/y\right]\,[z \mapsto z]}(u)\right)\left[a\left[b/y\right]/x\right] = \left(\lambda\left(z : A\left[b/y\right]\right).u\left[b/y\right]\right)\left[a\left[b/y\right]/x\right] = \lambda\left(z : \left(A\left[b/y\right]\right)\left[a\left[b/y\right]/x\right]\right).\left(u\left[b/y\right]\right)\left[a\left[b/y\right]/x\right]$.
  Applying the induction hypothesis on A and u, one can conclude that $((\lambda\,(x : A)\,.u)\,[a/x])\left[b/y\right] = \left((\lambda\,(x : A)\,.u)\left[b/y\right]\right)\left[a\left[b/y\right]/x\right]$.
- If t is a product, the case is analogous to the one already treated for abstraction.
- If $t = u\,v$, $((u\,v)\,[a/x])\left[b/y\right] = (u\,[a/x])\left[b/y\right]\left((v\,[a/x])\left[b/y\right]\right)$. And $\left((u\,v)\left[b/y\right]\right)\left[a\left[b/y\right]/x\right] = \left(u\left[b/y\right]\right)\left[a\left[b/y\right]/x\right]\left(\left(v\left[b/y\right]\right)\left[a\left[b/y\right]/x\right]\right)$.
  One can apply the induction hypothesis on u and v to conclude.

□

3.3.2 β-reduction

Until this point, it has already been claimed several times, that λ-abstractions are functions and that the whitespace stands for application. However, one only introduced syntactical constructions, whereas applying a function to an argument is expected to trigger some computations. For instance, if one writes $f : x \mapsto x^2$, then it is expected that $f(2)$ computes to 4.

The standard λ-calculus only contains one such rule of computation, which is called the β reduction.

Definition 3.3.6 (Renaming)**.** *A substitution σ is a* renaming *if it is injective and $\sigma[\mathcal{X}] \subseteq \mathcal{X}$.*

Definition 3.3.7 (Name-independent relation)**.** *A relation $R \subseteq \Lambda^2$ is* name-independent *if for all $(t, u) \in R$ and all renamings σ, we have $(t\sigma, u\sigma) \in R$.*

Definition 3.3.8 (Contextual Closure)**.** *Given a name independent relation $R \subseteq \Lambda^2$, the* congruence induced by *R is the smallest relation $\mathcal{C}$ such that:*

$$\mathcal{C} = R \cup \left\{ \begin{array}{ll} (\lambda\,(x : a)\,.t, \lambda\,(x : b)\,.t), & (\lambda\,(x : t)\,.a, \lambda\,(x : t)\,.b), \\ ((x : a) \Rightarrow t, (x : b) \Rightarrow t), & ((x : t) \Rightarrow a, (x : t) \Rightarrow b), \\ (a\,t, b\,t), & (t\,a, t\,b) \end{array} \middle| \; t \in \Lambda, (a, b) \in \mathcal{C} \right\}$$

As stated in Convention 3.2.11, in all the previous definitions, Λ is used to denote $\Lambda/_{\equiv_\alpha}$. Notably, the condition that R is name independent is required for the congruence it induces to be well-defined on $\Lambda/_{\equiv_\alpha}$.

Definition 3.3.9 (Head β-reduction)**.** *We define $\overset{\varepsilon}{\rightsquigarrow}_\beta$ as the name independent relation*

$$\{((\lambda\,(x : A)\,.t)\ u, t\,[u/_x]) \mid x \in \mathcal{X} \text{ and } t, u, A \in \Lambda\}$$

Definition 3.3.10 (β-reduction)**.** *$\rightsquigarrow_\beta$ is the congruence induced by $\overset{\varepsilon}{\rightsquigarrow}_\beta$.*

Example 3.3.11. *We can already note that $\rightsquigarrow_\beta$ is not terminating. Let us consider the term $\Delta = \lambda\,(x : A)\,.x\,x$, then the term $\Delta\,\Delta$ is not terminating since it reduces to itself*

$$\Delta\,\Delta = (\lambda\,(x : A)\,.x\,x)\ \Delta \rightsquigarrow_\beta (x\,x)\,[\Delta/_x] = \Delta\,\Delta.$$

Definition 3.3.12 (β-conversion)**.** *We denote by $\leftrightsquigarrow^*_\beta$ the reflexive, symmetric and transitive closure of $\rightsquigarrow_\beta$.*

Definition 3.3.13 (β-joinability)**.**

$$\downarrow_\beta = \left\{(t, u) \in \Lambda^2 \;\middle|\; \text{there is a } v \in \Lambda, \text{ such that } t \rightsquigarrow^*_\beta v \text{ and } u \rightsquigarrow^*_\beta v\right\}$$

Even if we defined two relations, which look different, since $\leftrightsquigarrow^*_\beta$ is transitive, whereas $\downarrow_\beta$ does not seem to be, actually $\rightsquigarrow_\beta$ is confluent, meaning that $\downarrow_\beta$ is transitive, and so those two relations are identical.

Theorem 3.3.14 (Church-Rosser Property)**.** $\downarrow_\beta = \leftrightsquigarrow^*_\beta$

Even if this chapter tends to reintroduce well-known notions and contains non-original proofs, we will not reprove this result, but we encourage the interested reader to read the original paper proving this result [CR36] or any book on the lambda-calculus. It must be noted however, that most of the books on the λ-calculus, like [Bar81, Kri93], prove this property on a system slightly different of the one introduced here, since the λ-abstractions are not annotated. It is not a problem, since their proof can be straightforwardly adapted to our setting, as shown by the proof that Barras did in Coq of this property [Bar07], which features annotated λ-abstractions.

Proposition 3.3.15 (Stability by substitution)**.** *Given $t, u \in \Lambda$ and σ a substitution, if $t \rightsquigarrow_\beta u$, then $t\sigma \rightsquigarrow_\beta u\sigma$.*

Proof. If $t \overset{\varepsilon}{\rightsquigarrow}_\beta u$, then let x be a variable not in the domain of σ. There are A, v and w such that $t = (\lambda\,(x : A)\,.v)\ w$ and $u = v\,[w/_x]$. Then

$$t\sigma = (\lambda\,(x : A\sigma)\,.v\sigma)\ (w\sigma) \overset{\varepsilon}{\rightsquigarrow}_\beta (v\sigma)\,[w\sigma/_x] = u\sigma.$$

If $t \rightsquigarrow_\beta u$, but not $t \overset{\varepsilon}{\rightsquigarrow}_\beta u$, one just has to do an induction on t to conclude. □

Proposition 3.3.16 (Injectivity of product)**.** *Given $A, A', B, B' \in \Lambda$, if $(x : A) \Rightarrow B \leftrightsquigarrow^*_\beta (x : A') \Rightarrow B'$, then $A \leftrightsquigarrow^*_\beta A'$ and $B \leftrightsquigarrow^*_\beta B'$.*

Proof. Thanks to the Church-Rosser property of Theorem 3.3.14, there is a common reduct to $(x : A) \Rightarrow B$ and $(x : A') \Rightarrow B'$. Since all beta-redices are necessarily in A, A', B and B', and because β-reduction cannot modify the fact that both term are products, one can conclude that $A \leftrightsquigarrow^*_\beta A'$ and $B \leftrightsquigarrow^*_\beta B'$. □

3.4 Typing Rules of Pure Type Systems

3.4.1 Specification and Contexts

Definition 3.4.1 (Pure Type System Specification)**.** *A* Pure Type System *(abbreviated as* PTS*) is specified by a set of names of sorts $\mathcal{S} \subsetneq \mathcal{X}$, such that $\mathcal{X} \setminus \mathcal{S}$ is infinite, a set of "axioms" $\mathcal{A} \subseteq \mathcal{S}^2$ and a set of "rules" $\mathcal{R} \subseteq \mathcal{S}^3$.*

Example 3.4.2 (PTS of the λ-cube)**.** *Among the great variety of PTS with finitely many sorts, those of the λ-cube are the most used. All the members of this family have in common: $\mathcal{S} = \{\star, \Box\}$ and $\mathcal{A} = \{\star : \Box\}$, but they differ by the rules.*

Let us mention two of them: $\lambda^\to$ has $\mathcal{R} = \{(\star, \star, \star)\}$, whereas $\lambda\Pi$ has $\mathcal{R} = \{(\star, \star, \star), (\star, \Box, \Box)\}$.

Example 3.4.3 (Other PTS of interest)**.** *Among PTS with infinitely many sorts, one can cite the predicative and impredicative infinite hierarchies, which are respectively : $\mathcal{P}^\infty$ is $\mathcal{S} = \{*_i \mid i \in \mathbb{N}\}; \mathcal{A} = \{(*_i, *_{i+1})\}; \mathcal{R} = \{(*_i, *_j, *_k) \mid k = \max(i, j)\}$ and $\mathcal{C}^\infty$ is $\mathcal{S} = \{*_i \mid i \in \mathbb{N}\}; \mathcal{A} = \{(*_i, *_{i+1})\}; \mathcal{R} = \{(*_i, *_j, *_k) \mid j \geqslant 1$ and $k = \max(i, j)\} \cup \{(*_i, *_0, *_0)\}$.*

Among the proof assistants, Agda [NAD+05] is predicative and implements an enrichment of $\mathcal{P}^\infty$, whereas Coq [CDT20] is impredicative and implements an enrichment of $\mathcal{C}^\infty$.

Notation 3.4.4 (Set of variable names)**.** *Given a PTS with sorts $\mathcal{S}$, we denote by $\mathcal{V}$ the set of variable names $\mathcal{X} \setminus \mathcal{S}$.*

Definition 3.4.5 (Top sort)**.** *$s \in \mathcal{S}$ is a* top sort *if for all $s' \in \mathcal{S}$, $(s, s') \notin \mathcal{A}$.*

Definition 3.4.6 (Context)**.** *A* context *is a finite list of elements of $\mathcal{V} \times \Lambda$.*

Syntactically, it is the set $\mathfrak{C}$ defined by the grammar:

$$\mathfrak{C} = [\,] \ \mid\ C \qquad C ::= \mathcal{V} : \Lambda \ \mid\ C, \mathcal{V} : \Lambda$$

Convention 3.4.7. *Once again, we have chosen to be very close to the concrete syntax in our definition, and did not require every context to begin with [],. But we will allow ourselves to consider that a context is either [] or a context followed by an ordered pair $x : A$.*

Definition 3.4.8 (Domain of a context)**.** *We define* $\mathrm{dom} : \mathfrak{C} \to \mathcal{P}(\mathcal{V})$ *by:*

$$\mathrm{dom}([\,]) = \emptyset \qquad \mathrm{dom}(\Gamma, x : A) = \mathrm{dom}(\Gamma) \cup \{x\}$$

3.4.2 The Typing Rules

The typing rules include 5 introduction rules related to the syntax, and 2 structural rules.

Definition 3.4.9 (Typing of PTS)**.** *Given a PTS specification $P = (\mathcal{S}; \mathcal{A}; \mathcal{R})$, $\_ \vdash_P \_ : \_$ is the relation included in $\mathfrak{C} \times \Lambda \times \Lambda$ defined by:*

$$\textit{(var)}\quad \frac{\Gamma \vdash_P A : s}{\Gamma, x : A \vdash_P x : A} \begin{cases} x \in \mathcal{V} \setminus \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases} \qquad \textit{(ax)}\quad \frac{}{[\,] \vdash_P s_1 : s_2}\,(s_1, s_2) \in \mathcal{A}$$

$$\textit{(prod)}\quad \frac{\Gamma \vdash_P A : s_1 \quad \Gamma, x : A \vdash_P B : s_2}{\Gamma \vdash_P (x : A) \Rightarrow B : s_3}\,(s_1, s_2, s_3) \in \mathcal{R}$$

$$\textit{(app)}\quad \frac{\Gamma \vdash_P t : (x : A) \Rightarrow B \quad \Gamma \vdash_P u : A}{\Gamma \vdash_P t\,u : B\left[{}^{u}/_{x}\right]} \qquad \textit{(abs)}\quad \frac{\Gamma \vdash_P (x : A) \Rightarrow B : s \quad \Gamma, x : A \vdash_P t : B}{\Gamma \vdash_P \lambda\,(x : A)\,.t : (x : A) \Rightarrow B}$$

$$\textit{(conv)}\quad \frac{\Gamma \vdash_P t : A \quad \Gamma \vdash_P B : s}{\Gamma \vdash_P t : B}\,A \leftrightsquigarrow_\beta^* B \qquad \textit{(weak)}\quad \frac{\Gamma \vdash_P A : s \quad \Gamma \vdash_P t : B}{\Gamma, x : A \vdash_P t : B} \begin{cases} x \in \mathcal{V} \setminus \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases}$$

where $\leftrightsquigarrow_\beta^*$ is the reflexive symmetric transitive closure of $\rightsquigarrow_\beta$.

Here we must note that this definition is not independent of the representative of the $\equiv_\alpha$ class. Indeed, one cannot derive $x : \star \vdash_{\lambda\to} \lambda\,(x : x)\,.x : (y : x) \Rightarrow x$, whereas, $x : \star \vdash_{\lambda\to} \lambda\,(y : x)\,.y : (y : x) \Rightarrow x$ can easily be proved. It is because of this issue, that in Convention 3.2.11 we stated that definitions are not all independent of the $\equiv_\alpha$ representative, but only independent of the representative in Barendregt's convention, and $\lambda\,(x : x)\,.x$ does not respect Barendregt's convention.

Remark 3.4.10 (Irrelevance of Names)**.** *Just like we did for the variables bound by λ and $\Rightarrow$, names of the variables in the context can be modified, provided that one avoids name clashes and "capture".*

We have been very precise regarding renaming for terms, but we will not provide the technical details regarding renaming for typing judgements, which follows the same principles.

One must also note that all variables occurring in a typing judgement (either in the types of the context, in the term or in the type) are declared earlier.

Lemma 3.4.11. *1. If $\Gamma, x : A, \Gamma' \vdash_P t : B$, then $\mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$,*

2. If $\Gamma \vdash_P t : B$, then $\mathrm{FV}(t) \subseteq \mathrm{dom}(\Gamma)$,

3. If $\Gamma \vdash_P t : B$, then $\mathrm{FV}(B) \subseteq \mathrm{dom}(\Gamma)$,

Contexts are presented here as ordered lists of pairs. This ordering is required since every type can refer to the previously introduced variables. However, if the variable does not occur free in the next type of the context, it is possible to exchange the order of those hypotheses.

Lemma 3.4.12 (Exchange)**.** *If $\Gamma, x : A, y : B, \Gamma' \vdash_P t : C$ and $x \notin \mathrm{FV}(B)$, then $\Gamma, y : B, x : A, \Gamma' \vdash_P t : C$.*

Those results are obtained by a direct induction on the proof tree.

3.4.3 Inversion Theorems

Since all the constructions of terms are associated to one introduction rule, one could expect, by looking to the shape of the term, to know what was the last typing rule used in the derivation. But the two rules (*weak*) and (*conv*) are "silent", in the sense that their application is not reflected by a modification of the term. All the following theorems state that any judgement has a proof which ends by the introduction rule associated to the shape of the term to type, followed by a conversion. One can note that especially, it means the weakening can always be pushed above the introduction rules of applications, λ-abstractions, dependent arrows and sorts. Hence, the only steps which are not guided by the syntax are the conversions.

Theorem 3.4.13 (Inversion of Sorts)**.** *If $\Gamma \vdash_P s : A$ with $s \in \mathcal{S}$, then there is $s' \in \mathcal{S}$ such that $A \leftrightsquigarrow_\beta^* s'$ and $(s, s') \in \mathcal{A}$.*

Proof. By induction on the typing derivation. The only three rules whose conclusion can be the typing of a sort are (*ax*), (*weak*) and (*conv*).

- If the last rule is (*ax*), the conclusion is of the form $\Gamma \vdash_P s : s'$ with $(s, s') \in \mathcal{A}$.
- If the last rule is (*weak*), then one of the hypotheses is also of the form $\Gamma' \vdash_P s : A$, and one can conclude by the induction hypothesis, using Lemma 3.4.12.
- If the last rule is (*conv*), then one of the hypotheses is of the form $\Gamma \vdash_P s : B$, with $B \leftrightsquigarrow_\beta^* A$. By induction hypothesis, there is a s' such that $(s, s') \in \mathcal{A}$ and $B \leftrightsquigarrow_\beta^* s'$, hence, since $\leftrightsquigarrow_\beta^*$ is transitive, $A \leftrightsquigarrow_\beta^* s'$. □

Theorem 3.4.14 (Inversion of Product)**.** *If $\Gamma \vdash_P (x : A) \Rightarrow B : C$, then there are $s_1, s_2, s_3 \in \mathcal{S}$ such that $C \leftrightsquigarrow_\beta^* s_3$, $(s_1, s_2, s_3) \in \mathcal{R}$, $\Gamma \vdash_P A : s_1$ and $\Gamma, x : A \vdash_P B : s_2$.*

Proof. By induction on the typing derivation. The only three rules which conclusion can be the typing of a product are (*prod*), (*weak*) and (*conv*).

- If the last rule is (*prod*), the conclusion is of the form $\Gamma \vdash_P (x : A) \Rightarrow B : s_3$ with the two hypotheses $\Gamma \vdash_P A : s_1$ and $\Gamma, x : A \vdash_P B : s_2$, and $(s_1, s_2, s_3) \in \mathcal{R}$.
- If the last rule is (*weak*), then the right hypothesis is also of the form $\Gamma' \vdash_P (x : A) \Rightarrow B : C$, and one can conclude by the induction hypothesis.
- If the last rule is (*conv*), then the left hypothesis is of the form $\Gamma \vdash_P (x : A) \Rightarrow B : D$, with $C \leftrightsquigarrow_\beta^* D$. By induction hypothesis, there are s_1, s_2, s_3 such that $\Gamma \vdash_P A : s_1$, $\Gamma, x : A \vdash_P B : s_2$, $D \leftrightsquigarrow_\beta^* s_3$ and $(s_1, s_2, s_3) \in \mathcal{R}$. Hence, since $\leftrightsquigarrow_\beta^*$ is transitive, $C \leftrightsquigarrow_\beta^* s_3$. □

The next two theorems have proofs analogous to the ones of the two previous theorems, hence they are not presented here.

Theorem 3.4.15 (Inversion of Abstraction)**.** *If $\Gamma \vdash_P \lambda (x : A).t : C$, then there is a $B \in \Lambda$ such that $C \leftrightsquigarrow_\beta^* (x : A) \Rightarrow B$ and $\Gamma, x : A \vdash_P t : B$.*

Theorem 3.4.16 (Inversion of Application)**.** *If $\Gamma \vdash_P t\, u : C$, then there are $A, B \in \Lambda$ such that $C \leftrightsquigarrow_\beta^* B\,[u/_x]$, $\Gamma \vdash_P t : (x : A) \Rightarrow B$ and $\Gamma \vdash_P u : A$.*

3.4.4 Embeddings of PTS

Definition 3.4.17 (Functional Pure Type System)**.** *A PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is called* functional *if axioms $\mathcal{A}$ and rules $\mathcal{R}$ are functional relations, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$.*

One can be even more constraining on the relations between sorts, and require $\mathcal{A}$ and $\mathcal{R}$ not simply to be functions, but total functions.

Definition 3.4.18 (Full Pure Type System)**.** *A PTS is called* full *if axioms and rules are total functions, respectively from $\mathcal{S}$ and $\mathcal{S} \times \mathcal{S}$ to $\mathcal{S}$.*

$\lambda^{\to}$ and $\lambda\Pi$, defined in Example 3.4.2, are functional PTS's, but are not full since $\Box$ does not inhabit any sort. On the other hand, $\mathcal{C}^{\infty}$ and $\mathcal{P}^{\infty}$, defined in Example 3.4.3 are full.

Proposition 3.4.19 (Unicity of typing)**.** *If P is a functional PTS, Γ is a context, t, A and B are terms such that $\Gamma \vdash_P t : A$ and $\Gamma \vdash_P t : B$, then $A \twoheadleftarrow\!\!\rightsquigarrow^*_\beta B$.*

Proof. See [Bar92, Lemma 5.2.21] □

Definition 3.4.20 (Specification embedding)**.** *Given $P_1 = (\mathcal{S}_1; \mathcal{A}_1; \mathcal{R}_1)$ and $P_2 = (\mathcal{S}_2; \mathcal{A}_2; \mathcal{R}_2)$ two PTS specifications, $f : \mathcal{S}_1 \to \mathcal{S}_2$ is an embedding of P_1 in P_2 if for all $(s, s') \in \mathcal{A}_1$, we have $(f(s), f(s')) \in \mathcal{A}_2$ and for all $(s, s', s'') \in \mathcal{R}_1$, we have $(f(s), f(s'), f(s'')) \in \mathcal{R}_2$.*

f is extended to terms of P_1 which do not use elements of $\mathcal{S}_2$ as variable names by:

$$f(x) = x, \text{ if } x \in \mathcal{V}; \qquad f(\lambda\,(x : A)\,.t) = \lambda\,(x : f(A))\,.f(t);$$
$$f(t\,u) = f(t)\,f(u); \qquad f((x : A) \Rightarrow B) = (x : f(A)) \Rightarrow f(B).$$

Then f is extended to contexts by: $f([\,]) = [\,]$ and $f(\Gamma, x : A) = f(\Gamma), x : f(A)$.

Proposition 3.4.21 (Soundness of the Embedding)**.** *If f is an embedding from a PTS P_1 to P_2, if $\Gamma \vdash_{P_1} t : A$ and no elements of $\mathcal{S}_2$ are used as variable in Γ, t, A, then $f(\Gamma) \vdash_{P_2} f(t) : f(A)$.*

Proof. By induction on the proof tree.

(var) By induction hypothesis, $f(\Gamma) \vdash_{P_2} f(A) : f(s)$. Since $f : \mathcal{S}_1 \to \mathcal{S}_2$, we have $f(s) \in \mathcal{S}_2$. Since f does not modify variable names, one still has $x \in \mathcal{V} \setminus \mathrm{dom}(f(\Gamma))$. Hence, one can use the (var) rule and obtain $f(\Gamma), x : f(A) \vdash_{P_2} x : f(A)$.

(weak) This case is very similar to the (var) one.

(ax) By definition of the embedding, since $(s_1, s_2) \in \mathcal{A}_1$, $(f(s_1), f(s_2)) \in \mathcal{A}_2$. Hence, one can use (ax) to conclude that $[\,] \vdash_{P_2} f(s_1) : f(s_2)$.

(prod) By induction hypothesis, $f(\Gamma) \vdash_{P_2} f(A) : f(s_1)$ and $f(\Gamma), x : f(A) \vdash_{P_2} f(B) : f(s_2)$. By definition of embedding, since $(s_1, s_2, s_3) \in \mathcal{R}_1$, $(f(s_1), f(s_2), f(s_3)) \in \mathcal{R}_2$. Once again, one can apply $(prod)$ and obtain $f(\Gamma) \vdash_{P_2} (x : f(A)) \Rightarrow f(B) : f(s_3)$.

(app) and (abs) Those cases are straightforward, just like the ones previously treated.

(conv) Since f does not affect the structure of terms, and only changes the name of the sorts, for any A and B such that $A \twoheadleftarrow\!\!\rightsquigarrow^*_\beta B$, one has $f(A) \twoheadleftarrow\!\!\rightsquigarrow^*_\beta f(B)$. Hence, one can simply apply the $(conv)$ rule to deduce from the induction hypotheses that $f(\Gamma) \vdash_{P_2} f(t) : f(B)$.

□

3.5 Subject Reduction

Until now, we have defined two orthogonal notions:

- in Section 3.3, we have defined β-reduction and said that it is the computation rule associated to function application;
- and in Section 3.4.2, we have defined a notion of typing.

Since computing should not modify the type of a term, we expect to have a theorem stating that a reduct has the same type as the original term. This property is called *subject reduction* and is proved in Theorem 3.5.4.

3.5.1 Substitution

A first step toward this property is to prove that one can eliminate "cuts", meaning that if a term inhabits type B, under hypothesis $\Gamma, x : A, \Gamma'$, and if under hypothesis Γ, one can construct a term u of type A, then it is possible to eliminate the hypothesis $x : A$, by replacing x by u. More formally:

Lemma 3.5.1 (Substitution Lemma)**.** *In a PTS P, if $\Gamma, x : A, \Gamma' \vdash_P t : B$ and $\Gamma \vdash_P u : A$, then $\Gamma, \Gamma'\left[u/_x\right] \vdash_P t\left[u/_x\right] : B\left[u/_x\right]$.*

Proof. First, one must note that since u is typable in the context Γ, $\mathrm{FV}(u) \subseteq \mathrm{dom}(\Gamma)$. Especially, $x \notin \mathrm{FV}(u)$. By induction on the proof tree, if the last rule is:

(var) We need to distinguish two subcases.

- If the introduced variable is not the one we are eliminating:

$$\frac{\Gamma, x : A, \Gamma' \vdash_P C : s}{\Gamma, x : A, \Gamma', y : C \vdash_P y : C} \begin{cases} y \in \mathcal{V} \setminus \mathrm{dom}(\Gamma, x : A, \Gamma') \\ s \in \mathcal{S} \end{cases}.$$

  By induction hypothesis, one has $\Gamma, \Gamma'\left[u/_x\right] \vdash_P C\left[u/_x\right] : s$.
  y is still in $\mathcal{V} \setminus \mathrm{dom}(\Gamma, \Gamma'\left[u/_x\right])$, hence one can apply the (var) rule and conclude $\Gamma, \Gamma'\left[u/_x\right], y : C\left[u/_x\right] \vdash_P y : C\left[u/_x\right]$.

- If the introduced variable is the one we are eliminating:

$$\frac{\Gamma \vdash_P A : s}{\Gamma, x : A \vdash_P x : A} \begin{cases} x \in \mathcal{V} \setminus \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases}.$$

  We want to prove $\Gamma \vdash_P x\left[u/_x\right] : A\left[u/_x\right]$. By hypothesis, $\Gamma \vdash_P u : A$. Since $\Gamma \vdash_P A : s$, and because all free variables in A are in Γ, one has $A\left[u/_x\right] = A$. Hence the conclusion is proved.

(ax) This case is irrelevant to this proof, since it requires the context to be empty.

(prod) $$\frac{\Gamma, x : A, \Gamma' \vdash_P B : s_1 \quad \Gamma, x : A, \Gamma', y : B \vdash_P C : s_2}{\Gamma, x : A, \Gamma' \vdash_P (y : B) \Rightarrow C : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

By induction, $\Gamma, \Gamma'\left[u/_x\right] \vdash_P B\left[u/_x\right] : s_1$ and $\Gamma, \Gamma'\left[u/_x\right], y : B\left[u/_x\right] \vdash_P C\left[u/_x\right] : s_2$. One can apply the $(prod)$ rule and conclude $\Gamma, \Gamma'\left[u/_x\right] \vdash_P (y : B\left[u/_x\right]) \Rightarrow C\left[u/_x\right] : s_3$. Since y is distinct from x, $(y : B\left[u/_x\right]) \Rightarrow C\left[u/_x\right] = ((y : B) \Rightarrow C)\left[u/_x\right]$. Hence this conclusion is the one we were trying to prove.

(app) $\dfrac{\Gamma, x : A, \Gamma' \vdash_P t : (y : B) \Rightarrow C \quad \Gamma, x : A, \Gamma' \vdash_P v : B}{\Gamma, x : A, \Gamma' \vdash_P t\, v : C\left[v/_y\right]}$

By induction hypothesis, one can prove $\Gamma, \Gamma'\left[u/_x\right] \vdash_P t\left[u/_x\right] : ((y : B) \Rightarrow C)\left[u/_x\right]$ and $\Gamma, \Gamma'\left[u/_x\right] \vdash_P v\left[u/_x\right] : B\left[u/_x\right]$.

Since y is a bound variable, it can be chosen distinct of x and of $\mathrm{FV}(u)$, hence $((y : B) \Rightarrow C)\left[u/_x\right] = (y : B\left[u/_x\right]) \Rightarrow C\left[u/_x\right]$, so the rule (app) can be applied to conclude: $\Gamma, \Gamma'\left[u/_x\right] \vdash_P t\left[u/_x\right]\ (v\left[u/_x\right]) : (C\left[u/_x\right])\left[v\left[u/_x\right]/_y\right]$.

Since all free variables in u are declared in Γ, y is not one of them, hence one can apply Lemma 3.3.5, to rewrite the conclusion as $\Gamma, \Gamma'\left[u/_x\right] \vdash_P (t\, v)\left[u/_x\right] : \left(C\left[v/_y\right]\right)\left[u/_x\right]$. This conclusion is the one we were trying to prove.

(abs) $\dfrac{\Gamma, x : A, \Gamma' \vdash_P (y : B) \Rightarrow C : s \quad \Gamma, x : A, \Gamma', y : B \vdash_P t : C}{\Gamma, x : A, \Gamma' \vdash_P \lambda\,(y : B)\,.t : (y : B) \Rightarrow C}$

By induction hypothesis, one can prove $\Gamma, \Gamma'\left[u/_x\right] \vdash_P ((y : B) \Rightarrow C)\left[u/_x\right] : s$ and also $\Gamma, \Gamma'\left[u/_x\right], y : B\left[u/_x\right] \vdash_P t\left[u/_x\right] : C\left[u/_x\right]$.

Once again, the substitution can be distributed over the arrow, to apply (abs) and then be refactorized to operate on the whole term. The obtained conclusion is the expected one, $\Gamma, \Gamma'\left[u/_x\right] \vdash_P (\lambda\,(y : B)\,.t)\left[u/_x\right] : ((y : B) \Rightarrow C)\left[u/_x\right]$.

(conv) $\dfrac{\Gamma, x : A, \Gamma' \vdash_P t : B \quad \Gamma, x : A, \Gamma' \vdash_P C : s}{\Gamma, x : A, \Gamma' \vdash_P t : C}\ B \leftrightsquigarrow_\beta^* C$

By induction hypothesis, one can prove $\Gamma, \Gamma'\left[u/_x\right] \vdash_P t\left[u/_x\right] : B\left[u/_x\right]$ and also $\Gamma, \Gamma'\left[u/_x\right] \vdash_P C\left[u/_x\right] : s$. By Proposition 3.3.15, $B\left[u/_x\right] \leftrightsquigarrow_\beta^* C\left[u/_x\right]$, hence one can apply the $(conv)$ rule and conclude.

(weak) Just like for the (var) rule, we need to distinguish two subcases.

- If the introduced variable is not the one we are eliminating:

$$\frac{\Gamma, x : A, \Gamma' \vdash_P B : s \quad \Gamma, x : A, \Gamma' \vdash_P t : C}{\Gamma, x : A, \Gamma', y : B \vdash_P t : C}\ \begin{cases} y \in \mathcal{V} \setminus \mathrm{dom}(\Gamma, x : A, \Gamma') \\ s \in \mathcal{S} \end{cases}.$$

  By induction hypothesis, $\Gamma, \Gamma'\left[u/_x\right] \vdash_P B\left[u/_x\right] : s$ and $\Gamma, \Gamma'\left[u/_x\right] \vdash_P t\left[u/_x\right] : C\left[u/_x\right]$. y is still in $\mathcal{V} \setminus \mathrm{dom}(\Gamma, \Gamma'\left[u/_x\right])$, hence one can apply the $(weak)$ rule and conclude $\Gamma, \Gamma'\left[u/_x\right], y : B\left[u/_x\right] \vdash_P t\left[u/_x\right] : C\left[u/_x\right]$.

- If the introduced variable is the one we are eliminating:

$$\frac{\Gamma \vdash_P A : s \quad \Gamma \vdash_P t : C}{\Gamma, x : A \vdash_P t : C}\ \begin{cases} x \in \mathcal{V} \setminus \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases}.$$

  We want to prove $\Gamma \vdash_P t\left[u/_x\right] : C\left[u/_x\right]$. And we have $\Gamma \vdash_P t : C$. Since all free variables in t and C are in Γ, $t\left[u/_x\right] = t$ and $C\left[u/_x\right] = C$. Hence the conclusion is proved. □

Before interesting ourselves to the interaction between typability and reduction, let us state a very useful first consequence of the substitution lemma.

Theorem 3.5.2 (Sortability)**.** *If $\Gamma \vdash_P t : A$, then either A is a top sort or there is a $s \in \mathcal{S}$ such that $\Gamma \vdash_P A : s$.*

Proof. By induction on the typing derivation

(var) The premise is $\Gamma \vdash_P A : s$. One can apply (*weak*) using twice this judgement to conclude that $\Gamma, x : A \vdash_P A : s$.

(ax) and (prod) The conclusion is an inhabitation in a sort, hence, either it is a top sort, or one can apply the (*ax*) rule to type it. In the case of (*prod*), one also has to do several weakenings to reconstruct the expected context. Here it must be noted that those weakenings can be done, since in a proof of $\Gamma, x : A, \Gamma' \vdash_P t : B$ there is a subtree which is a proof of $\Gamma \vdash_P B : s$ for a sort s.

(app) By induction, there is a $s \in \mathcal{S}$ such that $\Gamma \vdash_P (x : A) \Rightarrow B : s$, hence, by inversion (Section 3.4.3), there is a $s' \in \mathcal{S}$ such that $\Gamma, x : A \vdash_P B : s'$. Since one also has $\Gamma \vdash_P u : A$, by substitution (Lemma 3.5.1), one can conclude that $\Gamma \vdash_P B\left[u/_x\right] : s'$.

(abs) and (conv) The typability of the type of the conclusion is one of the hypotheses.

(weak) By induction hypothesis, if B is not a top sort, there is a $s \in \mathcal{S}$ such that $\Gamma \vdash_P B : s$. One can apply a weakening again to conclude that $\Gamma, x : A \vdash_P B : s$.

□

3.5.2 Subject Reduction

Lemma 3.5.3 (Head Subject Reduction)**.** *Let $\Gamma \in \mathfrak{C}$ and $A \in \Lambda$. If $t, u \in \Lambda$ are such that $t \overset{\varepsilon}{\rightsquigarrow}_\beta u$ and $\Gamma \vdash_P t : A$, then $\Gamma \vdash_P u : A$.*

Proof. Since $t \overset{\varepsilon}{\rightsquigarrow}_\beta u$, there are $v, B, w \in \Lambda$ such that $t = (\lambda\,(x : B)\,.v)\ w$ and $u = v\left[w/_x\right]$. By hypothesis $\Gamma \vdash_P (\lambda\,(x : B)\,.v)\ w : A$, hence, by inversion (Section 3.4.3), there are $C, D \in \Lambda$ such that $A \leftrightsquigarrow^*_\beta D\left[w/_x\right]$, $\Gamma \vdash_P w : C$ and $\Gamma \vdash_P \lambda\,(x : B)\,.v : (x : C) \Rightarrow D$. By inversion again, there is a $E \in \Lambda$ such that $(x : C) \Rightarrow D \leftrightsquigarrow^*_\beta (x : B) \Rightarrow E$ and $\Gamma, x : B \vdash_P v : E$.

By injectivity of product Proposition 3.3.16, $C \leftrightsquigarrow^*_\beta B$ and $D \leftrightsquigarrow^*_\beta E$. Hence, applying (*conv*), one deduces $\Gamma, x : B \vdash_P v : D$ and $\Gamma \vdash_P w : B$.

By the substitution lemma (Lemma 3.5.1), $\Gamma \vdash_P v\left[w/_x\right] : D\left[w/_x\right]$. Since $v\left[w/_x\right]$ is exactly u and $A \leftrightsquigarrow^*_\beta D\left[w/_x\right]$, the (*conv*) rule allows us to conclude that $\Gamma \vdash_P u : A$. □

Theorem 3.5.4 (Subject Reduction)**.** *Let $\Gamma \in \mathfrak{C}$ and $A \in \Lambda$. If $t, u \in \Lambda$ are such that $t \rightsquigarrow_\beta u$ and $\Gamma \vdash_P t : A$, then $\Gamma \vdash_P u : A$.*

Proof. Using inversion Section 3.4.3, one can easily select the subterm where the reduction occurred and then apply Lemma 3.5.3. □

3.6 Equivalent Presentations of the Typing Rules

Some other equivalent presentations of the typing rules can also be found in the literature. Some of them can be useful, since they permit to do some proofs more easily. In this section, we will present two of them.

3.6.1 Typing Rules with Context Formation Predicate

First, we discuss a very common presentation, with a predicate stating that a context is well-formed defined by mutual induction with the typing relation. This system is quite convenient since it avoids the weakening rule, hence simplifies the proofs by induction on the typing rules.

Definition 3.6.1 (Typing With Context Formation Predicate)**.** *is WF and* $\_ \vdash' \_ : \_$ *are predicates defined respectively on* $\mathfrak{C}$ *and* $\mathfrak{C} \times \Lambda \times \Lambda$*. They are defined by mutual induction, with the rules:*

$$\textit{(empty)}\quad \frac{}{[]\ \textit{is WF}} \qquad \textit{(decl)}\quad \frac{\Gamma \vdash' A : s}{\Gamma, x : A\ \textit{is WF}} \begin{cases} x \notin \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases}$$

$$\textit{(ax)}\quad \frac{\Gamma\ \textit{is WF}}{\Gamma \vdash' s : s'}\ (s, s') \in \mathcal{A} \qquad \textit{(var)}\quad \frac{\Gamma, x : A, \Gamma'\ \textit{is WF}}{\Gamma, x : A, \Gamma' \vdash' x : A}$$

$$\textit{(prod)}\quad \frac{\Gamma \vdash' A : s_1 \quad \Gamma, x : A \vdash' B : s_2}{\Gamma \vdash' (x : A) \Rightarrow B : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textit{(app)}\quad \frac{\Gamma \vdash' t : (x : A) \Rightarrow B \quad \Gamma \vdash' u : A}{\Gamma \vdash' t\, u : B\,[u/_x]} \qquad \textit{(abs)}\quad \frac{\Gamma \vdash' (x : A) \Rightarrow B : s \quad \Gamma, x : A \vdash' t : B}{\Gamma \vdash' \lambda\,(x : A)\,.t : (x : A) \Rightarrow B}$$

$$\textit{(conv)}\quad \frac{\Gamma \vdash' t : A \quad \Gamma \vdash' B : s}{\Gamma \vdash' t : B}\ A \leftrightsquigarrow^*_\beta B$$

Lemma 3.6.2 (Well-formedness of contexts)**.** *Given* $\Gamma \in \mathfrak{C}$ *and* $t, A \in \Lambda$*, if* $\Gamma \vdash' t : A$*, then* Γ *is WF.*

Proof. It is direct by induction. □

Lemma 3.6.3 (Context Extension)**.** *Given* $\Gamma, \Gamma', \Gamma'' \in \mathfrak{C}$ *and* $t, A \in \Lambda$*, if* $\Gamma, \Gamma'' \vdash' t : A$ *and* $\Gamma, \Gamma', \Gamma''$ *is WF, then* $\Gamma, \Gamma', \Gamma'' \vdash' t : A$*.*

Proof. By induction on the derivation of $\Gamma, \Gamma'' \vdash' t : A$.

(ax) Since $\Gamma, \Gamma', \Gamma''$ is WF, one can directly apply the (ax) rule.

(var) Since $x : A$ is in Γ, Γ'', it is in $\Gamma, \Gamma', \Gamma''$ which is well-formed. Hence one can apply the rule (var).

(prod) By induction hypothesis, $\Gamma, \Gamma', \Gamma'' \vdash' A : s_1$, hence $\Gamma, \Gamma', \Gamma'', x : A$ is WF. So, the induction hypothesis can be applied to the other premise, hence $\Gamma, \Gamma', \Gamma'', x : A \vdash' B : s_2$. Now, the rule $(prod)$ is applicable and leads to the conclusion we were looking for.

(abs) This case is analogous to the $(prod)$ one.

(app) and (conv) Since those cases do not involve a bound variable, they are direct by application of the induction hypothesis.

□

Proposition 3.6.4 (Completeness of $\vdash'$)**.** *Given* $\Gamma \in \mathfrak{C}$ *and* $t, A \in \Lambda$*, if* $\Gamma \vdash_P t : A$*, then* $\Gamma \vdash' t : A$*.*

Proof. By induction on the typing derivation for $\vdash_P$.

(prod), (abs), (app) and (conv) Those rules are identical in both systems, hence it is a direct application of the induction hypothesis.

(ax) This rule is simulated by the tree

$$(ax)\ \frac{(empty)\ \dfrac{}{[\,]\text{ is WF}}}{[\,]\vdash' s : s'}\ (s,s')\in\mathcal{A}$$

(var) This rule is simulated by the tree

$$\begin{array}{c}(decl)\ \dfrac{\Gamma\vdash' A : s}{\Gamma, x : A\text{ is WF}}\ \begin{cases} x\notin \mathrm{dom}(\Gamma)\\ s\in\mathcal{S}\end{cases}\\ (var)\ \dfrac{}{\Gamma, x : A\vdash' x : A}\end{array}$$

(weak) It is a consequence of Lemma 3.6.3.

□

Proposition 3.6.5 (Conservativity of $\vdash'$)**.** *Given $\Gamma\in\mathfrak{C}$ and $t, A\in\Lambda$, if $\Gamma\vdash' t : A$, then $\Gamma\vdash_P t : A$.*

Proof. By induction on the typing derivation for $\vdash'$. Once again, the cases (*prod*), (*app*), (*abs*) and (*conv*) are direct, hence the only two cases one has to focus on are (*ax*) and (*var*).

(ax) By induction on the length of Γ, we prove that if $\Gamma\vdash' s : s'$ with $(s,s')\in\mathcal{A}$, then $\Gamma\vdash_P s : s'$. The rule preceding (*ax*) is necessarily (*empty*) or (*decl*).

- If the rule preceding (*ax*) is (*empty*), then this sequence of two rules is simulated by the rule (*ax*) of $\vdash_P$.
- If the rule preceding (*ax*) is (*decl*), the derivation is:

$$\begin{array}{c}(\mathrm{decl})\ \dfrac{\Delta\vdash' A : s''}{\Delta, x : A\text{ is WF}}\ \begin{cases} x\notin\mathrm{dom}(\Delta)\\ s\in\mathcal{S}\\ (s,s')\in\mathcal{A}\end{cases}\\ (\mathrm{ax})\ \dfrac{}{\Delta, x : A\vdash' s : s'}\end{array}$$

  By the global induction hypothesis, $\Delta\vdash_P A : s''$. By Lemma 3.6.2, one has that Δ is WF, hence, one can apply (*ax*) to get a proof of $\Delta\vdash' s : s'$, and applying the local induction hypothesis, $\Delta\vdash_P s : s'$. Then, one can apply the rule (*weak*) to conclude.

(var) The rule preceding (*var*) is necessarily (*decl*) and we prove by induction on the length of Δ' that if $\Delta, x : A, \Delta'\vdash' x : A$, then $\Delta, x : A, \Delta'\vdash_P x : A$.

- If Δ' is empty, then the rules (*decl*) followed by (*var*) is simulated by the rule (*var*) of $\vdash_P$.
- Otherwise, the derivation is:

$$\begin{array}{c}(\mathrm{decl})\ \dfrac{\Delta, x : A, \Delta'\vdash' B : s}{\Delta, x : A, \Delta', y : B\text{ is WF}}\ \begin{cases} y\notin\mathrm{dom}(\Delta)\\ s\in\mathcal{S}\end{cases}\\ (\mathrm{var})\ \dfrac{}{\Delta, x : A, \Delta', y : B\vdash' x : A}\end{array}$$

  By the global induction hypothesis, $\Delta, x : A, \Delta'\vdash_P B : s$. By Lemma 3.6.2, one has that $\Delta, x : A, \Delta'$ is WF, so one can apply (*var*) to obtain $\Delta, x : A, \Delta'\vdash' x : A$, and applying the local induction hypothesis, $\Delta, x : A, \Delta'\vdash_P x : A$. Then, one can apply the rule (*weak*) and conclude. □

3.6.2 Type System With Explicit Sorting of All Types

Now, we will define a system which looks a lot like Definition 3.4.9, but always explicitly requires the typability of the type to apply a rule.

Definition 3.6.6 (Typing With Explicit Sorting of All Types)**.** $\_ \Vdash \_ : \_$ *is the relation included in* $\mathfrak{C} \times \Lambda \times \Lambda$ *defined by:*

$$\textit{(var)} \quad \frac{\Gamma \Vdash A : s}{\Gamma, x : A \Vdash x : A} \begin{cases} x \in \mathcal{V} \setminus \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases}$$

$$\textit{(ax)} \quad \frac{}{[\,] \Vdash s_1 : s_2}\ (s_1, s_2) \in \mathcal{A}$$

$$\textit{(prod)} \quad \frac{\Gamma \Vdash A : s_1 \quad \Gamma, x : A \Vdash B : s_2}{\Gamma \Vdash (x : A) \Rightarrow B : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textit{(abs)} \quad \frac{\Gamma \Vdash (x : A) \Rightarrow B : s \quad \Gamma, x : A \Vdash t : B}{\Gamma \Vdash \lambda\,(x : A)\,.t : (x : A) \Rightarrow B}$$

$$\textit{(conv)} \quad \frac{\Gamma \Vdash t : A \quad \Gamma \Vdash B : s}{\Gamma \Vdash t : B}\ A \leftrightsquigarrow^*_\beta B$$

$$\textit{(app)} \quad \frac{\Gamma \Vdash t : (x : A) \Rightarrow B \quad \Gamma \Vdash u : A \quad \Gamma \Vdash B\,[u/_x] : s}{\Gamma \Vdash t\,u : B\,[u/_x]}$$

$$\textit{(weak)} \quad \frac{\Gamma \Vdash A : s \quad \Gamma \Vdash t : B}{\Gamma, x : A \Vdash t : B} \begin{cases} x \in \mathcal{V} \setminus \mathrm{dom}(\Gamma) \\ s \in \mathcal{S} \end{cases}$$

The only difference between $\vdash_P$ and $\Vdash$ is the premise $\Gamma \Vdash B\,[u/_x]$ in the typing rule (*app*).

One should note, that in all the rules, but the weakening, either the type inhabited in the conclusion is a sort, or there is a premise stating that this type inhabits in a sort. Regarding the weakening, it is not possible to add a premise stating that B inhabits in a sort, since B might itself be a top sort. However, this is not a trouble, since the right premise of the weakening states that t inhabits in B, hence either B is a sort, or there is a sequence of weakening and directly above it a premise states that B inhabits in a sort.

At first glance, one could think that the equivalence between this system and Definition 3.4.9 is a direct consequence of Theorem 3.5.2, but the proof is in fact much more subtle than expected.

Indeed, Theorem 3.5.2 ensures us that $\Gamma \vdash_P t\,u : B\,[u/_x]$ implies that there is a sort s such that $\Gamma \vdash_P B\,[u/_x] : s$. However, the proof produced by this theorem does not appear to be smaller than the one of $\Gamma \vdash_P t\,u : B\,[u/_x]$, hence doing an induction does not appear possible to prove that $\vdash_P$ is a relation included in $\Vdash$.

Let us first note that the other direction is very simple:

Proposition 3.6.7 (Conservativity of $\Vdash$)**.** *Given* $\Gamma \in \mathfrak{C}$*,* $t, A \in \Lambda$ *such that* $\Gamma \Vdash t : A$*, then* $\Gamma \vdash_P t : A$*.*

Proof. This is a direct induction, one just has to erase the new premises. □

For the reverse implication, our strategy will be to first prove that $\Vdash$ is well-designed, in the sense that one can produce equivalents of Theorem 3.5.2, Theorem 3.4.14 and Lemma 3.5.1 that apply directly to $\Vdash$. Once those properties are proved, the implication we are looking for can be obtained by a rather straightforward induction, the premise added to the (*app*) rule can be crafted thanks to those three results.

Proposition 3.6.8. *If* $\Gamma \Vdash t : A$*, then* A *is a sort or there is a* $s \in \mathcal{S}$ *such that* $\Gamma \Vdash A : s$*.*

Proof. By design, whenever A is not a sort, one of the premises of the typing rule states the typability of A. Potentially, one has to go through a sequence of weakening to find this premise. □

Proposition 3.6.9. *If* $\Gamma \Vdash (x : A) \Rightarrow B : C$*, then there are* $s_1, s_2, s_3 \in \mathcal{S}$ *such that* $C \leftrightsquigarrow_\beta^* s_3$*,* $(s_1, s_2, s_3) \in \mathcal{R}$*,* $\Gamma \Vdash A : s_1$ *and* $\Gamma, x : A \Vdash B : s_2$*.*

Proof. The proof is identical to the one of Theorem 3.4.14. Indeed, the new rule (*app*) cannot lead to a conclusion which is the typing of a product. □

Proposition 3.6.10. *If* $\Gamma, x : A, \Gamma' \Vdash t : B$ *and* $\Gamma \Vdash u : A$*, then* $\Gamma, \Gamma'\left[u/_x\right] \Vdash t\left[u/_x\right] : B\left[u/_x\right]$*.*

Proof. By induction on the proof tree. The proof is identical to the one of Lemma 3.5.1 for all cases but the new one, which corresponds to the rule (*app*):

$$\textbf{(app)}\ \frac{\Gamma, x : A, \Gamma' \Vdash t : (y : B) \Rightarrow C \quad \Gamma, x : A, \Gamma' \Vdash v : B \quad \Gamma, x : A, \Gamma' \Vdash C\left[v/_y\right] : s}{\Gamma, x : A, \Gamma' \Vdash t\, v : C\left[v/_y\right]}$$

By induction hypothesis, one can prove $\Gamma, \Gamma'\left[u/_x\right] \Vdash t\left[u/_x\right] : ((y : B) \Rightarrow C)\left[u/_x\right]$, $\Gamma, \Gamma'\left[u/_x\right] \Vdash v\left[u/_x\right] : B\left[u/_x\right]$ and $\Gamma, \Gamma'\left[u/_x\right] \Vdash \left(C\left[v/_y\right]\right)\left[u/_x\right] : s$.

By Lemma 3.3.5, $\left(C\left[v/_y\right]\right)\left[u/_x\right] = \left(C\left[u/_x\right]\right)\left[v\left[u/_x\right]/_y\right]$, since y is a bound variable, which is chosen fresh whereas $\mathrm{FV}(u) \subseteq \mathrm{dom}(\Gamma)$, ensuring that y does not occur in u. So, one can rewrite the third induction hypothesis as $\Gamma, \Gamma'\left[u/_x\right] \Vdash \left(C\left[u/_x\right]\right)\left[v\left[u/_x\right]/_y\right] : s$

Since $((y : B) \Rightarrow C)\left[u/_x\right] = (y : B\left[u/_x\right]) \Rightarrow C\left[u/_x\right]$, so the rule (*app*) can be applied to conclude: $\Gamma, \Gamma'\left[u/_x\right] \Vdash t\left[u/_x\right]\ \left(v\left[u/_x\right]\right) : \left(C\left[u/_x\right]\right)\left[v\left[u/_x\right]/_y\right]$.

One can apply Lemma 3.3.5 in the reverse direction, to rewrite the conclusion as $\Gamma, \Gamma'\left[u/_x\right] \Vdash (t\, v)\left[u/_x\right] : \left(C\left[v/_y\right]\right)\left[u/_x\right]$. This conclusion is the one we were trying to prove. □

Those three propositions were the only results lacking to prove the non-trivial inclusion of $\vdash_P$ in $\Vdash$.

Theorem 3.6.11 (Completeness of $\Vdash$)**.** *Given* $\Gamma \in \mathfrak{C}$*,* $t, A \in \Lambda$ *such that* $\Gamma \vdash_P t : A$*, then* $\Gamma \Vdash t : A$*.*

Proof. By induction on the typing derivation for $\vdash_P$. Again, one just has to treat the case (*app*), since for all the other ones, the induction is very direct, since the premises are the same for $\vdash_P$ and $\Vdash$.

$$\textbf{(app)}\ \frac{\Gamma \vdash_P t : (x : A) \Rightarrow B \quad \Gamma \vdash_P u : A}{\Gamma \vdash_P t\, u : B\left[u/_x\right]}$$

By induction hypothesis, one has $\Gamma \Vdash t : (x : A) \Rightarrow B$ and $\Gamma \Vdash u : A$. By Proposition 3.6.8, one has a sort s such that $\Gamma \Vdash (x : A) \Rightarrow B : s$.

Then by Proposition 3.6.9, one can obtain a sort s' such that $\Gamma, x : A \Vdash B : s'$.

Applying Proposition 3.6.10, one has $\Gamma \Vdash B\left[u/_x\right] : s'$.

Here, one has the three required premises to apply the (*app*) rule, and then conclude that $\Gamma \Vdash t\, u : B\left[u/_x\right]$. □

Chapter 4

Rewriting Type Systems

With *Pure Type Systems*, we described a powerful way to declare new constructors and combine them to produce type-safe constructions, for instance of mathematical objects. As an illustration, one can declare that $\mathbb{N} : \star$, that $0 : \mathbb{N}$ and $+ : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$, and then conclude that $0 + 0$ (which is a human-readable notation to denote the term $+\,0\,0$) is a natural number (i.e. has type $\mathbb{N}$). More formally, $\mathbb{N} : \star, 0 : \mathbb{N}, + : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} \vdash_{\lambda\Pi} +\,0\,0 : \mathbb{N}$.

Furthermore, with β-reduction, PTS's are equipped with a process to perform computation.

However, one would like those computations to be performed on user-declared datatypes, like natural numbers, lists, booleans, rational numbers and so on. For instance, one could like $\texttt{List}\,(0 + 0)$ to be equal to $\texttt{List}\,0$.

One solution to have conversions between user-defined datatypes could be to encode them directly in the λ-calculus, and then use β-reduction to perform the computation of the functions the user defined. This approach is called the "Church encoding" of datatypes. However, this solution is not the one we are interested in. Indeed, this way to define types and values lead to very deeply encoded, unreadable, terms. Furthermore, as described extensively in the next chapter, this thesis is part of the "Dedukti project", which relies on the encoding of logical systems in a unique framework. To do so some of the logical parts of proofs are encoded, already affecting readability, there were no reason to obfuscate even more the statements we are working with by multiplying encodings.

Another subtlety must be noted here. Even if one could simply declare a symbol $=: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \text{Prop}$, and then have an axiom $0\_\texttt{neutral} : (x : \mathbb{N}) \Rightarrow 0 + x = x$, it would not be sufficient to have an operational proof checker. Indeed, it would then be possible to "prove" $0 + 0 = 0$ (i.e. to find a term which inhabits this type), however, due to the unicity of typing, no term can have both type $\texttt{List}\,0$ and $\texttt{List}\,(0 + 0)$. This means that, even if $[] : \texttt{List}\,0$ and if there is a proof that $0 + 0 = 0$, it is not simple to inhabit $\texttt{List}\,(0 + 0)$, since in most type theories, unicity of typing prevents $[]$ itself to live in $\texttt{List}\,(0 + 0)$. Indeed, the equality $0 = 0 + 0$ has been declared axiomatically, whereas $0 \not\hookrightarrow^*_\beta 0 + 0$, preventing us from using the rule (*conv*).

To cope with those issues, in *System T* [Göd58], Gödel introduces the recursor on natural numbers in order to enrich conversion with the usual functions on natural numbers. In most proof assistants now, the user is provided with a more general mechanism to enrich conversion, the ability to declare inductive types and to compute with their recursors [Acz77, PPM90].

4.1 Rewriting Rules

But one can have an even more general mechanism to enrich conversion, and allow a broader class of equalities to have a computational behaviour than just the definition of recursors. Those equalities "able to compute" are called rewriting rules, and are introduced in the Deduction Modulo Rewriting [DHK03, Bla01]. This will define an extension of Pure Type Systems, that we will call *Rewriting Type Systems*. Those type systems are a special case of *Type Systems Modulo* [Bla01], called by Blanqui *Algebraic Type Systems*, where the conversion relation is the joinability by a rewriting system, together with the β-reduction of the λ-calculus (Definition 3.3.10).

One must note here that we are performing what is called "higher-order rewriting" in this thesis. However, since the word "higher-order" is employed to designate a wide range of rewriting formalisms, with very different properties and expressiveness, one must be more explicit on what is meant by this expression. We employ it in a rather weak meaning, since we only mean here that we enrich the λ-calculus with rewriting and allow the right-hand sides of rewriting rules to contain λ-abstractions and applications of variables. However, our definition of patterns does not contain λ-abstractions. But it must be noted that even if the shape of the rewriting rules are rather restrictive, compared to most works on "higher-order rewriting", we allow ourselves to rewrite types and dependent arrows can appear in the right-hand sides of our rewriting rules.

There exist two main versions of such patterns with λ-abstractions: Nipkow's "Higher-Order Rewrite System" (HRS) [Nip91] and Klop's "Combinatory Reduction System" (CRS) [Klo80, KvOvR93]. The main differences between those two approaches are studied in details in [vOvR93]. One can emphasize the two main variations:

- to lighten the fact that rewriting rules are schemes of equalities between terms, CRS introduces a notion of meta-terms, which is a syntactic category disjoint of the one of term, whereas HRS reuses the standard terms and defines "patterns" like Miller [Mil91] as a subset of the set of terms;
- and HRS performs β-normalization on the fly when a rewriting rule is applied, whereas CRS adopts a more lazy evaluation strategy.

One could also cite Jouannaud and Okada's "Algebraic Functional System" (AFS) [JO97], which is, roughly speaking, a typed version of Klop's CRS frequently used in higher-order termination criteria. It has been enriched by Kop [Kop12] to "Algebraic Functional Systems with Meta-variables", a formalism designed to be a framework in which one can encode rewriting systems originating from various formalisms, while preserving non-termination. In this work, we present an extension of AFS for dependently-typed systems.

4.1.1 Signature

We will equip some of the symbols with computational behaviour. This will be achieved by the mean of *rewriting rules*, which are simply schemes of equalities with a favourite direction, which can then be instantiated.

In order to define rewriting rules, one has to distinguish two families of symbols, which are confused in the original PTS setting:

- the global symbols which are required to define the theory, that will be called "signature symbols";
- and the local variables that are temporarily introduced but will be bound in the final statement.

Definition 4.1.1 (Signature Symbols)**.** *Let $\mathcal{F}$ be a finite set of names, disjoint of $\mathcal{V}$ and $\mathcal{S}$.*

4.1.2 Patterns

A rewriting rule is more than a simple ordered pair of terms, since one does not want to declare that $0+0=0$ and have another rule to state that $0+1=1$, and so on. To overcome this, rewriting rules are relations between schemes of terms. However, since rules are oriented equalities, left-hand and right-hand sides do not play symmetrical roles, hence it is quite natural that they do not allow exactly the same syntax. It is more restricted in the left-hand side than in the right-hand one. With the signature, one identifies the symbols on which rewriting rules will be defined. This will be the first restriction on the left-hand sides of rules. One will also specify the shape of the arguments this symbol can be applied to. This is the syntactic class of *patterns*.

In fact, we will not use exactly the same syntax for *patterns* and terms, since patterns cannot contain λ-abstractions, sorts or products.

Definition 4.1.2 (Pattern)**.** *A* pattern *is described by the syntactic category P below.*

$$P ::= f\, M^* \text{ with } f \in \mathcal{F}$$
$$M ::= P \mid x \in \mathcal{V}$$

In the previous definition, M is the syntactic category of terms without sorts or products, but also without λ-abstractions. Furthermore, applications are restricted to the application of a function symbol from the signature, meaning especially that one cannot apply a variable or create a β-redex.

The class P is the specific class of pattern headed by a symbol of the signature, which are the one relevant to define rewriting.

Let us first finish to define what a rewriting rule is:

Definition 4.1.3 (Rewriting rule)**.** *A* rewriting rule *is an ordered pair, denoted $f\,\vec{l_i} \hookrightarrow r$, where*

- *the left-hand side $f\,\vec{l_i}$ is a pattern,*
- *and the right-hand side r is a term, whose free variables also occur in the left-hand side.*

A rewriting rule must be thought as representing a universally quantified scheme of equalities. For instance, if $\times$ is a symbol of the signature (applied with infix notation for readability purpose), then the rule $0 \times x \hookrightarrow 0$, denotes the proposition $\forall x, 0 \times x = 0$.

Of course, one would in particular expect that $2 \times 0 = 0$, but also that $1+(2\times 0) = 1+0$. This indicates that one wants to define the rewriting relation $\rightsquigarrow$ as the smallest relation containing the rules (denoted $\hookrightarrow$) and stable by substitution and context.

The rewriting relation is simply the smallest relation containing the rewriting rules and stable by substitution and contextual closure (Definition 3.3.8)

Definition 4.1.4 (Rewriting relation)**.** *A term t* head-reduces *to u, denoted $t \overset{\varepsilon}{\rightsquigarrow} u$, if there is a rule $f\,\vec{l} \hookrightarrow r$, and a substitution σ such that $t = \left(f\,\vec{l}\right)\sigma$ and $u = r\sigma$.*

The rewriting relation $\rightsquigarrow$ *is then defined as the union of $\rightsquigarrow_\beta$ and the contextual closure of $\overset{\varepsilon}{\rightsquigarrow}$.*

If a notion of position in terms has been defined, $t \rightsquigarrow u$ if there is a position p, a substitution σ and a rule $f\,\vec{l} \hookrightarrow r$ (or the β-rule $(\lambda\,(x : A)\,.t)\; u \hookrightarrow t\,[u/_x]$) such that $t|_p = \left(f\,\vec{l}\right)\sigma$ and $u = t[r\sigma]_p$.

In this thesis, we are mostly interested in strongly normalizing terms, which are terms from which start no infinite reduction sequence.

Definition 4.1.5 (Strongly Normalizing Term)**.** *A term t is said* strongly normalizing *if there are no infinite sequence $(t_i)_{i\in\mathbb{N}}$ with $t_0 = t$ and for all i, $t_i \rightsquigarrow t_{i+1}$.*

SN *is the set of strongly normalizing terms.*

A term which cannot be computed more is said in *normal form*.

Definition 4.1.6 (Term in Normal Form)**.** *If $t \in \Lambda$ is said in* normal form *if for all $v \in \Lambda$, $u \not\rightsquigarrow v$.*

The set of terms in normal form is denoted NF*.*

Furthermore, whenever all the computations starting from a term ends on the same result, it is called its *normal form*.

Definition 4.1.7 (Normal Form of a term)**.** *Given a term $t \in$ SN, if there exists a $u \in$ NF, such that for all sequences $t_0, \ldots, t_n$, such that $t_0 = t$, for all $i < n$, $t_i \rightsquigarrow t_{i+1}$ and $t_n \in$ NF, one has $t_n = u$, then u is the* normal form of *t, and we write $t{\Downarrow}= u$.*

4.1.3 Conversion

By construction, the rewriting relation is oriented. It is quite natural, since it aims at computing terms. However, the aim of this chapter was to enrich the conversion rule of PTS, and this rule states that if A and B are convertible types, inhabiting A and inhabiting B are equivalent. So one would like to define a more symmetrical relation than just reduction.

To do so, there exists two alternatives, either considering that two terms A and B are equivalent if performing computation steps of A (potentially zero) and of B leads at some point to the same term. This solution present the advantage to only require reduction (never expansion) and to be a decidable relation if $\rightsquigarrow$ is terminating and finitely branching.

Definition 4.1.8 (Joinability)**.** *If t and u are terms, $t \downarrow u$ if and only if there is a term w such that $t \rightsquigarrow^* w$ and $u \rightsquigarrow^* w$.*

However, this relation is not an equivalence relation, since one could imagine that a term t reduces both to u and v, with u and v two non joinable terms.

The property entailing such a situation cannot happen is called the *confluence property*.

Definition 4.1.9 (Confluent Relation)**.** *A binary relation R is said confluent if for all t, for all u and v such that tR^*u and tR^*v, there is a w such that uR^*w and vR^*w.*

But, in general, one could define the convertibility relation as the reflexive symmetric and transitive closure of $\rightsquigarrow$, which is denoted by $\leftrightsquigarrow^*$.

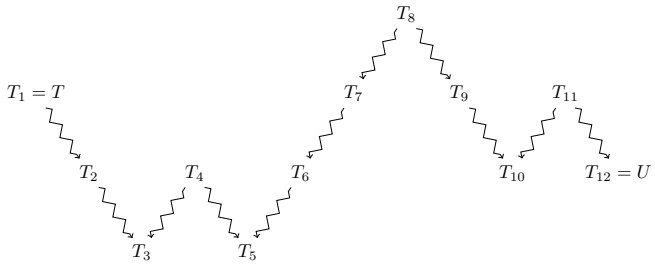
Definition 4.1.10 (Convertibility)**.** *If t and u are terms, $t \leftrightsquigarrow^* u$ if and only if there is a finite sequence $t = v_0, \ldots, v_n = u$ such that for all $i \in [\![0, n-1]\!]$, $v_i \rightsquigarrow v_{i+1}$ or $v_{i+1} \rightsquigarrow v_i$.*

Remark 4.1.11 (Choosing $\leftrightsquigarrow^*$ or $\downarrow$ for typing)**.** *If $\rightsquigarrow$ is confluent, then $\downarrow$ (Definition 4.1.8) and $\leftrightsquigarrow^*$ (Definition 4.1.10) are the same relations. Especially, Theorem 3.3.14 states that one does not have to chose which relation to use for the rule (conv) in PTS's, since $\downarrow_\beta = \leftrightsquigarrow^*_\beta$. However, now that we added user-declared rewriting rules, which might be non-confluent, to the conversion, it is not equivalent anymore to use $\downarrow$ or $\leftrightsquigarrow^*$ in the conversion rule. Hence, we must chose which of those relations will be used in the type system. Both choices would be possible, however, since computing the set of anti-reducts of a term is undecidable, in implementations, the conversion is always a sub-approximation of $\downarrow$[1], hence, we will present* Rewriting Type Systems *with the*

[1] Only a sub-approximation since for a non-terminating rewriting system, even $\downarrow$ is only semi-decidable.

joinability relation $\downarrow$, even if it is not an equivalence relation. Furthermore, $\lambda\Pi$ modulo rewriting is consistent (in a sense detailed in Section 5.2) only if we use this relation in the conversion rule.

The convertibility relation is a sequence of reductions and anti-reductions, $T \leftrightsquigarrow^* U$ can for instance be:



In this example, one also has $T \downarrow T_4 \downarrow T_8 \downarrow T_1 1 \downarrow U$. Hence, using the joinability relation rather than directly the convertibility one, is quite similar except that the rule (conv) may have to be used several time in a row. With the typing rule presented in the next section, this means that with our choice to use $\downarrow$, one has to give a sort to every peak occurring in the convertibility path. For instance, if T and U where types, such that $\Sigma; \Gamma \vdash a : T$, to obtain that $\Sigma; \Gamma \vdash a : U$, one has not only to find a sort s such that $\Sigma; \Gamma \vdash U : s$, but also sorts s_4, s_8 and s_{11} such that $\Sigma; \Gamma \vdash T_4 : s_4$, $\Sigma; \Gamma \vdash T_8 : s_8$ and $\Sigma; \Gamma \vdash T_{11} : s_{11}$.

4.2 Typing of Rewriting Type Systems

Definition 4.2.1 (Signature)**.** *A* signature *in a Rewriting Type System is a quadruple $(\mathcal{F}, \Theta, s, \mathfrak{R})$, where*

- *$\mathcal{F}$ is the finite set of signature symbols, described in Definition 4.1.1;*
- *$\Theta : \mathcal{F} \to \Lambda$ is the* typing map*, a function associating a type to every symbol of the signature;*
- *$s : \mathcal{F} \to \mathcal{S}$ is the function associating to every symbol the sort that its type inhabits;*
- *$\mathfrak{R}$ is the set of rewriting rules defining the computational behavior of the symbols of the signature.*

Now, one can enrich the typing rules of the PTS of specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ (see Definition 3.4.1) with the typing of the symbols of the signature and the enrichment of the convertibility with the joinability relation induced by user-defined rewriting rules:

Definition 4.2.2 (Typing Rules of RTS)**.** *Let $\Sigma = (\mathcal{F}, \Theta, s, \mathfrak{R})$ be the signature of a Rewriting Type System.*

$$\textit{(ax)} \qquad \frac{}{\Sigma; [] \vdash s_1 : s_2}\ (s_1, s_2) \in \mathcal{A}$$

$$\textit{(var)} \qquad \frac{\Sigma; \Gamma \vdash A : s}{\Sigma; \Gamma, x : A \vdash x : A}\ x \notin \mathrm{dom}(\Gamma)$$

$$\textit{(weak)} \qquad \frac{\Sigma; \Gamma \vdash A : s \quad \Sigma; \Gamma \vdash b : B}{\Sigma; \Gamma, x : A \vdash b : B}\ x \notin \mathrm{dom}(\Gamma)$$

$$\textit{(prod)} \qquad \frac{\Sigma; \Gamma \vdash A : s_1 \quad \Sigma; \Gamma, x : A \vdash B : s_2}{\Sigma; \Gamma \vdash (x : A) \Rightarrow B : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$(abs)\quad \frac{\Sigma;\Gamma, x : A \vdash b : B \quad \Sigma;\Gamma \vdash (x : A) \Rightarrow B : s}{\Sigma;\Gamma \vdash \lambda x : A.b : (x : A) \Rightarrow B}$$

$$(app)\quad \frac{\Sigma;\Gamma \vdash t : (x : A) \Rightarrow B \quad \Sigma;\Gamma \vdash a : A}{\Sigma;\Gamma \vdash t\, a : B\,[a/_x]}$$

$$(conv)\quad \frac{\Sigma;\Gamma \vdash a : A \quad \Sigma;\Gamma \vdash B : s}{\Sigma;\Gamma \vdash a : B}\ A \downarrow B$$

$$(sig)\quad \frac{\Sigma;[] \vdash \Theta(f) : s(f)}{\Sigma;[] \vdash f : \Theta(f)}\ f \in \mathcal{F}$$

One must note here that the signature is supposed given, so there are no possibility to enrich it by declaring new symbols or rewriting rules. Furthermore, checks are performed each time a symbol in $\mathcal{F}$ is used, but no checks are performed on the rewriting rules. All the properties of the conversion expected to ensure good behaviour of the type system (subject reduction, decidability of typing...), must be checked externally and are not provided by the type system. Whenever the signature is obvious from the context, it will be omitted in the typing judgement.

Chapter 5

λΠ-Calculus Modulo Rewriting

After all those preliminaries, it is time to focus on the proof system which will be used in thesis. It is the λΠ-calculus modulo rewriting, which is the Rewriting Type System which enriches λΠ (see Example 3.4.2) with higher-order rewriting.

This type system is the one of the type-checker Dedukti [Ded20], originally developed by M. Boespflug [BCH12]. Its current version was written in OCaml by R. Saillard [Sai15], with several enhancements by R. Cauderlier, F. Thiré, G. Férey and myself. The third version of it should be released soon, it has mainly been developed by R. Lepigre, F. Blanqui and G. Hondet.

The main purpose of Dedukti is to be a *logical framework*, meaning that it allows the user to encode the logic she wants to use, and then to use this encoded logic to do proofs.

There are multiple interests to encode logics and proofs in a logical framework: first of all, since λΠ-calculus modulo rewriting is just a relatively simple and well-understood logic, λΠ, enriched with a few symbols and rewriting rules, encoding another a logic L in it helps to understand L. The specificities of the logic will be easily identified, since they correspond to a small set of unusual rewriting rules.

Furthermore, it eases the comparison between logics, since in most of the cases similar "features" are encoded using similar rules, the intersection of the set of rules encoding two logics tells us what are the common points between them, whereas the symmetric difference enlights the differences.

Last, but not least, encoding several logics in the same framework makes it easier to build bridges between them, and one can hope to translate automatically proofs from one logic to another. Of course, since there exist logics with different expressive power or incompatible features, it is not always possible to translate a proof from one logic to another. Hence, necessarily some translations are partial.

Exploiting the pluripotency of this logical framework to do translations between various logics is the main goal of the development team of Dedukti. F. Thiré managed to translate the arithmetic library of Matita to many proof assistants (Coq, PVS, Lean, OpenTheory, Matita) [Thi18].

5.1 Specificities of the λΠ modulo rewriting

5.1.1 Clear Distinction Between Types and Terms

As already mentionned, λΠ modulo rewriting is the *Rewriting Type System* built above Edinburgh LF [HHP93], also called λΠ, introduced in Example 3.4.2. Its typing rules are the

of Rewriting Type Systems, presented in Section 4.2, with $\mathcal{S} = \{\star, \Box\}$, $\mathcal{A} = \{\star : \Box\}$ and $\mathcal{R} = \{(\star, \star, \star), (\star, \Box, \Box)\}$.

It only has two sorts $\star$ and $\Box$, with an axiom stating that $\star : \Box$. More importantly, since it only features a two rules to construct arrows, the one of simply-typed λ-calculus $(\star, \star, \star)$ and $(\star, \Box, \Box)$, which enables dependent types. For instance, it allows to state that if $\mathbb{N} : \star$, then $\mathbb{N} \Rightarrow \star : \Box$, allowing to declare the type of vectors indexed by their length $\texttt{Vec} : \mathbb{N} \Rightarrow \star$. However, it does not feature other arrows, like polymorphism, leading to a very simple stratification of all typable terms in 3 distinct layers:

- The terms which live in $\Box$, called *kinds*. In this layers, one can cite, of course $\star$, but also $\mathbb{N} \Rightarrow \star$ or $(n : \mathbb{N}) \Rightarrow \texttt{Vec}\, n \Rightarrow \star$.

  All terms of this layer are of the form $(x_1 : T_1) \Rightarrow \ldots \Rightarrow (x_n : T_n) \Rightarrow \star$, with the T_i's living in $\star$.

- The terms which live in a kind, called *type families*, or simply *types* for those which inhabit $\star$. In this layer live the inhabitants of $\star$, like $\mathbb{N}$, $\texttt{Vec}\, n$ or $(n : \mathbb{N}) \Rightarrow \texttt{Vec} n$, but also inhabitants of the other type families like $\texttt{Vec}$ which lives in $\mathbb{N} \Rightarrow \star$.

  The shape of type families are more complex than the one of kinds, since it contain, symbols declared by the user, potentially applied to objects and λ-abstractions, also potentially applied to objects. Since arrows can only inhabit sorts, the only kind inhabited by arrows is $\star$: types can be of the form $(x_1 : T_1) \Rightarrow \ldots \Rightarrow (x_n : T_n) \Rightarrow U$, where the T_i's and U are themselves types.

- The terms which lives in a type, called *objects*. In this layer live the inhabitants of $\mathbb{N}$, like 0, $(\lambda\,(x : \mathbb{N})\,.x)\ 0$, but also the inhabitants of $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$, like $+$ or $\lambda\,(x : \mathbb{N})\,.\lambda\,(y : \mathbb{N})\,.x$.

  Objects can be of various shapes, since it can be variable, symbol declared by the user, λ-abstraction or application of objects.

This stratification gives information on the sortability of the type in a typing judgement:

Lemma 5.1.1 (Stratification lemma [Sai15, Lemma 2.6.10])**.** *If $\Gamma \vdash t : A$ then we are in one of the following cases:*

- *t is an object, A a type family and $\Gamma \vdash A : \star$,*
- *t is an type family, A a kind and $\Gamma \vdash A : \Box$,*
- *t is an kind and $A = \Box$.*

5.1.2 Constructors

Contrary to many PTS, the distinction between those layers is quite rigid in λΠ, there are no variable representing a type family, and it is impossible for a type family to be an argument of an application.

This distinction leads to split the symbols of the signature in two sets, depending if it is a type family or an object. This distinction is made by the function s, which states in which sort the type of the symbol lives: objects live in types, which themselves live in $\star$, whereas type families live in kinds, which lives in $\Box$.

Definition 5.1.2 (Objects and Type Families of the Signature)**.** *We define $\mathcal{F}_\mathcal{T} = \{f \in \mathcal{F} \mid s(f) = \Box\}$ and $\mathcal{F}_o = \{f \in \mathcal{F} \mid s(f) = \star\}$.*

In the signature, all the symbols come with a type (given by the function Θ). This type is used to define the arity of a symbol.

Definition 5.1.3 (Arity)**.** *Given a symbol $f \in \mathcal{F}$, if $\Theta(f) = (x_1 : A_1) \Rightarrow \ldots \Rightarrow (x_n : A_n) \Rightarrow B$, where B is not a product, then the integer n is called the arity of f.*

We denote this $\mathrm{ar}(f) = n$.

It must be noted, that for type families the arity could not have been defined otherwise, since the type is necessarily of the shape $(x_1 : T_1) \Rightarrow \ldots \Rightarrow (x_n : T_n) \Rightarrow \star$, with the T_i's living in $\star$. But for objects, the notion of arity is more fuzzy, since with dependent types and rewriting, it can happen that a symbol f is of type $(x : A) \Rightarrow B$ where B is not an arrow (meaning that $\mathrm{ar}(f) = 1$, but such that there exists terms t and u such that $f\,t\,u$ is well-typed. This is because, even if B is not an arrow, when instantiating x with t, it triggers reduction and generates a new arrow.

Definition 5.1.4 (Type constants)**.**

$$\mathcal{C}_{\mathcal{T}} = \{C \in \mathcal{F}_{\mathcal{T}} \mid C \textit{ is not the head of any rule}\}$$

Here we split again $\mathcal{F}_{\mathcal{T}}$ in two sets, those which are defined by rewriting rules, and those which are not (the constructors $\mathcal{C}_{\mathcal{T}}$).

Definition 5.1.5 (Constructors)**.**

$$\mathcal{C}_o = \left\{ f \in \mathcal{F}_o \;\middle|\; \tau(f) = \overrightarrow{(x : T)} \Rightarrow C\,t_1 \ldots t_{\mathrm{ar}(C)} \textit{ with } C \in \mathcal{C}_{\mathcal{T}} \right\}.$$

Remark 5.1.6. *The constraint on the totality of the application of C is not a new constraint. Indeed, if C is partially applied it does not live in a sort, hence the product cannot be typed. However, there is a real restriction here, rewritable types do not have constructors.*

The signatures in the λΠ-calculus modulo rewriting are presented using a syntax similar to the one of Dedukti. It contains all the elements of $\mathcal{F}$, introduced by the keyword `symbol`. The function Θ is given, since all the symbols introduced come with a type, and the rewriting rules are shown explicitly, headed by a context, listing the free variables occurring in the rule. There are two arrows, the blue double arrow ⇒ is used for the products in types, whereas the green hooked arrow ↪ is used to declare rewriting rules.

Example 5.1.7 (A Signature in λΠ modulo rewriting)**.** *One can have type-level rewriting rules (*`El` *converts datatype codes into types of λΠ modulo rewriting).*

```
symbol Set : ⋆.
symbol arrow : Set ⇒ Set ⇒ Set.
symbol El : Set ⇒ ⋆.
[a,b] El (arrow a b) ↪ El a ⇒ El b.
```

One can declare functions on regular types, like natural numbers or booleans.

```
symbol Bool : ⋆.
symbol true  : Bool.
symbol false : Bool.
symbol not : Bool ⇒ Bool.
[] not true  ↪ false.
[] not false ↪ true.
```

```
symbol ℕ : ⋆.
symbol 0 : ℕ.
symbol s : ℕ ⇒ ℕ.
symbol infix + : ℕ ⇒ ℕ ⇒ ℕ.
[y]   0     + y ↪ y.
[x,y] (s x) + y ↪ s (x + y).
```

Rewriting rules can overlap, or match on defined symbol, for instance it is possible to define addition both by matching on its left and its right argument, and to declare that it is associative.

```
[x]     x + 0       ↪ x.
[x,y]   x + (s y)   ↪ s (x + y).
[x,y,z] x + (y + z) ↪ (x + y) + z.
```

One also has dependent types (`Vec` *is the type of lists parameterized with their length),*

```
symbol Vec : ℕ ⇒ ⋆.
symbol nil  : Vec 0.
symbol cons : (n : ℕ) ⇒ A ⇒ Vec n ⇒ Vec (s n).
symbol append : (m : ℕ) ⇒ Vec m ⇒ (n : ℕ) ⇒ Vec n ⇒ Vec (m + n).
[l]           append 0     nil           n l  ↪ l.
[m,a,l1,n,l2] append (s m) (cons m a l1) n l2 ↪
                                    cons (m + n) a (append m l1 n l2).
```

Thanks to the associativity of the addition, it is possible to have a rule stating that `append` *is also associative.*

```
[m,l1,n,l2,p,l3] append m l1 (n + p) (append n l2 p l3) ↪
                                  append (m + n) (append m l1 n l2) p l3.
```

It is also possible to have higher-order variables.

```
symbol map : (A ⇒ A) ⇒ (n : ℕ) ⇒ Vec n ⇒ Vec n.
[f]       map f 0     nil           ↪ nil.
[f,n,a,l] map f (s n) (cons n a l) ↪ cons n (f a) (map f n l).
```

Dependent types can make the code heavy. Here `filter` *is a function filtering elements out of a list along a boolean function* `f`*. To define it, we use not only a function* `fil_aux` *to match on the application of* `f` *to the head of the list, but also a function* `len_fil`*, to compute the length of the result of the filtering, which is required to type* `filter`*.*

```
symbol len_fil: (A ⇒ Bool) ⇒ (p : ℕ) ⇒ Vec p ⇒ ℕ.
symbol len_fil_aux: Bool ⇒ (A ⇒ Bool) ⇒ (p : ℕ) ⇒ Vec p ⇒ ℕ.
[f]       len_fil f 0     nil           ↪ 0.
[f,p,a,l] len_fil f (s p) (cons p a l) ↪ len_fil_aux (f x) f p l.
[f,p,l] len_fil_aux true  f p l ↪ s (len_fil f p l).
[f,p,l] len_fil_aux false f p l ↪ len_fil f p l.

symbol filter : (f : (A ⇒ Bool)) ⇒ (p : ℕ) ⇒ (l : Vec p)
                                                  ⇒ Vec (len_fil f p l).
symbol fil_aux : (b : Bool) ⇒ (f : (A ⇒ Bool)) ⇒ (p : ℕ) ⇒ A
                            ⇒ (l : Vec p) ⇒ Vec (len_fil_aux b f p l).
[]        filter f 0     nil           ↪ nil.
[f,p,a,l] filter f (s p) (cons p a l) ↪ fil_aux (f x) f p a l.
[f,p,a,l] fil_aux false f p a l ↪ filter f p l.
[f,p,a,l] fil_aux true  f p a l ↪ cons (len_fil f p l) a (filter f p l).
```

Once again, it is possible to match on the defined function `append` *to define* `filter`*.*

```
[f,p,l1,q,l2] len_fil f (p + q) (append p l1 q l2) ↪
                                  (len_fil f p l1) + (len_fil f q l2).
[f,p,l1,q,l2] filter  f (p + q) (append p l1 q l2) ↪
                                append (len_fil f p l1) (filter f p l1)
                                       (len_fil f q l2) (filter f q l2).
```

Note that this example cannot be represented in Coq or Agda because of the rules using matching on defined symbols like `+` or `append`. And its termination can be handled neither by [Wah07] nor by [Bla05] because the system is not orthogonal and has no strict decrease in every recursive call. It can however be handled by the termination criterion presented in Theorem 6.9.1.

To summarize, in $\lambda\Pi$-modulo rewriting, we have four sets of symbols of the signature:

- The type families not defined by rewriting: $\mathcal{C}_\mathcal{T}$, $\mathbb{N}$, `Bool` and `Vec` in the example,
- The type families defined (partially) by rewriting: $\mathcal{F}_\mathcal{T} \setminus \mathcal{C}_\mathcal{T}$, `El` in the example,
- The objects which construct an element of $\mathcal{C}_\mathcal{T}$: $\mathcal{C}_o$, it includes 0, `s`, `nil`, `cons`, but also more complex functions of the example, like `+` and `append`,
- The objects which construct an element of a type which is not headed by a $\mathcal{C}_\mathcal{T}$: $\mathcal{F}_o \setminus \mathcal{C}_o$.

Definition 5.1.8 (Neutral terms)**.** *A term is* neutral *if it is of the form:*

$$\begin{array}{rll} B & ::= (\lambda\,(x : A)\,.U)\,T_0 \dots T_m \mid F\,t_1 \dots t_n & \text{where } m \geqslant 0,\ F \in \mathcal{F}_\mathcal{T} \setminus \mathcal{C}_\mathcal{T} \text{ and } n \geqslant \mathrm{ar}(F) \\ b & ::= (\lambda\,(x : A)\,.u)\,t_0 \dots t_m \mid x\,t_1 \dots t_m \mid f\,t_1 \dots t_n & \text{where } m \geqslant 0,\ f \in \mathcal{F}_o \setminus \mathcal{C}_o \text{ and } n \geqslant \mathrm{ar}(f) \end{array}$$

depending on whether it is a family or an object. We denote by $\mathcal{N}_\mathcal{T}$ the set of neutral families *and $\mathcal{N}_o$ the one of* neutral objects*.*

Neutral terms are terms such that applying it to any term does not trigger new computation, meaning that t is neutral if for all u, the redices in $t\,u$ are either redices of t or redices of u. For instance, all variables are neutral, β-redices too, but $\lambda\,(x : \mathbb{N})\,.x$ is not. Regarding functions of the signature, which are not constructors, d is not neutral, since its arity is 1, but $d\,1$ is neutral.

The reason why constructors are not neutral, even when fully applied, it is due to the definition of the interpretation of types (Definition 6.2.1), useful for the normalization criterion introduced in the next chapter.

Lemma 5.1.9 (Shape of inhabited types)**.** *Let A be a type family. If there are a Γ and a t such that $\Gamma \vdash t : A$, then A is a product, a neutral type or a fully applied type constructor.*

Proof. Since abstractions and partially applied signature symbols inhabit products and no product is convertible to a sort, such families cannot be inhabited. □

5.2 Consistency

Until now, we only defined in this thesis schemes of logics (PTS and RTS), which include both consistent and inconsistent instances. For instance, regarding PTS, the calculus of constructions[1] is consistent [CH86], whereas the system U^{-2} is not [Gir72, Coq86].

[1] The sorts of the calculus of constructions are $\mathcal{S} = \{\star, \Box\}$, the axioms are $\mathcal{A} = \{\star : \Box\}$ and the rules are $\mathcal{R} = \{(\star,\star,\star), (\star,\Box,\Box), (\Box,\star,\star), (\Box,\Box,\Box)\}$

[2] The sorts of system U^- are $\mathcal{S} = \{\star, \Box, \triangle\}$, the axioms are $\mathcal{A} = \{\star : \Box, \Box : \triangle\}$ and the rules are $\mathcal{R} = \{(\star,\star,\star), (\Box,\star,\star), (\Box,\Box,\Box), (\triangle,\Box,\Box)\}$

When we are dealing with a specific logic, the first question which arises is consistency of this logic.

Since a logical framework is designed for encoding, there are two logics involved in the process, the "host logic" and the encoded one.

Let us prove first the consistency of the "host logic", here the λΠ-calculus modulo rewriting.

But first, one has to wonder what does it mean for a logic to be consistent? The natural answer is that one cannot prove false statements in it. But in PTS or RTS, there are no notion of falsity, hence one has to modify a little this definition.

The "ex falso quodlibet" principle states that from falsehood, one can deduce anything. Introduced in the 12^{th} century by William de Soissons, this principle is quite natural, and can be proved in one line. Assume that two contradictory statements A and $\neg A$ hold, then, since A is true, so is $A \vee B$, but since A is false, one can conclude that B no matter what B is, it could state that "Earth is flat", "Unicorns exist" or be your favorite mathematical open problem.

Equipped with this principle, being unable to prove false in a logic is the same thing as saying that there are some propositions unprovable in the logic. It is this version of the notion of consistency that will be proved here.

There are already several works aiming at proving the consistency of a Rewriting Type System, for instance [WCC08] or [Bla05, Thm 4.1]. However, those two papers are interested in enrichment of the Calculus of Constructions (CoC) with rewriting rules. Contrary to the λΠ-calculus, CoC allows the types to be an argument of a application. Because of this, the type $(T : \star) \Rightarrow T$ is valid, hence it is possible for the user to declare in CoC a symbol ε of this type[3]. Since, without rewriting rules, declaring a symbol of this type makes the logic inconsistent (any type T is inhabited by $\varepsilon\, T$), the criteria presented in [WCC08] requires all the symbols introduced by the user to be completely erased by the computations, and the one of [Bla05] restricts the types of the symbols which can be introduced. In [Sel98], Seldin shows the logical consistency of "strongly consistent" contexts, in the Calculus of Constructions, by syntactical means. It is not the case with the λΠ-calculus modulo rewriting, which is consistent without any restrictions on the symbols and rules declared by the user.

To prove this consistency, one introduces a new type $\bot$ at the end of the context, and the goal is to show that it is inhabitable. But, since some typing remove or add elements in the context, the property of being at the context is not stable while browsing the context, meaning that some $\bot$ can appear in the type of some variables of the context, hence it also can appear in the left of dependent arrows. Hence, one cannot prove that $\bot$ is uninhabited without proving that a larger family of terms are.

Definition 5.2.1 (Hereditarily Uninhabitable Types)**.** *Given $\bot \in \mathcal{V}$, $P_\bot$ and $N_\bot$ are sets of terms defined by:*

$$P_\bot ::= \bot \;\mid\; (y : N_\bot) \Rightarrow P_\bot$$
$$N_\bot ::= (y : P_\bot) \Rightarrow N_\bot \;\mid\; (y : N_\bot) \Rightarrow N_\bot \;\mid\; (y : P_\bot) \Rightarrow P_\bot \;\mid\; T$$

Where $y \in \mathcal{V} \setminus \{\bot\}$ and T is neither $\bot$ nor a product.

The types in $P_\bot$ will all be proved uninhabitable, whereas some of the types in $N_\bot$ are inhabited, even some of them which features $\bot$. For instance $\lambda\,(x : \bot)\,.x$ inhabits $(x : \bot) \Rightarrow \bot$, but since $(x : \bot) \Rightarrow \bot \in N_\bot$, this does not compromise the property we are trying to prove.

First of all, one must note that since the T case in the definition of $N_\bot$ is a "default case", every term is in $P_\bot \cup N_\bot$. One has to prove that those two sets really constitute a partition of the terms. For this, let us prove that no term is both in $P_\bot$ and in $N_\bot$.

[3]The notation ε comes from Hilbert, who uses ε to denote a witness of a proved existential proposition.

Lemma 5.2.2.

$$P_\perp \cap N_\perp = \emptyset$$

Proof. If one assumes that there is a minimal T such that $T \in P_\perp \cap N_\perp$.

- If T is $\perp$, then it is in not in $N_\perp$;
- if T is a product, is domain or is codomain is both in $P_\perp$ and in $N_\perp$. But this contradict the minimality hypothesis;
- If T is neither $\perp$, nor a product, it is not in $P_\perp$.

Hence, such a T cannot exists and $P_\perp \cap N_\perp = \emptyset$. □

One must note that belonging to $P_\perp$ or $N_\perp$ only depends of the position of the $\perp$ symbols in the products.

Furthermore, if $\perp$ is a local variable of type $\star$, those two sets are stable by reduction of well-typed term:

Lemma 5.2.3. *Let $\Sigma_\mathfrak{R}$ be a signature in which $\perp$ does not appear. Let Γ, t and A be such that $\perp : \star$ appears in Γ and $\Gamma \vdash t : A$. Then for all u such that $t \rightsquigarrow u$,*

- *if $t \in N_\perp$ then $u \in N_\perp$,*
- *and if $t \in P_\perp$ then $u \in P_\perp$.*

Proof. We will prove this simultaneously by induction on the structure of t.

- If $t = \perp$, then t does not reduce, since $\perp$ does not occur in $\Sigma_\mathfrak{R}$.
- If t is a product, it is a direct consequence of the induction hypothesis.
- If t is neither a product nor $\perp$, then t is in $N_\perp$. Since $\perp$ does not occur in $\Sigma_\mathfrak{R}$, the only possibility for t to reduce to $\perp$ is to have a rule of the shape $f\ \overrightarrow{l} \longhookrightarrow x$, and t of the form $f\ \overrightarrow{u}$ with $u_i = l_i\sigma$ for all i, and $\sigma(x) = \perp$. But in $\lambda\Pi$-calculus modulo rewriting, a term of type $\star$ cannot be the argument of a well-typed application. Hence, the u_i's cannot contain $\perp$.

  For the same reason, t cannot reduce to a product containing $\perp$, and all the terms in $P_\perp$ are of the form $\overrightarrow{(x_i : A_i)} \Rightarrow \perp$, so if t reduce to a product it is also a product in $N_\perp$. □

□

The easiest to prove that one cannot inhabits any type, is to introduce a brand new type, and to show that it cannot be inhabited without assuming anything more. But to prove such a theorem by induction, one needs to generalize a little the hypothesis and to prove :

Proposition 5.2.4. *Let $\Sigma_\mathfrak{R}$ be a signature in which $\perp$ does not appear. For all Γ, for all A_i's, B_j's in $N_\perp$ for all t, one cannot derive*

$$\Gamma, \perp : \star, \overrightarrow{x_i : A_i} \vdash t : \overrightarrow{(y_j : B_j)} \Rightarrow \perp$$

where the variables x_i's and y_j's are all different of $\perp$.

Proof. Let us assume that there is such a sequent which is derivable. Then there is a proof of such a sequent which is minimal, in the sense that no strict sub-tree of this derivation is a proof of a sequent of the targeted shape.

By case on the last rule used in the derivation, one will end up with a contradiction:

(ax) and (prod) The type in the judgement we are trying to derive is not a sort, hence it is impossible.

(var) Since $\bot$ is declared after Γ, $\bot$ cannot occur free in the types declared in Γ. But $\bot$ is a free variable in $\overrightarrow{(y_j : B_j)} \Rightarrow \bot$. Hence $\overrightarrow{(y_j : B_j)} \Rightarrow \bot$ must be one of the A_i's.

Since all the B_j's are in $N_\bot$, $\overrightarrow{(y_j : B_j)} \Rightarrow \bot$ is in $P_\bot$. So $\overrightarrow{(y_j : B_j)} \Rightarrow \bot$ cannot be one of the A_i's, thanks to Lemma 5.2.2.

(sig) Since $\bot$ does not occur free in any type in $\Sigma_{\mathfrak{R}}$, $\overrightarrow{(y_j : B_j)} \Rightarrow \bot$ cannot be the conclusion of a (sig) rule.

(app) Then, there are u, v, C and $(z : C) \Rightarrow D$ such that $\Gamma, \bot : \star, \overrightarrow{x_i : A_i} \vdash u : (z : C) \Rightarrow D$ and $\Gamma, \bot : \star, \overrightarrow{x_i : A_i} \vdash v : C$ with $D\left[{}^{v}/_{z}\right] = \overrightarrow{(y_j : B_j)} \Rightarrow \bot$.

Here, one must note that $D\left[{}^{v}/_{z}\right] \in P_\bot$. And in λΠ-calculus modulo rewriting, a term of type $\star$ cannot be the argument of an application. Hence, v cannot contain $\bot$ (but in the annotation of λ-abstractions). So D is also necessarily in $P_\bot$.

If $C \in P_\bot$, then the hypothesis $\Gamma, \bot : \star, \overrightarrow{x_i : A_i} \vdash v : C$ has the forbidden shape, contradicting the minimality hypothesis.

Otherwise, it is $(z : C) \Rightarrow D$ which would contradict the minimality of the conclusion of the proof tree.

(abs) If $t = \lambda\,(y_1 : B_1)\,.u$, then one of the hypothesis is $\Gamma : \star, \overrightarrow{x_i : A_i}, y_1 : B_1 \vdash u : (y_2 : B_2) \Rightarrow \ldots \Rightarrow (y_n : B_n) \Rightarrow \bot$, and it also contradicts the minimality of the chosen proof tree.

(weak) One of the hypothesis of the weakening as the same shape as its conclusion, hence having weakening as last rule would contradict minimality.

(conv) Assume that there is a U such that $\Gamma, \bot : \star, \overrightarrow{x_i : A_i} \vdash t : U$ and $\overrightarrow{(y_j : B_j)} \Rightarrow \bot \downarrow U$. Since $\overrightarrow{(y_j : B_j)} \Rightarrow \bot \in P_\bot$, by Lemma 5.2.3, all its reducts are in $P_\bot$. Since U is well-typed and as a reduct in $P_\bot$, it is also in $P_\bot$.

Hence, $\Gamma, \bot : \star, \overrightarrow{x_i : A_i} \vdash t : U$ is a sequent of the shape we are interesting in, occurring earlier in the proof tree, contradicting minimality of the chosen one. □

Corollary 5.2.5 (λΠ-modulo rewriting is consistent)**.** *Let $\Sigma_{\mathfrak{R}}$ be a signature in which $\bot$ does not appear. For all context Γ in which $\bot$ does not occur and all t, one cannot derive $\Gamma, \bot : \star \vdash t : \bot$.*

But this consistency of the "host logic" that is the λΠ-calculus modulo rewriting does not mean that all the logics that one can encode into it are also consistent. For instance, in Section 5.3 one will see that any functional finitely sorted PTS can be encoded in the λΠ-calculus modulo rewriting, and among this family of encodable logics stand the system U^- which is inconsistent.

Let us give another example of an encoding of an inconsistent logic. For this, one needs a first intuition of what a logic encoding looks like in λΠ modulo rewriting. In general, to encode a logic in λΠ-modulo theory, one first declare symbols to represent the family of types of the "source logic", and then declare a lifting function to transform this representation of types of the "source

logic" into types of the "host logic". Then the simplest inconsistent logic one could imagine to encode in λΠ modulo rewriting, is the logic with a symbol ε "à la Hilbert" which associates to each type a canonical inhabitant of it, meaning then that every types are inhabited.

The signature to encode such a logic is : Σ_i = type : $\star$, Lift : type$\Rightarrow\star$, ε : $(u : \text{type}) \Rightarrow$ Lift u.

Of course, in such an encoding, one can easily prove that for a newly introduced variable $\bot$ of type type, Lift $\bot$ is inhabited[4].

The exhaustive proof tree is:

$$\frac{\Pi_1 \qquad \Pi_2}{\bot : \text{type} \vdash \varepsilon\, \bot : \texttt{Lift}\, \bot}\ (\text{app})$$

where π is the tree

$$\cfrac{\cfrac{}{\vdash \star : \square}\ (\text{ax})}{\vdash \text{type} : \star}\ (\text{sig})$$

in

$$\Pi_1 = \cfrac{\pi \qquad \cfrac{\cfrac{\pi \qquad \cfrac{\cfrac{}{\vdash \star : \square}\ (\text{ax})}{\vdash \text{type} \Rightarrow \star : \square}\ (\text{prod})}{\vdash \texttt{Lift} : \text{type} \Rightarrow \star}\ (\text{sig})}{u : \text{type} \vdash \texttt{Lift} : \text{type} \Rightarrow \star}\ (\text{weak}) \qquad \cfrac{\pi}{u : \text{type} \vdash u : \text{type}}\ (\text{var})}{u : \text{type} \vdash \texttt{Lift}\, u : \star}\ (\text{app})}{\vdash (u : \text{type}) \Rightarrow \texttt{Lift}\, u : \star}\ (\text{prod})}{\bot : \text{type} \vdash \varepsilon : (u : \text{type}) \Rightarrow \texttt{Lift}\, u}\ (\text{sig})\ (\text{weak})$$

and

$$\Pi_2 = \frac{\pi}{\bot : \text{type} \vdash \bot : \text{type}}\ (\text{var})$$

5.3 Encoding Pure Type Systems in λΠ-modulo rewriting

In 2007, Cousineau and Dowek [CD07] proposed an encoding of any functional PTS in λΠ modulo rewriting. Their encoding contained two symbols for each sort, and one symbol for each axiom or rule. However, having an infinite number of symbols and rules is not well-suited for implementations. Hence, to encode *Pure Type Systems* with an infinite number of sorts, one prefers to have a type Sort for sorts and only one symbol for products [Ass15]. For *full Pure Type Systems*, this extension is quite straightforward. The general encoding of full PTS is:

First the PTS specification: a type of sorts and two functions for $\mathcal{A}$ and $\mathcal{R}$.

```
constant 𝒮 : ⋆.
symbol axiom : 𝒮 ⇒ 𝒮.      symbol rule : 𝒮 ⇒ 𝒮 ⇒ 𝒮.
```

For each sort `s`, a type `Univ s` containing the codes of its elements. Indeed, since the λΠ-calculus, does not allow to quantify over types, one needs to declare the type of the logic we are encoding, not directly as a type, but as a code, which can be decoded to a type using rewriting rules.

```
constant Univ : (s : 𝒮) ⇒ ⋆.
```

[4] $\bot$ is not inhabitable in this context, since it lives in type which is not a sort of the λΠ calculus modulo.

Then a symbol to decode the elements of `Univ s` as type of λΠ-modulo rewriting.

```
symbol Lift : (s : 𝒮) ⇒ Univ s ⇒ ⋆.
```

The encoding of sorts and the rewriting rule to decode it. (Simulates the rule (ax) of a PTS).

```
constant code : (s : 𝒮) ⇒ Univ (axiom s).
[s] Lift _ (code s) ↪ Univ s.
```

The encoding of products and its decoding rewriting rule. (Simulates the rule $(prod)$ of a PTS).

```
constant prod : (s1 : 𝒮) ⇒ (s2 : 𝒮) ⇒
       (A : Univ s1) ⇒ (Lift s1 A ⇒ Univ s2) ⇒ Univ (rule s1 s2).
[a,b,A,B] Lift _ (prod a b A B) ↪ (x : Lift a A) ⇒ Lift b (B x).
```

Then the peculiarity of each PTS is reflected in the encoding of the elements of $\mathcal{S}$, and in the implementation of `axiom` and `rule` to encode $\mathcal{A}$ and $\mathcal{R}$ respectively.

In this encoding, contrary to the original one, `prod`, `code` and `Lift` are unique, even if the PTS has many sorts and products.

Chapter 6

Termination Criterion and Dependency Pairs

As already discussed, termination of rewriting systems used in the $\lambda\Pi$-calculus modulo rewriting is crucial, since it participates to the decidability of typing, and in a context of proof-checking, it is a key property. Indeed, it would not be reasonable to claim: "I proved this result, but you have to trust me, since the proof is not checkable". So this chapter introduces a termination criterion for well-typed terms in the $\lambda\Pi$-calculus modulo rewriting.

This result is an extension of the work presented at FSCD 2019 by my advisors (F. Blanqui and O. Hermant) and myself [BGH19], to handle strictly positive inductive types, like Brouwer ordinals introduced in Example 6.1.7, which was not accepted by the 2019 version of the criterion.

The aim is to prove that, under certain conditions detailed later, all the typable terms are strongly normalizing, meaning that there are no infinite sequence of reduction starting from a well-typed term. To do so, in Section 6.2, we define a syntactic model of every type, containing only strongly normalizing terms, and the main part of the chapter is to refine successively conditions on the rewriting system to obtain an adequacy result stating that deriving the inhabitation of a term in a type induces that the terms lives in the interpretation of the type.

Before that, it is necessary in Section 6.1 to define a notion of accessible positions, inspired by what is done in [Bla05]. In general, being in the interpretation of its type is not stable by the subterm relation, and accessible position is a syntactic criterion to identify in which case subterms can be studied without losing membership to the interpretation of their type. This notion is defined before the interpretations, because it is used in the definition of the interpretation of type constructors (Definition 6.2.1).

As explained in Section 6.3, the definition of the types is an adaptation of the reducibility candidates[1] of Tait and Girard [Tai67, GLT88].

The first adequacy theorem is Theorem 6.4.3. However, this theorem states that the adequacy is a consequence of the validity of the typing map Θ, an undecidable property.

Having an undecidable termination criterion is not satisfactory, since the termination of the rewriting system is desired, to recover the decidability of typing. To refine the criterion, in order to obtain a decidable and correct one, the notion of dependency pairs, introduced by Arts and Giesl [AG00], is extended to the higher-order case with dependent types. Several previous works already introduced a notion of dependency pairs for higher-order rewriting [Bla06, KS07, KvR12, FK19], but all those works only consider the simply-typed case. Even if the notion of dependency

[1] Also called "logical relations".

pairs is not mentionned in it, in his thesis, Wahlstedt [Wah07] prove the weak normalization of a dependently typed language, using a well-founded "call relation" for recursive definition. This call relation is analogous to the "instantiated dependency pair" relation, defined in Definition 6.6.3. The only difference between them is that we allow the function symbols to be partially applied in dependency pairs, and in this case add arbitrary terms to complete the application. Thiemann and Giesl already observed the similarity between the call relation in the size-change principle and the dependency pairs in [TG05].

Dependency pairs aim at defining a call relation, which generalizes the notion of recursive call for functions, in order to detect all the potential calls leading to non-termination. If all those calls are proved to be non-dangerous, then the rewriting system is terminating.

To be very precise, the main result is Theorem 6.6.12, which states that the termination of a relation called "call relation" implies the termination of the rewriting. However, Theorem 6.6.12 still has two hypotheses, validity of the rewriting rules and well-foundedness of the call relation. Hence, Section 6.7 introduces a class of valid rewriting systems, the one which are *Accessible Variables Only*, abbreviated *AVO*. And Section 6.8 introduces a criterion to guarantee the call relation is well-founded. Even at the higher-order, dependency pairs evolved into a framework with several processors to prove the well-foundedness of the call relation [FK19]. After Wahlstedt [Wah07], we chose to use the size-change termination criterion [LJBA01].

The final theorem of the chapter (Theorem 6.9.1) summarizes everything. Several examples of usability of the criterion are given in the next chapter (Chapter 7) which presents the termination checker I implemented to have experimental results on the effectiveness of the criterion. Next chapter also contains discussions on the strengths and weaknesses of the criterion and on the perspectives to enhance it.

Of course, termination of the rewriting system is not the only condition required to have the decidability of typing. For instance termination, confluence (also called Church-Rosser property, see Theorem 3.3.14 for the β-reduction) and subject reduction (preservation of typing, see Theorem 3.5.4 for the β-reduction alone) imply decidability of typing.

Often, criteria to prove of those tree properties assume the two others. However, we do not want to generate circularities, hence we will assume a minimal set of hypotheses to obtain our termination criterion. First of all, we do not need to assume subject reduction to obtain the main result: Theorem 6.6.12, but to prove the validity of the rewriting system, we use the AVO criterion in Section 6.7 which requires that all the rewriting rules are well-typed, a property very similar to subject reduction (but more local). Even if preservation of types by the rewriting rules is not a prerequisite in this work, we expect the rules to respect the layers presented in Lemma 5.1.1.

Condition 6.0.1 (All rules preserve sorts)**.** *In all this chapter, we assume that the left-hand and the right-hand sides of rules are either both objects or both type families.*

Similarly, we do not require the full confluence of the rewriting system, but only a local version of it, called local confluence, which specifies that if one step of reduction can be performed in two different ways on the same term, the two one-step reducts in turn reduce to the same term, potentially in several step this time. More formally, the local confluence states that $(\leftsquigarrow\rightsquigarrow) \subseteq (\rightsquigarrow^* {}^*\!\leftsquigarrow)$.

Condition 6.0.2. *In all this chapter, we assume local confluence of the rewriting system.*

6.1 Accessibility

First of all, we should analyze a quite standard example of non-terminating rewriting system: the encoding of the pure λ-calculus.

Example 6.1.1 (Pure λ-calculus)**.**

```
symbol Term  : ⋆.
symbol abstr : (Term ⇒ Term) ⇒ Term.
symbol app   : Term ⇒ Term ⇒ Term.
[f] app (abstr f) ↪ f.
```

Naturally, just like in Example 3.3.11, one can define $\Delta =$ `abstr (λx. app x x):Term` and `app` Δ Δ`:Term` is looping.

This is an example of a well-typed, non-terminating term. We must note that, when ignoring typing, this phenomenon can often be replicated when a variable is applied in the right-hand side of a rewriting rule. For instance, this system, presented in [Kop12],

```
symbol Obj      : ⋆.
symbol ObjList  : ⋆.
symbol FunList  : ⋆.
symbol objCons  : Obj ⇒ ObjList ⇒ ObjList.
symbol funCons  : (Obj ⇒ Obj) ⇒ FunList ⇒ FunList
symbol fmap     : FunList ⇒ Obj ⇒ ObjList.
[f,x,l] fmap (funCons f l) x ↪ objCons (f x) (fmap l x).
```

is terminating, when we take types into account. But if one ignores them, one can construct the term $\Delta =$ `funCons (λx, fmap x x) y` and `fmap` Δ Δ is not terminating. Indeed

$$\begin{aligned}\texttt{fmap}\ \Delta\ \Delta &= \texttt{fmap (funCons (}\lambda\texttt{x, fmap x x) y)}\ \Delta\\ &\rightsquigarrow \texttt{objCons ((}\lambda\texttt{x, fmap x x)}\ \Delta\texttt{) (fmap y}\ \Delta\texttt{)}\\ &\rightsquigarrow_\beta \texttt{objCons (fmap}\ \Delta\ \Delta\texttt{) (fmap y}\ \Delta\texttt{).}\end{aligned}$$

which contains the term one is currently reducing.

Hence, considering typing is necessary when dealing with higher-order pattern variables. Then, let us define a notion of accessible variables, which will characterize the higher-order variables that can appear in the right-hand side of rewriting rules without compromising termination.

Condition 6.1.2 (Order on type symbols)**.** *Let assume given a well-founded total pre-order $\preccurlyeq$ on $\mathcal{C}_\mathcal{T}$.*

We denote $\approx$ the equivalence relation induced by $\preccurlyeq$ and $\prec$ its strict part.

Definition 6.1.3 (Type value)**.**

$$\mathcal{V}al_T = \left\{C\,\vec{t}\;\middle|\; C \in \mathcal{C}_\mathcal{T}, \left|\vec{t}\right| = \mathrm{ar}(C) \text{ and for all } i, t_i \in \mathrm{SN}\right\}$$

Definition 6.1.4 (Frozen type)**.** *For $C \in \mathcal{C}_\mathcal{T}$, we define the following grammars :*

$$\begin{aligned} T_{\preccurlyeq C} &::= (x : U_{\prec C}) \Rightarrow T_{\preccurlyeq C} \;\mid\; C'\,\vec{u} && \text{where } C' \preccurlyeq C \text{ and } |\vec{u}| = \mathrm{ar}(C')\\ U_{\prec C} &::= (x : U_{\prec C}) \Rightarrow U_{\prec C} \;\mid\; C'\,\vec{u} && \text{where } C' \prec C \text{ and } |\vec{u}| = \mathrm{ar}(C') \end{aligned}$$

We denote those sets respectively by $\mathrm{FrozTyp}_{\preccurlyeq C}$ *and* $\mathrm{FrozTyp}_{\prec C}$*.*

Remark 6.1.5 (Stability by substitution)**.** *Since being in* $\mathrm{FrozTyp}_{\preccurlyeq C}$ *and* $\mathrm{FrozTyp}_{\prec C}$ *only depends of the structure of the type, in terms of arrows and type constructors and completely ignores the objects, it is stable by substitution.*

Definition 6.1.6 (Accessible arguments)**.** *For $f \in \mathcal{C}_o$, where $\Theta(f) = \overrightarrow{(x : T)} \Rightarrow C\, t_1 \dots t_{\mathrm{ar}(C)}$, we define*

$$\mathrm{Acc}(f) = \left\{ i \leqslant \mathrm{ar}(f) \;\middle|\; T_i \in \mathrm{FrozTyp}_{\preccurlyeq C} \right\}.$$

When the computation is enriched using inductive definitions, this possibility is often restricted to strictly positive inductive types [Men87], meaning that the types of all the constructors of inductive types are "frozen" (Definition 6.1.4), ensuring them that the interpretation of values is stable by subterm. The definition of accessible position is the analogous of this restriction for functions defined by rewriting.

In the example at the beginning of this chapter (see Example 6.1.1), the first argument of `abstr` is not accessible, since `Term` $\Rightarrow$ `Term` is not in $\mathrm{FrozTyp}_{\preccurlyeq \texttt{Term}}$. If one anticipates a bit on Theorem 6.9.1, this means that a rule like `[f] app (abstr f)` $\hookrightarrow$ `f` is not accepted, because the variable `f` occurs in the right-hand side whereas it is at an inaccessible position in the left-hand side (as first argument of `abstr`).

Anticipating a bit more, accessible positions are the ones for which we are sure (thanks to typing) that the subterm at this position belongs to the interpretation of their type, hence a variable at an accessible position in the left-hand side of a rule can be safely reused in the right-hand side.

However, restricting ourselves to accessible positions does not prevent all the usage of functional variables. For instance, this example is perfectly acceptable:

Example 6.1.7 (Brouwer's ordinals)**.**

```
symbol Nat : ⋆
symbol Ord : ⋆
  symbol 0 : Ord
  symbol s : Ord ⇒ Ord
  symbol lim : (Nat ⇒ Ord) ⇒ Ord

symbol ordrec : X ⇒ (Ord ⇒ X ⇒ X) ⇒ ((Nat ⇒ Ord) ⇒ (Nat ⇒ X) ⇒ X)
                  ⇒ Ord ⇒ X
[x,y,z]   ordrec x y z 0       ↪ x
[x,y,z,o] ordrec x y z (s o)   ↪ y o (ordrec x y z o)
[x,y,z,f] ordrec x y z (lim f) ↪ z f (λ n : Nat.ordrec x y z (f n))
```

In particular, in the last rule, the variable f is of functional type, but it is not an issue, since it occurs as the first argument of `lim` *and if one chooses that* `Nat` $\prec$ `Ord`*, then* `Nat` $\Rightarrow$ `Ord` *is in* $\mathrm{FrozTyp}_{\preccurlyeq \texttt{Ord}}$*, so the first argument of* `lim` *is accessible in such a setting.*

6.2 Interpretations

Our aim in this chapter is to provide sufficient conditions, such that every typable term is strongly normalizing. To do so, we will adapt the interpretation technique proposed by Tait [Tai67] and Girard [GLT88]. An interpretation is a syntactical model, meaning that it is a function associating to each type a set of terms. The aim is to separate the implication "a typable term is strongly normalizing" into two parts:

- If a term is typable, then it is in the interpretation of its type;
- The interpretations of types only contain strongly normalizing terms.

As mentioned earlier, the first result, called adequacy, will be obtained conditionally several times in this chapter (Theorem 6.4.3, Corollary 6.6.13 and Theorem 6.9.1), with conditions being more and more usable while new notions are introduced.

6.2.1 Interpretation of type values

Definition 6.2.1 (Interpretation of type values)**.** *Given $C \in \mathcal{C}_\mathcal{T}$, let $\bar{C} = (C_1, \dots, C_n)$ be the tupled version of the equivalence class of the type constructor C with respect to $\preceq$. By "tupled version", we mean that we assume the existence of a total order on the symbols in the equivalence classes, such that for all i and j, $\bar{C}_i = \bar{C}_j$ with the order preserved.*

Let $f : \{C' \in \mathcal{C}_\mathcal{T} \mid C' \prec C\} \to \mathcal{P}(\Lambda)$ be a function associating an interpretation to every type constructor strictly smaller than the one we are currently considering, namely C.

We define the function

$$K^f_{\bar{C}} : \quad \mathcal{P}(\Lambda)^n \quad \to \quad \mathcal{P}(\Lambda)^n$$

$$(X_i)_{i \leqslant n} \quad \mapsto \quad \left(\left\{ t \in \mathrm{SN} \;\middle|\; \begin{array}{l} \text{if } t \rightsquigarrow^* c\, v_1 \dots v_m \text{ with } \left\{ \begin{array}{l} c \in \mathcal{C}_o \\ \Theta(c) = \overrightarrow{(x : T)} \Rightarrow (C_i\, \vec{s}) \\ m \geqslant \mathrm{ar}(c) \end{array} \right. \\ \text{then for all } j \in \mathrm{Acc}(c), v_j \in R^f_{\bar{C}}(T_j, (X_k)_{k \leqslant n}) \end{array} \right\} \right)_{i \leqslant n}$$

$$\text{with } R^f_{\bar{C}}(T, (X_k)_{k \leqslant n}) = \begin{cases} X_i & \text{if there are } \vec{l} \text{ such that } T = C_i\, \vec{l} \\ \left\{ t \in \Lambda \;\middle|\; \text{for all } u \in R^f_{\bar{C}}(T_1, \vec{X}), t\, u \in R^f_{\bar{C}}(T_2\,[u/_x], \vec{X}) \right\} & \text{if } T = (x : T_1) \Rightarrow T_2 \\ f(C') & \text{if } T = C'\, \vec{u}, \text{ and } C' \prec C \end{cases}$$

One must note here, that the definition of the accessible positions of a constructor ensures us that the first argument of $R^f_{\bar{C}}$ is either a product or the application of a type constructor, hence that no reduction can occur at the head of this type. This property ensures us that the case analysis on the shape of the type in the definition of $R^f_{\bar{C}}$ is stable by reduction. Furthermore, $R^f_{\bar{C}}$ is only defined on $\mathrm{FrozTyp}_{\preccurlyeq C}$, for which there are no issues thanks to the definition of accessibility.

It must also be noted, that the arguments of the type constructors are completely irrelevant in our interpretation of type values. Especially, in the definition of $R^f_{\bar{C}}$, a substitution is performed in the codomain of products to be similar to what is done usually with dependent types, however performing or not this substitution has no impact on the definition of the function $R^f_{\bar{C}}$, since it only affects arguments of a type constructor.

Lemma 6.2.2 (Monotonicity of $R^f_{\bar{C}}$)**.** *Let $C \in \mathcal{C}_\mathcal{T}$, $f : \{C' \in \mathcal{C}_\mathcal{T} \mid C' \prec C\} \to \mathcal{P}(\Lambda)$, and $\bar{C}$ be the equivalence class of C.*

- *for all $T \in \mathrm{FrozTyp}_{\prec C}$, $R^f_{\bar{C}}(T, .)$ is constant,*
- *for all $T \in \mathrm{FrozTyp}_{\preccurlyeq C}$, $R^f_{\bar{C}}(T, .)$ is increasing.*

Proof. By mutual induction on the number of products in T:

- If T is not a product, then
  - if $T \in \mathrm{FrozTyp}_{\prec C}$, $R^f_{\bar{C}}(T, (X_i)_{i \leqslant n})$ does not depend on $(X_i)_{i \leqslant n}$, so $R^f_{\bar{C}}(T, .)$ is constant,
  - if $T \in \mathrm{FrozTyp}_{\preccurlyeq C}$, $R^f_{\bar{C}}(T, .)$ is increasing, since constant functions and projections are increasing.
- If $T = (x : T_1) \Rightarrow T_2$ is a product, contravariant and covariant positions of products, together with the induction hypothesis, allow us to conclude:

– If $T \in \mathrm{FrozTyp}_{\prec C}$, then $T_1, T_2 \in \mathrm{FrozTyp}_{\prec C}$.
Hence, by induction hypothesis, $R^f_{\bar{C}}(T_1, \vec{X})$ is independent of $\vec{X}$. Similarly for $R^f_{\bar{C}}(T_2, \vec{X})$.
So $R^f_{\bar{C}}((x : T_1) \Rightarrow T_2, \vec{X}) = \left\{ t \in \Lambda \;\middle|\; \text{for all } u \in R^f_{\bar{C}}(T_1, \vec{X}), t\,u \in R^f_{\bar{C}}(T_2\,[u/_x], \vec{X}) \right\}$ is independent of $\vec{X}$.

– If $T \in \mathrm{FrozTyp}_{\preccurlyeq C}$, then $T_1 \in \mathrm{FrozTyp}_{\prec C}$ and $T_2 \in \mathrm{FrozTyp}_{\preccurlyeq C}$.
Let $(X_i)_{i \leqslant n} \subseteq_{prod} (Y_i)_{i \leqslant n} \in \mathcal{P}(\Lambda)^n$, $t \in R^f_{\bar{C}}(T, (X_i)_{i \leqslant n})$ and $u \in R^f_{\bar{C}}(T_1, (Y_i)_{i \leqslant n})$.
Since $T_1 \in \mathrm{FrozTyp}_{\prec C}$, $R^f_{\bar{C}}(T_1, (Y_i)_{i \leqslant n}) = R^f_{\bar{C}}(T_1, (X_i)_{i \leqslant n})$, hence $u \in R^f_{\bar{C}}(T_1, (X_i)_{i \leqslant n})$.
By definition of $R^f_{\bar{C}}(T, (X_i)_i)$, $t\,u \in R^f_{\bar{C}}(T_2\,[u/_x], (X_i)_{i \leqslant n})$. Since $T_2 \in \mathrm{FrozTyp}_{\preccurlyeq C}$ and $\mathrm{FrozTyp}_{\preccurlyeq C}$ is stable by substitution (see Remark 6.1.5), $T_2\,[u/_x] \in \mathrm{FrozTyp}_{\preccurlyeq C}$ too.
So one can apply the induction hypothesis, to get that $R^f_{\bar{C}}(T_2\,[v/_x], .)$ is increasing, so $t\,u \in R^f_{\bar{C}}(T_2\,[u/_x], (Y_i)_{i \leqslant n})$.
We can then conclude that $t \in R^f_{\bar{C}}(T, (Y_i)_{i \leqslant n})$. □

Lemma 6.2.3 (Monotonicity of $K^f_{\bar{C}}$)**.** *For all $C \in \mathcal{C}_\mathcal{T}$ and $f : \{C' \in \mathcal{C}_\mathcal{T} \mid C' \prec C\} \to \mathcal{P}(\Lambda)$, $K^f_{\bar{C}}$ is increasing.*

Proof. Let $(X_i)_{i \leqslant n} \subseteq_{prod} (Y_i)_{i \leqslant n} \in \mathcal{P}(\Lambda)^n$, $k \leqslant n$ and $t \in \pi_k\left(K^f_{\bar{C}}((X_i)_{i \leqslant n})\right)$.

- If t does not reduce to any term of the shape $c\,\vec{v}$ where c is a constructor of C_i and $|\vec{v}| \geqslant \mathrm{ar}(c)$, then $t \in \pi_k\left(K^f_{\bar{C}}((Y_i)_{i \leqslant n})\right)$, since $t \in \pi_k\left(K^f_{\bar{C}}((X_i)_{i \leqslant n})\right)$ ensures that t is strongly normalizing, and it is the only requirement for t to be in $\pi_k\left(K^f_{\bar{C}}((Y_i)_{i \leqslant n})\right)$.

- If $t \rightsquigarrow^* c\,\vec{v}$ where c is a constructor of C_i and $|\vec{v}| \geqslant \mathrm{ar}(c)$. Then $\Theta(c) = \overrightarrow{(x : T)} \Rightarrow C_k\,\vec{s}$ and for all $j \in \mathrm{Acc}(c)$, $v_j \in R^f_{\bar{C}}(U_j, (X_i)_{i \leqslant n})$.
Since for all $j \in \mathrm{Acc}(c)$ with $j \leqslant r$, $U_j \in \mathrm{FrozTyp}_{\preccurlyeq C_k}$, Lemma 6.2.2 ensures us that for all $j \in \mathrm{Acc}(c)$ with $j \leqslant r$, $v_j \in R^f_{\bar{C}}(T_j, (Y_i)_{i \leqslant n})$. So $t \in \pi_k\left(K^f_{\bar{C}}((Y_i)_{i \leqslant n})\right)$. □

For all n, $\mathcal{P}(\Lambda)^n$ is a complete lattice, hence every increasing function of $\mathcal{P}(\Lambda)^n \to \mathcal{P}(\Lambda)^n$ has a least fixpoint, by the Knaster-Tarski theorem [Tar28].

Definition 6.2.4 (Pre-interpretation of $C\,\vec{t}$)**.** *Let $C\,\vec{t}$ be a type value, such that there are no $D \prec C$. Let $(C_1, \ldots, C_n)$ be the tupled version of the equivalence class $\bar{C}$ and i be the index such that $C_i = C$. Then, the* pre-interpretation *of $C\,\vec{t}$ is $\mathcal{I}_C$, the i^{th} projection of the least fixpoint of $K^\emptyset_{\bar{C}}$.*

Iteratively, if $C\,\vec{t}$ is a type value such that $\mathcal{I}_D$ is defined for all $D \prec C$, $(C_1, \ldots, C_n)$ is the tupled version of the equivalence class $\bar{C}$ and i be the index such that $C_i = C$. Then, the pre-interpretation *of $C\,\vec{t}$ is $\mathcal{I}_C$, the i^{th} projection of the least fixpoint of $K^\mathcal{I}_{\bar{C}}$.*

As already stated, even if we are defining the pre-interpretation of $C\,\vec{t}$, since we only want to interpret types, this interpretation only depends of C, and not of the arguments $\vec{t}$.

6.2.2 Interpretation of $\star$ and of types

Now that the interpretation of type values, which was the key point in the definition, is fully elucidated, we can describe the interpretation of any type. To do so, we will define simultaneously the interpretation of the sort $\star$ and of its elements.

For this, we will define the interpretation of $\star$ as the fixpoint of $[\![\star]\!]^- : \mathcal{C}_\mathcal{T}/_\approx \cup \{\bot\} \to \mathrm{Ord} \to \mathcal{P}(\Lambda)$. At every step, one will add some new types in the interpretation of $\star$ and then define their interpretations, before constructing the next step of the interpretation of $\star$.

Definition 6.2.5 (Seeding of the interpretation of $\star$)**.** *We add $\bot$ as a minimum to $\mathcal{C}_\mathcal{T}/_\approx$ with respect to $\prec$. Then $[\![\star]\!]^0_\bot = \emptyset$.*

Definition 6.2.6 (Hereditary interpretation of $\star$ and of types)**.** *Let α be an ordinal and $\bar{D} \in \mathcal{C}_\mathcal{T}/_\approx \cup \{\bot\}$, such that $[\![\star]\!]^\alpha_{\bar{D}}$ is defined and for all $T \in [\![\star]\!]^\alpha_{\bar{D}}$, $[\![T]\!]$ is defined too.*

Then let

$$N^\alpha_{\bar{D}} = \{A \in \mathcal{N}_\mathcal{T} \mid \textit{for all } U, A \rightsquigarrow U \textit{ implies } U \in [\![\star]\!]^\alpha_{\bar{D}}\}$$
$$\Pi^\alpha_{\bar{D}} = \{(x : A) \Rightarrow B \mid A \in [\![\star]\!]^\alpha_{\bar{D}} \textit{ and for all } a \in [\![A]\!], B\,[a/_x] \in [\![\star]\!]^\alpha_{\bar{D}}\}$$

We define:

$$[\![\star]\!]^{\alpha+1}_{\bar{D}} = [\![\star]\!]^\alpha_{\bar{D}} \cup N^\alpha_{\bar{D}} \cup \Pi^\alpha_{\bar{D}}$$

and if

$$T \in (N^\alpha_{\bar{D}} \cup \Pi^\alpha_{\bar{D}}) \setminus [\![\star]\!]^\alpha_{\bar{D}}\,,$$

then

$$[\![T]\!] = \begin{cases} \mathcal{I}_C & \textit{if } T\Downarrow = C\,\vec{u} \textit{ with } C \in \mathcal{C}_\mathcal{T} \\ \{t \mid \textit{for all } u \in [\![A]\!], t\,u \in [\![B\,[u/_x]]\!]\} & \textit{if } T\Downarrow = (x : A) \Rightarrow B \\ \mathrm{SN} & \textit{if } (T\Downarrow) \in \mathcal{N}_\mathcal{T} \end{cases}$$

For $[\![T]\!]$ to be well-defined, $T\Downarrow$ must be defined, meaning that T must be strongly normalizing and have a unique normal form. The proof that $T \in \mathrm{SN}$ is in Proposition 6.3.5. This proposition is proved later only for readability purpose, and could have been included here, interleaved with the definitions of interpretations. More generally, all the definitions and lemmas of Section 6.2 and Section 6.3 are interdependent. The choice to present them separately require the readers to make an effort to convince themselves that all those results could have presented interleaved. However, for the author, this effort was much preferable to a very hard to disentangle proof of five pages.

Once the strong normalization has been proved, the unicity of the normal form is a consequence of Newman's lemma [New42], thanks to the local confluence hypothesis (Condition 6.0.2).

In this interpretation of successor ordinals, $\bar{D}$ was not really used. This is because the construction of the interpretation is done in two steps, iterated several times, one first adds the type values of the smallest class not yet interpreted (when the ordinal is 0), and then saturates it, to interpret neutral types and dependent arrows which can be interpreted.

Definition 6.2.7 (Limit and initial interpretations of $\star$ and types)**.** *Let μ be a limit ordinal, $\bar{D} \in \mathcal{C}_\mathcal{T}/_\approx \cup \{\bot\}$, such that for all $(\bar{Z}, \alpha) <_{lex} (\bar{D}, \mu)$, $[\![\star]\!]^\alpha_{\bar{Z}}$ is defined and for all T in $[\![\star]\!]^\alpha_{\bar{Z}}$, $[\![T]\!]$ is defined too.*

Then

$$[\![\star]\!]^\mu_{\bar{D}} = \bigcup_{\alpha<\mu} [\![\star]\!]^\alpha_{\bar{D}}$$

Furthermore, if for all $(\bar{Z}, \alpha) <_{lex} (\bar{D}, 0)$, $[\![\star]\!]^\alpha_{\bar{Z}}$ is defined and for all T in $[\![\star]\!]^\alpha_{\bar{Z}}$, $[\![T]\!]$ is defined too,

$$[\![\star]\!]^0_{\bar{D}} = \bigcup_{\bar{Z}<\bar{D}} \overline{[\![\star]\!]_{\bar{Z}}} \cup \left\{C\,\vec{t} \in \mathcal{V}al_T \;\middle|\; C \in \bar{D}\right\}$$

Where, for $\bar{Z} \in \mathcal{C}_{\mathcal{T}}/_{\approx} \cup \{\bot\}$, $\overline{[\![\star]\!]_{\bar{Z}}}$ *is the fixpoint of the increasing sequence* $\left([\![\star]\!]^{\alpha}_{\bar{Z}}\right)_{\alpha}$.
And, if

$$T \in \left\{C\,\vec{t} \in \mathcal{V}al_T \;\middle|\; C \in \bar{D}\right\},$$

then there are C' *and* $\vec{t}$*, such that* $T = C'\,\vec{t}$ *and then*

$$[\![T]\!] = \mathcal{I}_{C'}$$

Finally,

$$[\![\star]\!] = \bigcup_{\bar{D} \in {}^{\mathcal{C}_{\mathcal{T}}}/_{\approx} \cup \{\bot\}} \overline{[\![\star]\!]_{\bar{D}}}$$

It must be noted that we do not give an interpretation to all the types, but only to the ones which are in $[\![\star]\!]$. This means that even if T is syntactically a type, it must be checked that $T \in [\![\star]\!]$ before stating properties on $[\![T]\!]$.

6.2.3 Interpretation of $\Box$ and of kinds

Definition 6.2.8 (Interpretation of Kind)**.** *We define* $[\![\mathrm{Kind}]\!]$ *as the smallest fixpoint of the increasing function defined on* $\mathcal{P}(\mathbb{K})$*:*

$$X \mapsto \{\star\} \cup \{(x : A) \Rightarrow K \mid A \in [\![\star]\!] \text{ and for all } a \in [\![A]\!], K\left[{}^{a}/_{x}\right] \in X\}$$

Definition 6.2.9 (Interpretation of kinds)**.** *We define the interpretation of kinds in* $[\![\mathrm{Kind}]\!]$ *by:*

- $[\![\star]\!]$ *was defined earlier,*
- $[\![(x : A) \Rightarrow K]\!] = \{t \mid \text{for all } u \in [\![A]\!], t\,u \in [\![K\left[{}^{u}/_{x}\right]]\!]\}$.

Since we are dealing with kinds, an induction on the number of products is possible. Hence $K\left[{}^{u}/_{x}\right]$ is smaller than $(x : A) \Rightarrow K$. This guarantees that the interpretations of kinds are well-defined.

Like for types, $[\![\Box]\!]$ does not contain all the kinds, hence there are kinds which are not given any interpretation.

6.3 Reducibility Candidates

Tait [Tai67] introduced a notion of "convertible" term, a property stronger than being terminating. Then, to deal with polymorphism, Girard [GLT88] enriched Tait's interpretation and called it *reducibility candidates*. The interpretation proposed is an extension to rewriting systems of this technique.

Definition 6.3.1 (Reducibility candidates)**.** $S \in \mathcal{P}(\Lambda)$ *is a* reducibility candidate *if*

- $S \subseteq \mathrm{SN}$,
- $\{u \mid \text{there is a } t \in S, t \rightsquigarrow u\} \subseteq S$,
- *if* t *is neutral and* $\{u \mid t \rightsquigarrow u\} \subseteq S$ *then* $t \in S$.

We denote by Cand *the set of reducibility candidates.*

Let us start by a useful lemma on the stability of candidates by products.

Lemma 6.3.2 (Product of candidates)**.** *If $P \in \text{Cand}$ and, for all $a \in P$, $Q(a) \in \text{Cand}$, then $\{t \mid \text{for all } a \in P, t\,a \in Q(a)\} \in \text{Cand}$.*

Proof. Let $R = \{t \mid \text{for all } a \in P, t\,a \in Q(a)\}$.

- Let $t \in R$. We have to prove that $t \in \text{SN}$. Let $x \in \mathcal{V}$. Since $P \in \text{Cand}$, $x \in P$. So, $t\,x \in Q(x)$. Since $Q(x) \in \text{Cand}$, $Q(x) \subseteq \text{SN}$. Therefore, $t\,x \in \text{SN}$, and $t \in \text{SN}$.
- Let $t \in R$ and t' such that $t \rightsquigarrow t'$. We have to prove that $t' \in R$. Let $a \in P$. We have to prove that $t'\,a \in Q(a)$. By definition, $t\,a \in Q(a)$ and $t\,a \rightsquigarrow t'a$. Since $Q(a) \in \text{Cand}$, $t'a \in Q(a)$.
- Let t be a neutral term such that $\{u \mid t \rightsquigarrow u\} \subseteq R$. We have to prove that $t \in R$.

  Hence, we take $a \in P$ and prove that $t\,a \in Q(a)$. Since $P \in \text{Cand}$, we have $a \in \text{SN}$ and $\{u \mid a \rightsquigarrow^* u\} \subseteq P$.

  We now prove that, for all $b \in \{u \mid a \rightsquigarrow^* u\}$, $t\,b \in Q(a)$, by induction on $\rightsquigarrow$. Since t is neutral, $t\,b$ is neutral too and it suffices to prove that $\{u \mid t\,b \rightsquigarrow u\} \subseteq Q(a)$. Since t is neutral, $\{u \mid t\,b \rightsquigarrow u\} = \{u\,b \mid t \rightsquigarrow u\} \cup \{t\,u \mid b \rightsquigarrow u\}$.

  - By induction hypothesis, $\{t\,u \mid b \rightsquigarrow u\} \subseteq Q(a)$.
  - By assumption, $\{u \mid t \rightsquigarrow u\} \subseteq R$. So, $\{u\,a \mid t \rightsquigarrow u\} \subseteq Q(a)$. Since $Q(a) \in \text{Cand}$, $\{u\,b \mid t \rightsquigarrow u\} \subseteq Q(a)$ too.

  Therefore, $t\,a \in Q(a)$ and $t \in R$. □

Now, we will prove that all our interpretations of types given in Section 6.2 are reducibility candidates.

For that, we first state a useful lemma on the possibility to reduce a type without modifying its interpretation.

Lemma 6.3.3. *If $T, U \in [\![\star]\!]^{\alpha}_{\bar{D}}$ and $T \rightsquigarrow U$, then $[\![T]\!] = [\![U]\!]$.*

Proof. When a type has an interpretation, it only depends on the normal form. □

Lemma 6.3.4 (Reducibility of $\mathcal{I}_C$)**.** *If $C \in \mathcal{C}_{\mathcal{T}}$, then $\mathcal{I}_C \in \text{Cand}$.*

Proof. First note that $\mathcal{I}_C = \pi_k\left(K^{\mathcal{T}}_{\bar{C}}(\mathcal{I}_{C_1}, \ldots, \mathcal{I}_{C_n})\right)$.

- By definition of $K^{\mathcal{T}}_{\bar{C}}$, we have $K^{\mathcal{T}}_{\bar{C}}(\mathcal{I}_{C_1}, \ldots, \mathcal{I}_{C_n}) \subseteq \text{SN}^n$.
- Let $t \in \mathcal{I}_C$ and u be such that $t \rightsquigarrow u$. Since $t \in \text{SN}$, $u \in \text{SN}$ too. If $u \rightsquigarrow^* c\,\vec{v}$, with c a constructor of C and $|\vec{v}| \geqslant \text{ar}(c)$, then $t \rightsquigarrow^* c\,\vec{v}$ so the constraint on the reducibility of the accessible arguments is fulfilled. Hence, $\mathcal{I}_C$ is stable by reduction.
- Let t be a neutral term such that $\{u \mid t \rightsquigarrow u\} \subseteq \mathcal{I}_C$. $t \in \text{SN}$. If $t \rightsquigarrow^* c\,\vec{v}$, with c a constructor of C and $|\vec{v}| \geqslant \text{ar}(c)$, then since t is neutral, $t \neq c\,\vec{v}$. Hence, there is a $u \in \{u \mid t \rightsquigarrow u\}$ such that $u \rightsquigarrow^* c\,\vec{v}$ so the reducibility of the accessible arguments is fulfilled. □

Proposition 6.3.5 (Reducibility of the interpretation of types)**.** *For all $T \in [\![\star]\!]$, $T \in \text{SN}$ and $[\![T]\!] \in \text{Cand}$.*

Proof. Let us show that for all $T \in [\![\star]\!]^{\alpha}_{\bar{D}}$, $T \in \text{SN}$ and $[\![T]\!] \in \text{Cand}$ by mutual induction on $(\bar{D}, \alpha)$.

- If $(\bar{D}, \alpha) = (\bot, 0)$, then $[\![\star]\!]^\alpha_{\bar{D}} = \emptyset$, so the proposition is vacuously true.
- Let $\bar{D} \in \mathcal{C}_\mathcal{T}/_\approx \cup \{\bot\}$ and α an ordinal be such that for all $T \in [\![\star]\!]^\alpha_{\bar{D}}$, $T \in \mathrm{SN}$ and $[\![T]\!] \in \mathrm{Cand}$, and let us show that this remains true for $(\bar{D}, \alpha+1)$.

  To prove that $[\![\star]\!]^{\alpha+1}_{\bar{D}} \subset \mathrm{SN}$, one has to show that both $N^\alpha_{\bar{D}} \subset \mathrm{SN}$ and $\Pi^\alpha_{\bar{D}} \subset \mathrm{SN}$:

  - Since by induction hypothesis $[\![\star]\!]^\alpha_{\bar{D}} \subset \mathrm{SN}$, one also has
    $$N^\alpha_{\bar{D}} = \{A \in \mathcal{N}_\mathcal{T} \mid \text{for all } U, A \rightsquigarrow U \text{ implies } U \in [\![\star]\!]^\alpha_{\bar{D}}\} \subset \mathrm{SN}\,.$$
  - To show that $\Pi^\alpha_{\bar{D}} = \{(x:A) \Rightarrow B \mid A \in [\![\star]\!]^\alpha_{\bar{D}} \text{ and for all } a \in [\![A]\!], B\,[a/_x] \in [\![\star]\!]^\alpha_{\bar{D}}\} \subseteq \mathrm{SN}$, one just has to show that $A, B \in \mathrm{SN}$, since if $(x:A) \Rightarrow B \rightsquigarrow^* T$, then $T = (x:A') \Rightarrow B'$ with $A \rightsquigarrow^* A'$ and $B \rightsquigarrow^* B'$.

    By induction hypothesis $A \in [\![\star]\!]^\alpha_{\bar{D}} \subset \mathrm{SN}$ and $[\![A]\!]$ is a reducibility candidate, hence $x \in [\![A]\!]$, since any variable is a normal neutral term. So $B\,[x/_x] = B \in [\![\star]\!]^\alpha_{\bar{D}}$, so $B \in \mathrm{SN}$.

  Let $T \in [\![\star]\!]^{\alpha+1}_{\bar{D}}$ and let us show that $[\![T]\!] \in \mathrm{Cand}$. If $T \in [\![\star]\!]^\alpha_{\bar{D}}$, by induction hypothesis $[\![T]\!] \in \mathrm{Cand}$, otherwise we have shown that $T \in \mathrm{SN}$, and one can do a case distinction on the reason why $T \in [\![\star]\!]^{\alpha+1}_{\bar{D}}$:

  - If $T \in \Pi^\alpha_{\bar{D}}$, then $T = (x:A) \Rightarrow B$ and $[\![T]\!] = \{t \mid \text{for all } a \in [\![A]\!], t\,a \in [\![B\,[a/_x]]\!]\}$. By induction hypothesis, $[\![A]\!] \in \mathrm{Cand}$ and, for $a \in [\![A]\!]$, $[\![B\,[a/_x]]\!] \in \mathrm{Cand}$. Therefore, by Lemma 6.3.2 $[\![T]\!] \in \mathrm{Cand}$.
  - If $T \in N^{\alpha+1}_{\bar{D}}$, then for all U such that $T \rightsquigarrow U$, $U \in [\![\star]\!]^\alpha_{\bar{D}}$. If T is normal, then $[\![T]\!] = \mathrm{SN}$ which is a candidate, otherwise, there is a U_0 such that $T \rightsquigarrow U_0$, $U_0 \in [\![\star]\!]^\alpha_{\bar{D}}$, and by Lemma 6.3.3, we have $[\![T]\!] = [\![U_0]\!]$, which is a candidate by induction hypothesis. □
- Since in limit cases the interpretation $[\![\star]\!]^\mu_{\bar{D}}$ is simply a directed union of the previous $[\![\star]\!]^\alpha_{\bar{D}}$, it does not contain any new type T, to conclude one just has to study the cases of the form $(\bar{D}, 0)$. By definition of $\mathcal{V}al_T$, one has $\left\{C\,\vec{t} \in \mathcal{V}al_T\right\} \subset \mathrm{SN}$, so for all $T \in [\![\star]\!]^0_{\bar{D}}$, $T \in \mathrm{SN}$, and the newly added T are of the form $C\,\vec{t}$, so their interpretations are some $\mathcal{I}_C$, which are candidates, as proved in Lemma 6.3.4.

Proposition 6.3.6 (Reducibility of $[\![\star]\!]$)**.** *$[\![\star]\!]$ is a reducibility candidate.*

Proof.
- We proved in Proposition 6.3.5 that for all $(\bar{D}, \alpha)$, $[\![\star]\!]^\alpha_{\bar{D}} \in \mathrm{SN}$;
- Let us show that $[\![\star]\!]^\alpha_{\bar{D}}$ is closed by reduction, by induction on $(\bar{D}, \alpha)$.
  - $\emptyset = [\![\star]\!]^0_\bot$ is closed by reduction.
  - Let $T \in [\![\star]\!]^{\alpha+1}_{\bar{D}}$ and T' such that $T \rightsquigarrow T'$.
    If $T \in \{A \in \mathcal{N}_\mathcal{T} \mid \text{for all } U, A \rightsquigarrow U \text{ implies } U \in [\![\star]\!]^\alpha_{\bar{D}}\}$, then by definition, $T' \in [\![\star]\!]^\alpha_{\bar{D}} \subseteq [\![\star]\!]^{\alpha+1}_{\bar{D}}$.
    If $T \in \{(x:A) \Rightarrow B \mid A \in [\![\star]\!]^\alpha_{\bar{D}} \text{ and for all } a \in [\![A]\!], B\,[a/_x] \in [\![\star]\!]^\alpha_{\bar{D}}\}$, then there are A and B such that $T = (x:A) \Rightarrow B$. Then either $T' = (x:A') \Rightarrow B$ with $A \rightsquigarrow A'$ or $T' = (x:A) \Rightarrow B'$ with $B \rightsquigarrow B'$. In both case, by induction hypothesis, T' is still in $\{(x:A) \Rightarrow B \mid A \in [\![\star]\!]^\alpha_{\bar{D}} \text{ and for all } a \in [\![A]\!], B\,[a/_x] \in [\![\star]\!]^\alpha_{\bar{D}}\} \subset [\![\star]\!]^{\alpha+1}_{\bar{D}}$.
  - $\left\{C\,\vec{t} \in \mathcal{V}al_T\right\}$ is stable by reduction since $C \in \mathcal{C}_\mathcal{T}$ hence no reduction can happen in head and SN is stable by reduction.

- Let T be a neutral term such that $\{U \mid T \rightsquigarrow U\} \subseteq [\![\star]\!]$.

  Since the rewriting system is finitely branching, $\{U \mid T \rightsquigarrow U\}$ is finite. Since $[\![\star]\!]$ is defined as the directed union of the $[\![\star]\!]^{\alpha}_{\bar{D}}$, there is a $\bar{D} \in \mathcal{C}_{\mathcal{T}}/_{\approx} \cup \{\bot\}$ and a α such that $\{U \mid T \rightsquigarrow U\} \subseteq [\![\star]\!]^{\alpha}_{\bar{D}}$. Then $T \in [\![\star]\!]^{\alpha+1}_{\bar{D}} \subseteq [\![\star]\!]$. □

Proposition 6.3.7 (Reducibility of $\Box$)**.** *$[\![\Box]\!]$ is a reducibility candidate.*

Proof. Being a stable by reduction subset of SN is stable by the function defining $[\![\Box]\!]$ and there is no neutral kind. □

Proposition 6.3.8 (Reducibility of kinds)**.** *For all $K \in [\![\Box]\!]$, $[\![K]\!]$ is a reducibility candidate.*

Proof. It is already proved for $\star$ and the Lemma 6.3.2 ensures us that we can form products. □

To conclude this section on the properties of the interpretation, we will show that the interpretation is stable by reduction:

Lemma 6.3.9 (Stability by reduction)**.** *For all T such that $[\![T]\!]$ is defined and for all U such that $T \rightsquigarrow U$, $[\![U]\!]$ is defined and $[\![T]\!] = [\![U]\!]$.*

Proof.

- If T is a type, since $[\![T]\!]$ is defined, $T \in [\![\star]\!]$. Since $[\![\star]\!]$ is a reducibility candidate, $[\![\star]\!]$ is stable by reduction, so $U \in [\![\star]\!]$, hence $[\![U]\!]$ is defined.

  Furthermore, by Lemma 6.3.3 $[\![T]\!] = [\![U]\!]$.

- If T is a kind, since $[\![T]\!]$ is defined, $T \in [\![\Box]\!]$. Since $[\![\Box]\!]$ is a reducibility candidate (Proposition 6.3.7), $[\![\Box]\!]$ is stable by reduction, so $U \in [\![\Box]\!]$, hence $[\![U]\!]$ is defined.

  To prove that $[\![T]\!] = [\![U]\!]$, let us do an induction on the number of arrows at the head of T.

  - If T does not contain any arrow, it means that $T = \star$, and $T \in \mathrm{NF}$, so the proposition is vacuously true,
  - If $T = (x : A) \Rightarrow K$, then either $U = (x : A') \Rightarrow K$ with $A \rightsquigarrow A'$, or $U = (x : A) \Rightarrow K'$ with $K \rightsquigarrow K'$.

    In the first case, since A is a type, one can use the result we just proved to state that $[\![A]\!] = [\![A']\!]$. Since $[\![T]\!] = \{t \mid \text{for all } u \in [\![A]\!], t\,u \in [\![K\,[u/_x]]\!]\}$ and $[\![U]\!] = \{t \mid \text{for all } u \in [\![A']\!], t\,u \in [\![K\,[u/_x]]\!]\}$, we obtain that $[\![T]\!] = [\![U]\!]$.

    In the other case, K has less arrows than T, so by induction hypothesis $[\![K]\!] = [\![K']\!]$. Since objects cannot contain arrows, one also has that for all u, $[\![K\,[u/_x]]\!] = [\![K'\,[u/_x]]\!]$, hence $[\![T]\!] = \{t \mid \text{for all } u \in [\![A]\!], t\,u \in [\![K\,[u/_x]]\!]\} = \{t \mid \text{for all } u \in [\![A]\!], t\,u \in [\![K'\,[u/_x]]\!]\} = [\![U]\!]$. □

This property allows us to explicit what the interpretation of type values is.

Lemma 6.3.10 (Explicit interpretation of type values)**.** *For all $C \in \mathcal{C}_{\mathcal{T}}$, if for all $c \in \mathcal{C}_o$ such that there is $\vec{s}$ such that $\Theta(c) = \overrightarrow{(x : T)} \Rightarrow (C\,\vec{s})$, one has $\Theta(c) \in [\![\star]\!]$, then*

$$\mathcal{I}_C = \left\{ t \in \mathrm{SN} \;\middle|\; \begin{array}{l} \text{if } t \rightsquigarrow^* c\,v_1 \dots v_m \text{ with } \left\{ \begin{array}{l} c \in \mathcal{C}_o \\ \Theta(c) = \overrightarrow{(x : T)} \Rightarrow (C\,\vec{s}) \\ m \geqslant \mathrm{ar}(c) \end{array} \right. \\ \text{then for all } j \in \mathrm{Acc}(c), v_j \in [\![T_j]\!] \end{array} \right\}$$

Proof. Let $\bar{C} = (C_1, \dots, C_n)$ be the tupled version of the equivalence class of C. We have $\mathcal{I}_{C_i} = K^{\mathcal{I}}_{\bar{C}}(\mathcal{I}_{C_1}, \dots, \mathcal{I}_{C_n})$ (see Definition 6.2.1).

$$\text{So} \quad \mathcal{I}_{C_i} \quad = \quad \left\{ t \in \text{SN} \;\middle|\; \begin{array}{l} \text{if } t \rightsquigarrow^* c\, v_1 \dots v_m \text{ with } \left\{ \begin{array}{l} c \in \mathcal{C}_o \\ \Theta(c) = \overrightarrow{(x : T)} \Rightarrow (C_i\, \vec{s}) \\ m \geqslant \text{ar}(c) \end{array} \right. \\ \text{then for all } j \in \text{Acc}(c), v_j \in R'_{\bar{C}}(T_j) \end{array} \right\}, \quad \text{with}$$

$$R'_{\bar{C}}(T) = \begin{cases} \mathcal{I}_{C_i} & \text{if there are } \vec{l} \text{ such that } T = C_i\, \vec{l} \\ \left\{ t \in \Lambda \;\middle|\; \text{for all } u \in R'_{\bar{C}}(T_1), t\, u \in R'_{\bar{C}}(T_2\, [u/_x]) \right\} & \text{if } T = (x : T_1) \Rightarrow T_2 \\ \mathcal{I}_{C'} & \text{if } T = C'\, \vec{u}, \text{ and } C' \prec C \end{cases}$$

Here $R'_{\_}(\_)$ is used as a shortcut for the function denoted in Definition 6.2.1 by $R^{\mathcal{I}}_{\_}(\_, (\mathcal{I}_{C_1}, \dots, \mathcal{I}_{C_n}))$.

Hence, to conclude the proof, one has to show that for all $T \in \text{FrozTyp}_{\preccurlyeq C}$, such that $T \in [\![\star]\!]$, $R'_{\bar{C}}(T) = [\![T]\!]$.

We prove it by induction on the number of products in T.

- If T is not a product, it is of the shape $D\, \vec{t}$ with $D \in \mathcal{C}_{\mathcal{T}}$, and $D \preccurlyeq C$, so $R'_{\bar{C}}(T) = \mathcal{I}_D = [\![T]\!]$.
- If $T = (x : T_1) \Rightarrow T_2$, then by induction hypothesis $R'_{\bar{C}}(T_1) = [\![T_1]\!]$ and for all $u \in [\![T_1]\!]$, $R'_{\bar{C}}(T_2\, [u/_x]) = [\![T_2\, [u/_x]]\!]$, since substituting a variable by an object in a frozen type does not modify the number of products. Since the interpretations are stable by reduction, one also has $R'_{\bar{C}}(T_1) = [\![T_1 \Downarrow]\!]$ and or all $u \in [\![T_1 \Downarrow]\!] = [\![T_1]\!]$, $R'_{\bar{C}}(T_2\, [u/_x]) = [\![T_2 \Downarrow [u/_x]]\!]$. By definition $[\![T]\!] = \{t \in \Lambda \mid \text{for all } u \in [\![T_1 \Downarrow]\!], t\, u \in [\![T_2 \Downarrow [u/_x]]\!]\}$, which is equal to $\mathbb{R}'_{\bar{C}}(T) = \left\{ t \in \Lambda \;\middle|\; \text{for all } u \in R'_{\bar{C}}(T_1), t\, u \in R'_{\bar{C}}(T_2\, [u/_x]) \right\}$. □

6.4 Validity

Definition 6.4.1 (Valid substitution)**.** *$\sigma \vDash \Gamma$ if for all $(x : A) \in \Gamma$, $x\sigma \in [\![A\sigma]\!]$*

Definition 6.4.2 (Valid typing map)**.** *Θ is valid if for all $f \in \mathcal{F}$, such that $\vdash \Theta(f) : s(f)$ and $\Theta(f) \in [\![s(f)]\!]$, we have $f \in [\![\Theta(f)]\!]$.*

Theorem 6.4.3 (Adequacy)**.** *Let σ be a substitution, Θ a typing map, Γ an environment and t and T two terms.*

If Θ is valid, $\Gamma \vdash t : T$ and $\sigma \vDash \Gamma$, then $t\sigma \in [\![T\sigma]\!]$.

Proof. Let σ be such that $\sigma \vDash \Gamma$. We prove this lemma by induction on $\Gamma \vdash t : T$.

(ax) $\star \in [\![\Box\sigma]\!] = [\![\Box]\!]$ by definition,

(var) By assumption on σ, we have for all $(x : A) \in \Gamma$, $x\sigma \in [\![A\sigma]\!]$,

(prod) Up to α-renaming, we can assume $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$. So $((x : A) \Rightarrow B)\, \sigma = (x : A\sigma) \Rightarrow B\sigma$. By induction hypothesis, $A\sigma \in [\![\star]\!]$. Let $a \in [\![A\sigma]\!]$. Let $\sigma' = [a/_x, \sigma]$. $A\sigma = A\sigma'$, so $\sigma' \vDash \Gamma, x : A$, and by induction hypothesis, $B\sigma' \in [\![s\sigma']\!] = [\![s]\!]$. So $((x : A) \Rightarrow B)\, \sigma \in [\![s\sigma]\!] = [\![s]\!]$.

(abs) Up to α-renaming, we can assume the $x \notin \text{dom}(\sigma) \cup \text{FV}(\sigma)$. So $(\lambda(x : A).t)\sigma = \lambda(x : A\sigma).t\sigma$ and $((x : A) \Rightarrow B)\, \sigma = (x : A\sigma) \Rightarrow B\sigma$. Let $a \in [\![A\sigma]\!]$ and $\sigma' = [a/_x, \sigma]$. By induction hypothesis, we have $t\sigma' \in [\![B\sigma']\!]$, because, as above, $\sigma' \vDash \Gamma, x : A$.

Let a', A' and t' be such that $a \rightsquigarrow^* a'$, $A\sigma \rightsquigarrow^* A'$ and $t\sigma \rightsquigarrow^* t'$. We prove that $(\lambda(x : A').t')\, a' \in [\![B\sigma']\!]$ by induction on the reduction of (A', t', a'). Since $(\lambda(x : A').t')\, a'$ is neutral and $[\![B\sigma']\!] \in \text{Cand}$, it suffices to prove that $\{u \mid (\lambda(x : A\sigma).t\sigma)\, a \rightsquigarrow u\} \subseteq [\![B\sigma']\!]$.

- For the toplevel β-reduction, $(\lambda(x : A').t')\, a' \rightsquigarrow t' \left[a'/_x\right]$. By induction on the derivation, we have $t\sigma' \in [\![B\sigma']\!]$. Since $[\![B\sigma']\!]$ is a candidate, the reduct $t' \left[a'/_x\right]$ of $t\sigma'$ is also in $[\![B\sigma']\!]$.
- Otherwise the reduction takes place in A', t' or a', we conclude by induction hypothesis on the reduction sequences.

(app) By induction hypothesis, $t\sigma \in [\![\Pi(x : A\sigma).B\sigma]\!]$ and $u\sigma \in [\![A\sigma]\!]$. By definition, of the interpretation of a product, we have $(t\sigma)\, u\sigma = (t\, u)\sigma \in [\![B\sigma\, [u\sigma/_x]]\!] = [\![B\, [u/_x]\, \sigma]\!]$, since x can be chosen not in $\mathrm{dom}(\sigma) \cup \mathrm{FV}(\sigma)$ by α-renaming.

(sig) By induction hypothesis, we have $\Theta(f) \in [\![s(f)]\!]$. So, since $\vdash \Theta(f) : s(f)$, we have $f \in [\![\Theta(f)]\!]$ since Θ is valid.

(conv) By induction hypothesis, $B\sigma \in [\![s]\!]$, so $[\![B\sigma]\!]$ is defined. Since A and B are joinable, there is a C such that $A \rightsquigarrow^* C$ and $B \rightsquigarrow^* C$. Since $\rightsquigarrow$ is stable by substitution, one also has $A\sigma \rightsquigarrow^* C\sigma$ and $B\sigma \rightsquigarrow^* C\sigma$. By applying several time Lemma 6.3.9, one obtains that $[\![A\sigma]\!] = [\![B\sigma]\!]$ and since by induction hypothesis $t\sigma \in [\![A\sigma]\!]$, we have $t\sigma \in [\![B\sigma]\!]$.

(weak) Since $\sigma \vDash \Gamma, x : A$, one has $\sigma \vDash \Gamma$, so by induction hypothesis, $t\sigma \in [\![T\sigma]\!]$. □

Obtaining "adequacy lemmas" is the main goal of this chapter, so one could imagine that we are over now that Theorem 6.4.3 is proved. But we are only halfway there, since the adequacy was obtained under the hypothesis that Θ is valid, an undecidable property. Hence, we are now looking for conditions to ensure that a typing map is valid. This quest leads us to Theorem 6.6.12, allowing to formulate a new version of the adequacy in Corollary 6.6.13.

6.5 Fully Applied Signature Symbol and Structural Order

In Section 6.1, we defined a notion of accessible position under a constructor. The choice of the word "accessible" let us think that one want to restrict the subterm relation to only be able to extract the subterms which are at an accessible position.

Definition 6.5.1 (Order associated to accessible subterms)**.** *We define $\triangleright_{acc}$ as the transitive closure of $\left(c\, t_1 \dots t_{\mathrm{ar}(c)}\right) \triangleright_{acc} (t_i\, \vec{u})$, where $c \in \mathcal{C}_o$, $\Theta(c) = \overrightarrow{(x : T)} \Rightarrow U$, $i \in \mathrm{Acc}(c)$, $T_i = \overrightarrow{(y : V)} \Rightarrow W$ and $\left[\overrightarrow{u/_y}\right] \vDash \overrightarrow{(y : V)}$.*

$\triangleright_{acc}$ is not the restriction of the subterm relation to accessible positions, since we allow ourselves to "invent" the arguments $\vec{u}$ which will be applied to the subterm t_i. This order is similar to the *structural order* of Coquand [Coq92], and we will prove that it is a well-founded relation on the set of terms composed of a symbol of the signature applied to arguments in the interpretation of the expected type.

Definition 6.5.2 (Function applied to reducible terms)**.** *Let*

$$\mathbb{U} = \left\{ f\, \vec{t} \;\middle|\; \begin{array}{c} f \in \mathcal{F}, \\ \vdash \Theta(f) : s(f), \\ \Theta(f) \in [\![s(f)]\!], \\ \Theta(f) = \overrightarrow{(x : T)} \Rightarrow U, \\ |\vec{t}| = \mathrm{ar}(f) \\ \left[\overrightarrow{t/_x}\right] \vDash \overrightarrow{(x : T)} \end{array} \right\}$$

Lemma 6.5.3. *There is no infinite sequence $(t_i)_{i\in\mathbb{N}}$ such that $t_0 \in \mathbb{U}$ and for all $i \in \mathbb{N}$, $t_i \triangleright_{acc} t_{i+1}$.*

Proof. Let us assume that there is such an infinite sequence.

First note that every t_i is headed by an element of $\mathcal{C}_o$, otherwise there is no t such that $t_i \triangleright_{acc} t$.

Hence, among all infinite sequences, let us choose a minimal one in the sense that $t_0 = f\,\vec{u}$ with $\Theta(f) = \overrightarrow{(x:T)} \Rightarrow C\,\vec{v}$ and for all $C' \prec C$, there is no infinite sequence starting by a constructor of C'. So all the t_i's are headed by a constructor of a $D \approx C$.

We will prove by induction that for every i, there is a t'_i such that $t_i = t'_i\,\vec{w}$ and for all i, $t'_{i+1} \triangleleft t'_i$.

We start with $t'_0 = t_0$. Let i be such that the sequence of $(t'_k)_{0\leqslant k\leqslant i}$ is constructed. Hence $t_i = t'_i\,\vec{w}$. Since t_i is headed by a constructor, t'_i too, $t'_i = c\,\vec{u}$. Since $t'_i\,\vec{w} \triangleright_{acc} t_{i+1}$, two options:

- either $t_{i+1} = u_j\,\vec{a}$, then $u_j \triangleleft t'_i$ hence there is a t'_{i+1} (namely u_j) such that $t_{i+1} = t'_{i+1}\,\vec{a}$ and $t'_{i+1} \triangleleft t'_i$.
- or $t_{i+1} = w_j\,\vec{a}$, in this case i is necessarily strictly greater than 0, since by definition $t'_0 = t_0$, hence $|\vec{w}| = 0$ in the case $i = 0$.

  This case contradicts the minimality hypothesis. Indeed $t_{i-1} = c_{i-1}\,\vec{v}$, with $\Theta(c_{i-1}) = \overrightarrow{(x:T)} \Rightarrow C_{i-1}\,\vec{u}'$, and there is a $r \in \mathrm{Acc}(c_{i-1})$, such that $t'_i = v_r$, $t_i = t'_i\,\vec{w}$ and $T_r = \overrightarrow{(x:U)} \Rightarrow V \in \mathrm{FrozTyp}_{\preccurlyeq C}$. So all U_k are products ended by a $D_k\,\vec{v}'$ with $D_k \in \mathcal{C}_\mathcal{T}$ and $D_k \prec C$.

  Hence, for all k, $w_k \in [\![U_k]\!] = \mathcal{I}_{D_k}$.

  So, since w_j is headed by a constructor, because t_{i+1} is, this constructor is one of D_k which violates the minimality property.

Hence the existence of such an infinite sequence for $\triangleright_{acc}$ leads to the existence of an infinite sequence of subterms of t_0, contradicting the well-foundedness of $\triangleleft$. □

We will even prove more than the well-foundedness of $\triangleright_{acc}$ on $\mathbb{U}$, we will also allow to interleave reduction in arguments.

Definition 6.5.4 ($\rightsquigarrow_{arg}$)**.** *Let $f\,\vec{t} \in \mathbb{U}$. $f\,\vec{t} \rightsquigarrow_{arg} u$ if $u = f\,\vec{t'}$, there is a i such that $t_i \rightsquigarrow t'_i$ and for all $j \neq i$, $t_j = t'_j$.*

Lemma 6.5.5. *$\triangleright_{acc} \cup \rightsquigarrow_{arg}$ is well-founded on $\mathbb{U}$.*

Proof. $\triangleright_{acc}$ is well-founded (Lemma 6.5.3). $\rightsquigarrow_{arg}$ is also well-founded, since all interpretations contain only strongly normalizing terms. Hence, it suffices to prove that $\triangleright_{acc}$ can be postponed (i.e. $\triangleright_{acc} \rightsquigarrow_{arg} \subseteq (\triangleright_{acc} \cup \rightsquigarrow_{arg} \triangleright_{acc})$) to get the well-foundedness of the union.

If $f\,\vec{t} \triangleright_{acc} t_i\,\vec{v} \rightsquigarrow_{arg} t'_i\,\vec{v}$, with $t_i \rightsquigarrow_{arg} t'_i$. The reduction in t_i can happen before the $\triangleright_{acc}$ and $f\,\vec{t} \rightsquigarrow_{arg} f\,t_1 \ldots t_{i-1}\,t'_i\,t_{i+1} \ldots \triangleright_{acc} t'_i\,\vec{v}$

On the other hand, if $f\,\vec{t} \triangleright_{acc} t_i\,\vec{v} \rightsquigarrow_{arg} t_i\,v_1 \ldots v_{j-1}\,v'_j\,v_{j+1} \ldots$, with $v_j \rightsquigarrow v'_j$, since the interpretations are closed by reduction, we can invent the reduced form v'_j, so $f\,\vec{t} \triangleright_{acc} t_i\,v_1 \ldots v_{j-1}\,v'_j\,v_{j+1} \ldots$.

So $\triangleright_{acc}$ can be postponed, hence $\triangleright_{acc} \cup \rightsquigarrow_{arg}$ is well-founded on $\mathbb{U}$. □

The relation $\triangleright_{acc}$ is the relation we want to define for the theoretical termination criterion. However, $\triangleright_{acc}$ does not simply extracts subterms, it allows to apply them to invented new arguments, under the condition that those arguments are in the interpretation of their types. This interpretability is undecidable in general, so what is really implemented is a sub-approximation of it:

Definition 6.5.6 (Decidable Subapproximation of $\triangleright_{acc}$)**.** *Let $C \in \mathcal{C}_{\mathcal{T}}$ and $T \in \mathrm{FrozTyp}_{\preccurlyeq C}$. Then we denote by X_T the set*

$$X_T = \left\{ x\,\vec{u} \;\middle|\; \begin{array}{l} \vec{u} \text{ are made only with variables and symbols of the signature} \\ \text{which are not defined by rewriting} \end{array} \right\}$$
$$\cup \left\{ c\,u_{i+1}\,\ldots\,u_n \;\middle|\; \begin{array}{l} c \in \mathcal{C}_o \\ C' \in \mathcal{C}_{\mathcal{T}} \\ T = (x_{i+1} : U_{i+1}) \Rightarrow \ldots \Rightarrow (x_n : U_n) \Rightarrow C'\,\vec{v} \\ \Theta(c) = (y_1 : V_1) \Rightarrow \ldots \Rightarrow (y_n : V_n) \Rightarrow C'\,\vec{v}' \\ \text{for all } j > i, u_j \in X_{V_j} \end{array} \right\}$$

The subapproximation of $\triangleright_{acc}$ *is the transitive closure of the relation*

$$\left\{ \left(f\,t_1 \ldots t_{\mathrm{ar}(c)}, t_i\,\vec{u}\right) \;\middle|\; \begin{array}{l} f \in \mathcal{C}_o \\ \Theta(f) = \overrightarrow{(x_j : U_j)} \Rightarrow C\,\vec{v} \\ i \in \mathrm{Acc}(f) \\ U_i = \overrightarrow{(y_k : V_k)} \Rightarrow W \\ \text{for all } k, u_k \in X_{V_k} \end{array} \right\}$$

This relation is a subapproximation of $\triangleright_{acc}$ since terms respecting this constraint are all in the interpretation of their types.

Indeed, variables are in the interpretation of any type, since it is a reducibility candidate (Section 6.3) and a variable is a normal neutral term. Furthermore, symbols of the signature which are not defined by rewriting rules are all in the interpretation of their type, as shown by the following lemma:

Lemma 6.5.7. *Symbols of the signature which are not defined by rewriting rules are all in the interpretation of their type.*

Proof.

- For non-constructors, let $f \in \mathcal{F}_o \setminus \mathcal{C}_o$ with $\Theta(f) = \overrightarrow{(x : A)} \Rightarrow B$, and let $u_i \in \left[\!\left[A_i \left[u_1/_{x_1}, \ldots, u_{i-1}/_{x_{i-1}}\right]\right]\!\right]$, $f\,\vec{u}$ is neutral, strongly normalizing and all its reducts are neutral, so $f\,\vec{u}$ is in all the interpretations, in particular $f\,\vec{u} \in \left[\!\left[B\left[\overrightarrow{u/_x}\right]\right]\!\right]$, and by definition of the interpretation of products $f \in \left[\!\left[\overrightarrow{(x : A)} \Rightarrow B\right]\!\right]$.

- For constructors, let $c \in \mathcal{C}_o$ with $\Theta(c) = \overrightarrow{(x : A)} \Rightarrow C\,\vec{v}$. Let $u_i \in \left[\!\left[A_i \left[u_1/_{x_1}, \ldots, u_{i-1}/_{x_{i-1}}\right]\right]\!\right]$. Since c is not defined, all the reducts of $c\,\vec{u}$ are of the shape $c\,\vec{u}'$ with for all i, $u_i \rightsquigarrow^* u_i'$. Since the interpretations are stable by reduction, all the $u_i' \in \left[\!\left[A_i \left[u_1/_{x_1}, \ldots, u_{i-1}/_{x_{i-1}}\right]\right]\!\right]$, especially when $i \in \mathrm{Acc}(c)$. Hence, by Lemma 6.3.10, $c\,\vec{u} \in \left[\!\left[(C\,\vec{v})\left[\overrightarrow{u/_x}\right]\right]\!\right]$, and by definition of the interpretations of products, $c \in \left[\!\left[\overrightarrow{(x : A)} \Rightarrow C\,\vec{v}\right]\!\right]$.

□

6.6 Dependency pairs

Dependency pairs is a generalisation to the case of rewriting of the notion of recursive calls, since a dependency pair $f\,\vec{l} > g\,\vec{m}$ simply states that one of the rule defining f calls g and the arguments of f and g are $\vec{l}$ and $\vec{m}$ respectively.

It was introduced by Arts and Giesl [AG00] as a complete technique to prove termination of first-order rewriting systems, since a first-order rewriting relation is terminating if and only if there are no infinite sequences of dependency pairs interleaved with reductions in the arguments. The termination condition given by Arts and Giesl's theorem is a necessary and sufficient condition, hence it is undecidable. Several "processors" were built to verify this condition and the dependency pairs evolved into a complete "framework" [Thi07]. Today, all state-of-the-art first-order termination checkers use the dependency pair framework.

Several versions of higher-order dependency pairs have been introduced. A "dynamic" version [KvR12] takes variable application into account, whereas the "static" version [Bla06, KS07, FK19] excludes them, at the price of small restrictions on the class of considered systems.

All those works are in a simply-typed context. As far as I know, in [BGH19], I introduced with my advisors the first definition of dependency pairs in presence of dependent types. This version is an extension of the "static" dependency pairs to this context.

Definition 6.1.1 (Dependency pairs)**.** *Let $f\vec{l} > g\vec{m}$ if there is a rule $f\vec{l} \hookrightarrow r \in \mathcal{R}$, such that $g \in \mathcal{F}$, $g\,\vec{m}$ occurs in r, g is the head of the left-hand side of a rewriting rule, $|\vec{m}| \leqslant \mathrm{ar}(g)$ and if $|\vec{m}| < \mathrm{ar}(g)$ then $\vec{m}$ are all the arguments to which g is applied (ie. $\vec{m}$ are the arguments to which g is applied truncated to the arity of g).*

Example 6.6.2. *One can list the dependency pairs generated from the signature given in Example 5.1.7.*

```
A:                          El (arrow a b) > El a
B:                          El (arrow a b) > El b
C:                               (s x) + y > x + y
D:                               x + (s y) > x + y
E:                             x + (y + z) > (x + y) + z
F:                             x + (y + z) > x + y
G:         append (s m) (cons m a l1) n l2 > m + n
H:         append (s m) (cons m a l1) n l2 > append m l1 n l2
I: append m l1 (n + p) (append n l2 p l3) >
                                    append (m + n) (append m l1 n l2) p l3
J: append m l1 (n + p) (append n l2 p l3) > m + n
K: append m l1 (n + p) (append n l2 p l3) > append m l1 n l2
L:                map f (s n) (cons n a l) > map f n l
M:           len_fil f (s p) (cons p a l) > len_fil_aux (f x) f p l
N:                 len_fil_aux true f p l > len_fil f p l
O:                len_fil_aux false f p l > len_fil f p l
P:             filter f (s p) (cons p a l) > fil_aux (f x) f p a l
Q:                   fil_aux false f p a l > filter f p l
R:                    fil_aux true f p a l > len_fil f p l
S:                    fil_aux true f p a l > filter f p l
T:   len_fil f (p + q) (append p l1 q l2) >
                                      (len_fil f p l1) + (len_fil f q l2)
U:   len_fil f (p + q) (append p l1 q l2) > len_fil f p l1
V:   len_fil f (p + q) (append p l1 q l2) > len_fil f q l2
W:    filter f (p + q) (append p l1 q l2) >
                                   append (len_fil f p l1) (filter f p l1)
                                          (len_fil f q l2) (filter f q l2)
X:    filter f (p + q) (append p l1 q l2) > len_fil f p l1
Y:    filter f (p + q) (append p l1 q l2) > filter f p l1
Z:    filter f (p + q) (append p l1 q l2) > len_fil f q l2
A':   filter f (p + q) (append p l1 q l2) > filter f q l2
```

Since `cons` and `s` are not defined, the applications of them in the right-hand side of rules do not create new dependency pairs, so neither `map f (s n) (cons n a l) > cons n (f a) (map f n l)`, nor `(s x) + y > s (x + y)` are dependency pairs.

Definition 6.6.3 (Instantiated call relation). *$f\,t_1 \dots t_{\mathrm{ar}(f)} \mathrel{\widetilde{\succ}} g\,u_1 \dots u_{\mathrm{ar}(g)}$ if there are a dependency pair $f\,l_1 \dots l_i > g\,m_1 \dots m_j$ and a substitution σ such that for all $k \leqslant i$, $t_k \rightsquigarrow^* l_k\sigma$ and for all $k \leqslant j$, $m_k\sigma = u_k$.*

If f or g are under-applied in the dependency pair, then we add arbitrary missing arguments. If those arguments were not added, it would not be possible to chain the calls, and $\widetilde{\succ}$ would be well-founded even for non-terminating system like:

```
symbol f : ℕ ⇒ ℕ ⇒ ℕ.          symbol g : ℕ ⇒ ℕ.
[] f 0 ↪ g.                    [x] g x ↪ f x x.
```

With $\widetilde{\succ}$ as defined in Definition 6.6.3, there are infinite sequences of calls for this example, like $f\,0\,0 \mathrel{\widetilde{\succ}} g\,0 \mathrel{\widetilde{\succ}} f\,0\,0 \mathrel{\widetilde{\succ}} \dots$. This sequence is infinite only thanks to the liberty to add the argument 0, to derive that $f\,0\,0 \mathrel{\widetilde{\succ}} g\,0$ from the dependency pair $f\,0 > g$.

A similar situation occurs with the non terminating system:

```
[x,y] app x y ↪ x y.           [x,y] f x y ↪ app (f x) y.
```

where the only dependency pairs are `f x y > app (f x) y` and `f x y > f x`

Now, that we have defined the call relation, to recover the usual correctness of dependency pairs, we would like to show that if $\widetilde{\succ}$ is well-founded, so is $\rightsquigarrow$.

Thanks to Theorem 6.4.3, to get this result, it suffices to show that if $\widetilde{\succ}$ is well-founded, then Θ is valid. Unfortunately, it is not always the case, and it requires another hypothesis, the well-structuration of the rewrite system, which intuitively states that the system is divided in layers, and typing the definition of a function only requires to use the symbols which are in the previous layers.

Those layers are defined by the order on symbols of the signature $\sqsubset$.

Definition 6.6.4 (Symbol order). *Let $\sqsupseteq$ be the smallest pre-order on $\mathcal{F}$ such that*

- *if g occurs in $\Theta(f)$, then $f \sqsupseteq g$,*
- *if there is a rule $f\,\vec{l} \hookrightarrow r$, with g occurring in r, then $f \sqsupseteq g$.*

We denote by $\sqsubset$ the strict part of $\sqsubseteq$.

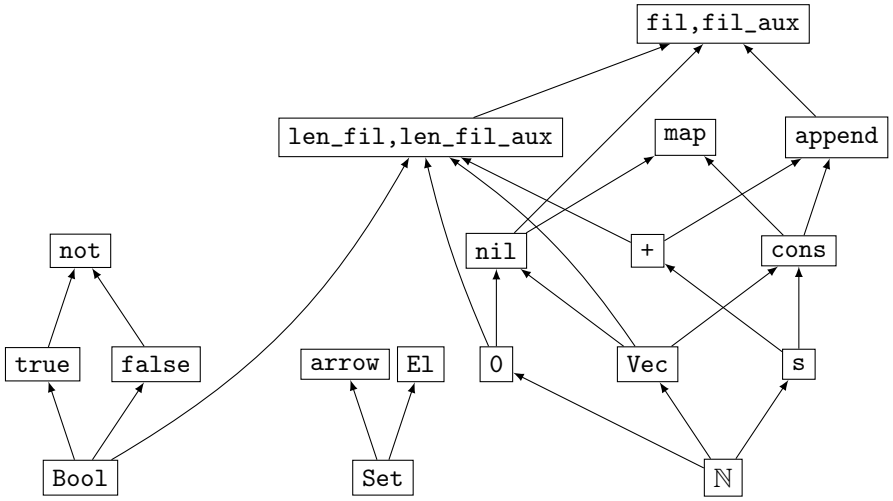
Example 6.6.5. *One can again consider the signature of Example 5.1.7.*
The comparisons related to the typing of symbols are:

`arrow`	$\sqsupseteq$ `Set`	`Vec`	$\sqsupseteq$ $\mathbb{N}$
`El`	$\sqsupseteq$ `Set`	`nil`	$\sqsupseteq$ `Vec`, `0`
`true`	$\sqsupseteq$ `Bool`	`cons`	$\sqsupseteq$ $\mathbb{N}$, `Vec`, `s`
`false`	$\sqsupseteq$ `Bool`	`append`	$\sqsupseteq$ $\mathbb{N}$, `Vec`, `+`
`not`	$\sqsupseteq$ `Bool`	`map`	$\sqsupseteq$ $\mathbb{N}$, `Vec`
`0`	$\sqsupseteq$ $\mathbb{N}$	`len_fil`	$\sqsupseteq$ `Bool`, $\mathbb{N}$, `Vec`
`s`	$\sqsupseteq$ $\mathbb{N}$	`len_fil_aux`	$\sqsupseteq$ `Bool`, $\mathbb{N}$, `Vec`
`+`	$\sqsupseteq$ $\mathbb{N}$	`filter`	$\sqsupseteq$ `Bool`, $\mathbb{N}$, `Vec`, `len_fil`
		`fil_aux`	$\sqsupseteq$ `Bool`, $\mathbb{N}$, `Vec`, `len_fil_aux`

The one related to the rewriting rules are:

not	$\sqsupseteq$ true, false	len_fil	$\sqsupseteq$ 0, len_fil_aux, +
+	$\sqsupseteq$ s	len_fil_aux	$\sqsupseteq$ s, len_fil
append	$\sqsupseteq$ +, cons	filter	$\sqsupseteq$ nil, fil_aux, append, len_fil
map	$\sqsupseteq$ nil, cons	fil_aux	$\sqsupseteq$ filter, cons, len_fil

This precedence can be summed up in the following diagram, where symbols in the same box are equivalent:



Condition 6.6.6 (Well-foundedness of $\sqsubset$)**.** *From now on, we assume that $\sqsubset$ is well-founded.*

It should be noted that this is always the case if $\mathcal{F}$ is finite, an hypothesis which is always fulfilled in practise, since one cannot write an infinite DEDUKTI file in the finite memory of a computer. Furthermore, this finiteness hypothesis is require in the last section of this chapter (Section 6.8), and so is an hypothesis of the final theorem (Theorem 6.9.1).

Definition 6.6.7 (Ordered typing)**.** *Let $f \in \mathcal{F}$. Let $\vdash_{\sqsubset f}$ be the relation defined as $\vdash$ of Section 4.2, but where the rule (sig) is replaced by:*

$$\textit{(sig')}\ \frac{\Gamma \vdash_{\sqsubset f} \Theta(g) : s(g) \qquad g \sqsubset f}{\Gamma \vdash_{\sqsubset f} g : \Theta(g)}$$

Definition 6.6.8 (Computability closure)**.** *Let $f\,\vec{l}$ be the left-hand side of a rewriting rule. Let $\vdash_{f\vec{l}}$ be the relation defined by:*

$$(\text{ax})\ \frac{}{[] \vdash_{f\vec{l}} \star : \square}$$

$$(\text{var})\ \frac{\Gamma \vdash_{\sqsubset f} A : s}{\Gamma, x : A \vdash_{f\vec{l}} x : A}\ x \notin \text{dom}(\Gamma)$$

$$(\text{weak})\ \frac{\Gamma \vdash_{\sqsubset f} A : s \qquad \Gamma \vdash_{f\vec{l}} b : B}{\Gamma, x : A \vdash_{f\vec{l}} b : B}\ x \notin \text{dom}(\Gamma)$$

$$(\text{prod})\ \frac{\Gamma \vdash_{f\vec{l}} A : \star \qquad \Gamma, x : A \vdash_{f\vec{l}} B : s}{\Gamma \vdash_{f\vec{l}} (x : A) \Rightarrow B : s}$$

$$(\text{abstr})\ \frac{\Gamma \vdash_{\sqsubset f} (x:A) \Rightarrow B : s \qquad \Gamma, x:A \vdash_{f\vec{l}} t : B}{\Gamma \vdash_{f\vec{l}} \lambda\,(x:A)\,.t : (x:A) \Rightarrow B}$$

$$(\text{app})\ \frac{\Gamma \vdash_{f\vec{l}} t : (x:A) \Rightarrow B \qquad \Gamma \vdash_{f\vec{l}} u : A \qquad \Gamma \vdash_{\sqsubset f} (x:A) \Rightarrow B : s}{\Gamma \vdash_{f\vec{l}} t\,u : B\,[u/_x]}$$

$$(\text{conv})\ \frac{\Gamma \vdash_{f\vec{l}} t : A \qquad \Gamma \vdash_{\sqsubset f} A : s \qquad \Gamma \vdash_{\sqsubset f} B : s}{\Gamma \vdash_{f\vec{l}} t : B}\ A \downarrow B$$

$$(\text{dp})\ \frac{\vdash_{\sqsubset f} \Theta(g) : s(g) \qquad \overrightarrow{\Gamma \vdash_{f\vec{l}} u : U\gamma}}{\Gamma \vdash_{f\vec{l}} g\,\vec{u} : V\gamma}\ \begin{cases} \Theta(g) = \overrightarrow{(x:U)} \Rightarrow V \\ \gamma = \left[\overrightarrow{u/_x}\right] \\ g\,\vec{u} < f\,\vec{l} \end{cases}$$

$$(\text{const})\ \frac{\Gamma \vdash_{\sqsubset f} \Theta(g) : s(g)}{\Gamma \vdash_{f\vec{l}} g : \Theta(g)}\ \begin{cases} g \text{ is not defined by a rewriting rule} \\ g \sqsubset f \end{cases}$$

This computability closure is similar to the usual typing rules, with restrictions on the symbols of the signature which can appear, as explained by the following lemma.

Lemma 6.6.9. *For $f, g \in \mathcal{F}$, $\vec{l}$, Γ, t and T, such that g occurs in t*

- *if $\Gamma \vdash_{f\vec{l}} t : T$ then $g \sqsubseteq f$,*
- *if $\Gamma \vdash_{\sqsubset f} t : T$ then $g \sqsubset f$.*

Proof. By induction on the proof tree. □

Furthermore, the application of defined symbols of the signature is restricted, since even if there a rule (*app*), it is not possible to introduce a defined symbol without going through the rule (*dp*) which includes already several applications. This means this system is crafted to type check right-hand sides of rewriting rules and not arbitrary terms.

However, it is crucial to note that this restriction only acts on $\vdash_{f\vec{l}}$. Whenever we are trying to sort a type, we use the relation $\vdash_{\sqsubset f}$, which does not restrict the application of symbols of the signature. This possibility to introduce symbols without going through dependency pairs is even required for types. Otherwise, it would not be possible to have a term living in a type defined by rewriting in the right-hand side of rules.

Example 6.6.10. *One could define an function* `return_zero` *of variable arity which always ignores its inputs and outputs 0. Such a function has a type of the shape* `(n:ℕ)⇒⋆`*, where the first argument is a natural number, telling us how many elements will be ignored.*

```
symbol D : ℕ ⇒ ⋆.
[]  D 0     ↪ ℕ.
[n] D (s n) ↪ ℕ ⇒ D n.

symbol return_zero : (n : ℕ) ⇒ D n.
[]  return_zero 0     ↪ 0.
[n] return_zero (s n) ↪ λx, return_zero n.
```

In this example, the only dependency pair are `D (s n) > D n` *and* `return_zero (s n) > return_zero n`. *but the expected type of* `return n` *in the right-hand side of the last rule is* `D n`. *Since there is no dependency pair* `return_zero (s n) > D n` *and* `D` *is defined by rewriting rule, neither* (*dp*) *not* (*const*) *can be applied, so there is no hope to prove* $\Gamma \vdash_{\texttt{return\_zero n}} D\, n : \star$. *But it is possible to prove* $n : \mathbb{N} \vdash_{\sqsubset \texttt{return\_zero}} D\, n : \star$, *which is required to type-check the right-hand side of the last rule.*

Definition 6.6.11 (Well-structured rule)**.** *A rule* $f\,\vec{l} \hookrightarrow r$ *with* $\Theta(f) = \overrightarrow{(x : T)} \Rightarrow U$, $\Gamma = \overrightarrow{(x : T)}$ *and* $\pi = \left[\overrightarrow{l/_x}\right]$ *is* well-structured *if there is a context* Δ *such that :*

- $\Delta \vdash_{f\,\vec{l}} r : U\pi$,
- *for all* σ *such that* $\pi\sigma \vDash \Gamma$, *we have* $\sigma \vDash \Delta$.

Theorem 6.6.12. Θ *is valid if* $\widetilde{\succ}$ *terminates on* $\mathbb{U}$ *and all rules in* $\mathfrak{R}$ *are well-structured.*

Proof. We will prove this by induction on $\sqsubset$. Let $f \in \mathcal{F}$, with $\Theta(f) = \overrightarrow{(x : T)} \Rightarrow T'$, $\vdash \Theta(f) : s(f)$ and $\Theta(f) \in [\![s(f)]\!]$ be such that for all $g \sqsubset f$, such that $\vdash \Theta(g) : s(g)$ and $\Theta(g) \in [\![s(g)]\!]$, $g \in [\![\Theta(g)]\!]$.

We want to prove that $f \in \left[\!\!\left[\overrightarrow{(x : T)} \Rightarrow T'\right]\!\!\right]$ which by definition of the interpretation of a product is equivalent to show that $f\,\vec{t} \in [\![T'\gamma]\!]$ for any $\gamma = \left[\overrightarrow{t/_x}\right] \models \overrightarrow{(x : T)}$. Hence, we take an $f\,\vec{t} \in \mathbb{U}$. Let $\gamma = \left[\overrightarrow{t/_x}\right]$.

Since $\rightsquigarrow_{arg}$ and $\widetilde{\succ}$ terminate on $\mathbb{U}$ and $(\rightsquigarrow_{arg} \widetilde{\succ}) \subseteq \widetilde{\succ}$, we have that $\rightsquigarrow_{arg} \cup \widetilde{\succ}$ terminates on $\mathbb{U}$.

We now prove that, $f\,\vec{t} \in [\![T'\gamma]\!]$ by induction on $\rightsquigarrow_{arg} \cup \widetilde{\succ}$.

- If $f \in \mathcal{F}_o \setminus \mathcal{C}_o$ or $f \in \mathcal{F}_\mathcal{T} \setminus \mathcal{C}_\mathcal{T}$, $f\,\vec{t}$ is neutral. Hence, it suffices to prove that, for all u such that $f\,\vec{t} \rightsquigarrow u$, we have $u \in [\![T'\gamma]\!]$. There are two cases:
  - $u = f\,\vec{u}$ with $f\,\vec{t} \rightsquigarrow_{arg} f\,\vec{u}$. Then, we can conclude by induction hypothesis.
  - There are $f l_1 \dots l_k \hookrightarrow r \in \mathcal{R}$ and σ such that $u = (r\sigma)\, t_{k+1} \dots t_n$ and, for all $i \in \{1, \dots, k\}$, $t_i = l_i\sigma$. Let $\pi = \left[\overrightarrow{l_i/_{x_i}}\right]$. Since for all i, $x_i\pi\sigma = l_i\sigma = t_i \in [\![T_i\gamma]\!]$, we have that $\pi\sigma \vDash \overrightarrow{(x_i : T_i)}$. Since all rules are well-structured, there is Δ such that $\sigma \vDash \Delta$ and $\Delta \vdash_{f\vec{l}} r : ((x_{k+1} : T_{k+1}) \Rightarrow \dots \Rightarrow (x_n : T_n) \Rightarrow T') \left[l_1/_{x_1}, \dots, l_k/_{x_k}\right]$.
    We now prove the pseudo-adequacy result that for all u and U, if $\Delta' \vdash_{f\vec{l}} u : U$, $\sigma \vDash \Delta'$, then $u\sigma \in [\![U\sigma]\!]$, by induction on the structure of the derivation of $\Delta' \vdash_{f\vec{l}} u : U$. The proof is the same as for Theorem 6.4.3 except the case (sig) replaced by (dp) and (const).
    (dp) Let g and $(u_1, \dots, u_j)$ be such that the conclusion is the typing of $g\,\vec{u}$. Let $\Theta(g) = \overrightarrow{(x : V)} \Rightarrow W$, and $\gamma = \left[\overrightarrow{u/_x}\right]$. In this case, for all i, we have $\Delta' \vdash_{f\vec{l}} u_i : V_i\gamma$. By the inner induction hypothesis, $u_i\sigma \in [\![V_i\gamma\sigma]\!]$, so $\gamma\sigma \vDash \overrightarrow{(x : V)}$.
    For every $\vec{v} \in \left[\!\!\left[V_{j+1}\left[\overrightarrow{u\sigma/_x}\right]\right]\!\!\right] \times \dots \times \left[\!\!\left[V_n\left[\overrightarrow{u\sigma/_x}, v_{j+1}/_{x_{j+1}}, \dots, v_{n-1}/_{x_{n-1}}\right]\right]\!\!\right]$, $g\ (\vec{u}\sigma)\ \vec{v} \in \mathbb{U}$ and $f\,\vec{t} \widetilde{\succ} g\ (\vec{u}\sigma)\ \vec{v}$. Therefore, by the outer induction hypothesis, $g\ (\vec{u}\sigma)\ \vec{v} \in [\![W\gamma\sigma]\!]$.
    By definition of the interpretation of products:

$$g\ (\vec{u}\sigma) \in [\![((x_{j+1} : V_{j+1}) \Rightarrow \dots \Rightarrow (x_n : V_n) \Rightarrow W)\,\gamma\sigma]\!]\,.$$

(const) By hypothesis, $g \sqsubset f$. We then have $g \in [\![\Theta(g)]\!]$ by hypothesis. Since $\Theta(g)$ is closed, $g \in [\![\Theta(g)\sigma]\!]$.

By pseudo-adequacy, we can conclude from

$$\Delta \vdash_{fl} r : ((x_{k+1} : T_{k+1}) \Rightarrow \ldots \Rightarrow (x_n : T_n) \Rightarrow T') \left[l_1/_{x_1}, \ldots, l_k/_{x_k} \right],$$

that:

$$r\sigma \in \left[\!\!\left[((x_{k+1} : T_{k+1}) \Rightarrow \ldots \Rightarrow (x_n : T_n) \Rightarrow T') \left[l_1/_{x_1}, \ldots, l_k/_{x_k} \right] \sigma \right]\!\!\right]$$

Which by definition of the interpretation of a product implies that $(r\sigma)\, t_{k+1} \ldots t_n = u \in [\![T'\gamma]\!]$, since $f\,\vec{t} \in \mathbb{U}$ ensures that all the t_i's are in the interpretation of their types.

- If $f \in \mathcal{C}_o$, then $T' = C\,\vec{u}$ with $C \in \mathcal{C}_\mathcal{T}$.

  By Lemma 6.3.10, we know that it is sufficient to prove that $f\,\vec{t} \in \mathrm{SN}$ and for every $c\,\vec{v}$ such that $f\,\vec{t} \rightsquigarrow^* c\,\vec{v}$ with $c \in \mathcal{C}_o$, $\Theta(c) = \overrightarrow{(x : T)} \Rightarrow C\,\vec{w}$ and $|u| \geqslant \mathrm{ar}(g)$, for all $j \in \mathrm{Acc}(c)$, v_j is in the interpretation of the expected type.

  In the case of non-constructors, we proved that all the direct reduct of $f\,\vec{t}$ are in the interpretation of the expected type. Even if this property is not sufficient to conclude that $f \in [\![\Theta(f)]\!]$ when $f \in \mathcal{C}_o$, this proof still holds in this case and ensures us that $f\,\vec{t} \in \mathrm{SN}$.

  Three cases can occur:

  - $c\,\vec{v} = f\,\vec{t}$, then by definition of $\mathbb{U}$, every t_i is in the interpretation of the expected type;
  - The first reduction to go between $f\,\vec{t}$ and $c\,\vec{v}$ occurs in an argument. We can conclude by induction hypothesis on $\rightsquigarrow_{arg} \cup \widetilde{\succ}$;
  - Otherwise, the first reduction occurs in the head. We can do as in the case of non-constructors to prove that the direct reduct of $f\,\vec{t}$ is in $[\![T'\gamma]\!]$.

    We conclude thanks to the stability by reduction of the interpretations that $c\,\vec{v}$ is also in $[\![T'\gamma]\!]$.

- If $f \in \mathcal{C}_\mathcal{T}$, then all reductions occur in arguments and we can conclude by induction hypothesis, since we have the guarantee that $f\,\vec{t} \in \mathrm{SN}$ and does not reduce to a product. □

Corollary 6.6.13 (Adequacy)**.** *Let σ be a substitution, Θ a typing map, Γ an environment and t and T two terms.*

If $\widetilde{\succ}$ terminates on $\mathbb{U}$ and all rules in $\mathfrak{R}$ are well-structured, $\Gamma \vdash t : T$ and $\sigma \vDash \Gamma$, then $t\sigma \in [\![T\sigma]\!]$.

Proof. This a direct consequence of Theorem 6.4.3 and Theorem 6.6.12. □

We replaced the undecidable hypothesis of validity of the typing map by two hypotheses, the well-structuration of the rewriting rules and the well-foundedness of the call relation $\widetilde{\succ}$. This does not seem to be a real progress, since it still is an undecidable termination property. But we will see in the next two sections how to verify those hypotheses in practice. In Section 6.7, we define *Accessible Variables Only* rules, which are all well-structured, and in Section 6.8, we present *size-change termination*, a criterion (one would say processor [Thi07, FK19]) to verify the well-foundedness of $\widetilde{\succ}$.

6.7 Accessible Variables Only Rules

Well-structuration was both an hypothesis of typability of the right-hand side of the rule, and of interpretability of the variables occurring in it.

Since accessible arguments (Definition 6.1.6) were defined to guarantee the preservation of interpretability by the subterm operation, it is quite natural to reuse the notion to craft a criterion of interpretability of the variables occurring in the right-hand side of a rewriting rule.

Since Acc was defined to identify the arguments which can be accessed, to identify the variables which are in the interpretation of their types, it can happen that one has to go several time under a constructor, this is why we define the function AccPosCstr.

Furthermore, our aim is to study the rules defining f to ensure that f is in the interpretation of its type. Hence, we assume that all the direct arguments of f are themselves in the interpretation of the expected type, so accessibility is not an issue for terms which are directly under the head of the rule, but only for those which are nested in an argument. Hence, we define AccPosHd which assumes the interpretability of all the direct arguments of the head symbol.

Definition 6.7.1 (Accessible position)**.** *Given t and T, we define the function* AccPosCstr *by:*

$$\begin{array}{lll} \textit{If} & \left\{\begin{array}{l} t = c\,u_1 \dots u_k \\ c \in \mathcal{C}_o \\ \Theta(c) = (y_1 : U_1) \Rightarrow \dots \Rightarrow (y_r : U_r) \Rightarrow C\,\vec{v} \\ \exists W_{k+1}, \dots, W_r, \vec{v}',\ \textit{such that } T \rightsquigarrow^* (y_{k+1} : W_{k+1}) \Rightarrow \dots \Rightarrow (y_r : W_r) \Rightarrow C\,\vec{v}' \end{array}\right. \\ \textit{then:} & \text{AccPosCstr}(t,T) = \{\varepsilon\} \cup \bigcup_{i \in \text{Acc}(c)} \left\{ i.p \;\middle|\; p \in \text{AccPosCstr}\left(u_i, U_i\left[\overrightarrow{u/y}\right]\right)\right\} \\ \textit{else:} & \text{AccPosCstr}(t,T) = \{\varepsilon\} \end{array}$$

and the function:

$$\text{AccPosHd} : (f\,\vec{t}) \mapsto \bigcup_i \{i.x \mid x \in \text{AccPosCstr}(t_i, T_i)\} \;\; \textit{with } \Theta(f) = \overrightarrow{(x:T)} \Rightarrow U$$

The type conditions might seem quite surprising, since the W_i's and the U_i's are not related, neither are $\vec{v}$ and $\vec{v}'$. Indeed, one could expect that $(y_{k+1} : W_{k+1}) \Rightarrow \dots \Rightarrow (y_r : W_r) \Rightarrow C\,\vec{v}' = ((y_{k+1} : U_{k+1}) \Rightarrow \dots \Rightarrow (y_r : U_r) \Rightarrow C\,\vec{v}')\left[\overrightarrow{u_i/y_i}\right]$, but we do not need such a restriction:

- since the indices after k do not correspond to any subterm, it has no meaning to wonder if one can access them,
- and the interpretation of $C\,\vec{v}'$ is independent of $\vec{v}'$.

Furthermore, it would be useful to be loose on the type condition, if one wants to allow patterns to be ill-typed, which is a quite frequent situation with dependent types. Indeed, the rule

```
[f,n,a,l]map f (s n) (cons n a l) ⟶ cons n (f a) (map f n l)
```

presented in Example 5.1.7 is not left-linear, which is a trouble in an implementation, since it requires to do a conversion test each time the rule could be used[2]. But linear versions of the rule, like

```
[f,m,n,a,l] map f (s m) (cons n a l) ⟶ cons n (f a) (map f n l)
```

[2]Non-linear rules are also a trouble for confluence checking, but since we assume local confluence, this does not impact us.

or even

$$\texttt{[f,p,n,a,l] map f p (cons n a l)} \hookrightarrow \texttt{cons n (f a) (map f n l)},$$

preserves typing since every well-typed instance of the left-hand side $(\texttt{map}\, f\ (\texttt{s}\, m)\ (\texttt{cons}\, n\, a\, l))\,\sigma$, is such that $n\sigma = m\sigma$ ($(\texttt{s}\, n)\,\sigma = p\sigma$ in the second variant).

Saillard [Sai15] proposed an algorithm to verify the type preservation of rewriting rules in the $\lambda\Pi$-calculus modulo rewriting, which was then enriched by Blanqui [Bla20]. Both of those algorithms construct a set of convertibility constraint while performing type inference of the left-hand-sides of rules, and then use those constraints while type-checking the right-hand sides. It would be possible to follow their methods to construct not only a typed context (i.e. infer a type for each variable), but also a set of constraints. However, since the type-checking of rules is not our purpose, we do a simple type inference for variables, meaning that the rules presented in the previous paragraph are not featured in the current definition of AVO rules (even if they are well-structured). Hence, the looseness of the typing constraints in the definition of AccPosCstr are paving the way to this extension of the definition of AVO.

Well-structuration is both an interpretability and a typability condition. The function AccPosHd was designed to deal with the interpretability. For the typing condition, one has to define a function to infer the type of the variables. The function InferType takes position p and a term t to infer the type of the term at position p in t. The type of an argument of an application does not only depends of the head symbol of this application, but also of the previous arguments.

Definition 6.7.2 (Inferred Type at a Position)**.** *We now define the functions:*

$$\text{InferType} : (p, f\,\vec{t}) \mapsto \begin{cases} T_i\left[\overrightarrow{t/y}\right] & \text{if } p = i \text{ and } \Theta(f) = \overrightarrow{(y:T)} \Rightarrow U \\ \text{InferType}(p', t_i) & \text{if } p = i.p', p' \neq \varepsilon \text{ and } \Theta(f) = \overrightarrow{(y:T)} \Rightarrow U \end{cases}$$

The inferred type defined is identical to what Blanqui calls "derived type" [Bla05], which is the type inferred from the declared type of the head symbol of applications. Similarly to what Blanqui did, we then use this type to show that is all variables used in rules are accessible, then the rewriting system is well-structured.

We must note here that, even if $f\,\vec{t}$ is a pattern (Definition 4.1.2), $\text{InferType}(p, f\,\vec{t})$ could be ill-defined for some position p in $f\,\vec{t}$. This can only occur if a symbol of the signature is over-applied (i.e. has more arguments than its arity), which is perfectly possible with dependent types. However, constructors cannot be over-applied, since their codomain is not defined by rewriting rules, so cannot reduce to a product. Especially, if $p \in \text{AccPosHd}(f\,\vec{t})$, then $\text{InferType}(p, f\,\vec{t})$ is well-defined.

Definition 6.7.3 (AVO rules)**.** *A rule $f\,l_1 \dots l_n \hookrightarrow r$ is* Accessible Variable Only (AVO) *if*

- *there is a function $\phi : \text{FV}(r) \to \text{AccPosHd}(f\,\vec{l})$ selecting an occurrence of each free variable, meaning that $(f\,\vec{l})|_{\phi(x)} = x$, for all $x \in \text{FV}(r)$.*
- *$\Delta_r \vdash r : T_r$, where*
  - *$\Delta_r = \left\{ x : \text{InferType}(\phi(x), f\,\vec{l}) \;\middle|\; x \in \text{FV}(r) \right\}$ ordered by the alphabetical order on the positions $\phi(x)$.*
  - *$T_r = U\left[\overrightarrow{l/x}\right]$, where $\Theta(f) = (x_1 : T_1) \Rightarrow \dots \Rightarrow (x_n : T_n) \Rightarrow U$.*

With dependent types, the contexts are ordered, since a variable occurring earlier in the context can appear in the type of another variable. With InferType, one has the guarantee that the only instantiations of types which are performed, are done with terms occurring at a position which is more at the left in $f\,\vec{l}$ than the one we are interested in. Hence, we have the guarantee that ordering variables from left to right will not create circularities. This is why Δ_r is sorted by the alphabetical order on the position of the selected occurrence of each variable, which is the same as from left to right in $f\,\vec{l}$, since it ensures its well-formedness.

Lemma 6.7.4. *AVO rules are well-structured.*

Proof. Let $f\,\vec{l} \hookrightarrow r$ be an AVO rule with $\Theta(f) = \overrightarrow{(x : T)} \Rightarrow U$, $\Gamma = \overrightarrow{(x : T)}$ and $\pi = \left[\overrightarrow{l/_x}\right]$. Let σ be such that $\pi\sigma \vDash \Gamma$.

In the definition of well-structured rules Definition 6.6.11, we must chose a context Δ. Since we are dealing with AVO rules, we take the context Δ_r.

- We have to prove that, $\sigma \vDash \Delta_r$, meaning that for all $(y : V) \in \Delta_r$, $y\sigma \in [\![V\sigma]\!]$.
  - If $\phi(y) = i$, then, by hypothesis, $x_i\pi\sigma = y\sigma \in [\![T_i\pi\sigma]\!]$. Yet $V = \text{InferType}(i, f\,\vec{l}) = T_i\pi$. So we have indeed $y\sigma \in [\![V\sigma]\!]$.
  - If $\phi(y) = i.p'$ with $p' \neq \varepsilon$, then by hypothesis, there is a $c \in \mathcal{C}_o$ and $u_1 \dots u_k$ such that $l_i = c\,u_1 \dots u_k$ since $\phi(y) \in \text{AccPosHd}(f\,\vec{l})$. Besides, $x_i\pi\sigma = (c\,u_1 \dots u_k)\sigma \in [\![T_i\pi\sigma]\!]$. Since $T_i\pi \rightsquigarrow^* (y_{k+1} : V_{k+1}) \Rightarrow \dots \Rightarrow \left(y_{\text{ar}(c)} : V_{\text{ar}(c)}\right) \Rightarrow C\,\vec{v}$ for some $\vec{v}$, where $\Theta(c) = \overrightarrow{(z : V)} \Rightarrow C\,\vec{v}'$, we know that $(c\,\vec{u})\sigma \in \left[\!\left[\left((y_{k+1} : V_{k+1}) \Rightarrow \dots \Rightarrow \left(y_{\text{ar}(c)} : V_{\text{ar}(c)}\right) \Rightarrow C\,\vec{v}\right)\sigma\right]\!\right]$. Hence, by definition of the interpretation of products and the interpretation of type values Lemma 6.3.10, for $j \in \text{Acc}(c)$ with $j \leqslant k$, we have $u_j\sigma \in [\![V_j\sigma]\!] = \left[\!\left[\text{InferType}(i.j, f\,\vec{l})\sigma\right]\!\right]$. Following p', we finally reach $y\sigma \in [\![V\sigma]\!]$.
- We also have to prove that $\Delta_r \vdash_{f\vec{l}} r : U\pi$. This is the same typing derivation as the one of $\Delta_r \vdash r : U\pi$, but where (sig) followed by (app) is replaced by (dp) since the definition of a dependency pair ensures us that all the application in r are smaller than $f\,\vec{l}$ for the dependency pair order $<$ of Definition 6.6.1.

□

6.8 Size-Change Termination

We now start the last section of this quite long chapter on the termination criterion. As already mentioned, dependency pairs are a framework and several so called "processors" [FK19] were proposed to prove the well-foundedness of the call relation $\widetilde{\succ}$.

Among them, the *size-change principle* was originally proposed as an independent criterion to study termination of a functional first-order language [LJBA01]. David Wahlstedt [Wah07] used it as a weak termination criterion for a dependently typed language with functions defined by dependent pattern matching. The size-change principle was later introduced in the dependency pairs framework for first-order rewriting [TG05]. We propose here an adaptation of this principle to the λΠ-calculus modulo rewriting.

The original principle of size-change termination is to track the variation of the "size" of the arguments through recursive calls. Since dependency pairs are the analogous of recursive calls in a context with rewriting, we will have to track the evolution of the size of arguments in dependency pairs.

The main ingredient to define size-change termination is to have a representation of the evolution of the size at each call. In the original article [LJBA01], Lee, Jones and Ben-Amram used labeled bipartite graphs. We will prefer the presentation of Hyvernat and Raffalli using matrices [HR10]. Even if they do not call it "size-change termination", Abel and Altenkirch [AA02] already used a presentation with matrix for the size-change criterion with the structural ordering, in the case of functions defined by dependent pattern matching.

Definition 6.8.1 (Size Matrix)**.** *To a dependency pair $f\, l_1 \dots l_p > g\, m_1 \dots m_q$, We associate the matrix $(a_{i,j})_{i \leq \mathrm{ar}\, f, j \leq \mathrm{ar}\, g}$ where:*

- *if $l_i \rhd^+_{acc} m_j$, then $a_{i,j} = -1$;*
- *if $l_i = m_j$, then $a_{i,j} = 0$;*
- *otherwise $a_{i,j} = \infty$ (in particular if $i > p$ or $j > q$).*

For instance, the matrices associated to the four first dependency pairs of Example 6.6.2 are:

A

El	El
	a
arrow a b	−1

B

El	El
	b
arrow a b	−1

C

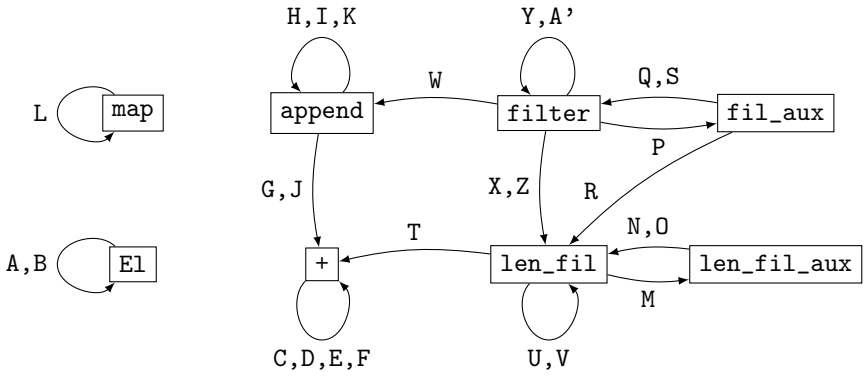
		+	
		x	y
+	s x	−1	∞
	y	∞	0

D

		+	
		x	y
+	x	0	∞
	s y	∞	−1

Definition 6.8.2 (Call graph)**.** *The* call graph $\mathcal{G}(\mathcal{R})$ *associated to $\mathcal{R}$ is the directed labeled graph on the defined symbols of $\mathcal{F}_o \cup \mathcal{F}_\mathcal{T}$ such that there is an edge between f and g if and only if there is a dependency pair $f\, l_1 \dots l_p > g\, m_1 \dots m_q$. This edge is labeled with the size matrix associated to this dependency pair.*

The call-graph associated to Example 6.6.2 is:



Definition 6.8.3 (Size-Change Termination)**.** $\mathcal{R}$ *is* size-change terminating (SCT) *if, in the transitive closure of $\mathcal{G}(\mathcal{R})$ (using the min-plus semi-ring to multiply the matrices labeling the edges), all idempotent matrices labeling a loop have at least one -1 on the diagonal.*

For instance, the multiplication of the matrices C and D given above is:

$$C \times D = \begin{pmatrix} \min(-1+0, \infty+\infty) & \min(-1+\infty, \infty+-1) \\ \min(\infty+0, 0+\infty) & \min(\infty+\infty, 0+(-1)) \end{pmatrix} = \begin{pmatrix} -1 & \infty \\ \infty & -1 \end{pmatrix}$$

This matrix is idempotent and has a -1 on the diagonal.

Like in the original size-change principle, we do not consider information more precise than "the argument is not comparable" or "the argument decreased", especially the knowledge that an argument decreased twice is forgotten, so $-1 + (-1) = -1$. Hyvernat proposed to have more accurate information on size variations for a functional language with functions defined by pattern matching [Hyv14].

Lemma 6.8.4. *$\widetilde{\succ}$ terminates on $\mathbb{U}$ if $\mathcal{F}_\mathcal{T} \cup \mathcal{F}_o$ is finite and $\mathfrak{R}$ is SCT.*

Proof. Suppose that there is an infinite sequence $\chi = f_1\,\vec{t}_1 \mathrel{\widetilde{\succ}} f_2\,\vec{t}_2 \mathrel{\widetilde{\succ}} \dots$. Then, there is an infinite path in the call graph going through nodes labeled by $f_1, f_2, \dots$. Since $\mathcal{F}_\mathcal{T} \cup \mathcal{F}_o$ is finite, there is a symbol g occurring infinitely often in this path. So, there is an infinite sequence $(g\,\vec{u}_1), (g\,\vec{u}_2), \dots$ extracted from χ. Hence, for every $i, j \in \mathbb{N}^*$, there is a matrix in the transitive closure of the graph which labels the loops on g corresponding to the relation between $\vec{u}_i$ and $\vec{u}_{i+j}$. The number of such matrices are finite, since the algebra considered only has 3 elements, and the width of the matrices is bounded by the maximal arity of the finitely many symbols of the signature,

By Ramsey's theorem[3] [Ram30], there is an infinite sequence ϕ_i and a matrix M such that M corresponds to all the transitions $(g, \vec{u}_{\phi_i}), (g, \vec{u}_{\phi_j})$ with $i \neq j$. M is idempotent, since $(g, \vec{u}_{\phi_i}), (g, \vec{u}_{\phi_{i+2}})$ is labeled by M^2 by definition of the transitive closure and by M due to Ramsey's theorem, so $M = M^2$. Since, by hypothesis $\mathcal{R}$ satisfies SCT, there is a j such that $M_{j,j}$ is -1. So, for all i, $u^{(j)}_{\phi_i} (\rightsquigarrow^* \triangleright_{acc})^+ u^{(j)}_{\phi_{i+1}}$, which contradicts the fact that $(\rightsquigarrow_{arg} \cup \triangleright_{acc})$ is well-founded on $\mathbb{U}$. □

6.9 Final Criterion

We can then combine all the results of this chapter to craft our final termination criterion:

Theorem 6.9.1. *$\rightsquigarrow = (\rightsquigarrow_\beta \cup \rightsquigarrow_\mathcal{R})$ terminates on terms typable in $\lambda\Pi$ modulo rewriting if*

- *$\rightsquigarrow$ is locally confluent,*
- *$\mathcal{F}_\mathcal{T} \cup \mathcal{F}_o$ is finite,*
- *$\mathfrak{R}$ is AVO,*
- *$\mathfrak{R}$ is size-change terminating for $\triangleright_{acc}$.*

Proof. Since all the interpretations are reducibility candidates (see Section 6.3), to show that all typable terms are strongly normalizing, it suffices to show that they all belong to the interpretation of their types. By Theorem 6.4.3, it suffices to show that the typing map Θ is valid. By Theorem 6.6.12, the validity of the typing map is a consequence of the well-foundedness of $\widetilde{\succ}$ and the well-structuration of $\mathfrak{R}$. By Lemma 6.7.4, AVO rules are well-structured and by Lemma 6.8.4 the well-foundedness of $\widetilde{\succ}$ is a consequence of the finiteness of the signature and the size-change termination. □

[3] Let X be an infinite denumerable set. Let n and c be natural numbers. We denote by $X^{(n)}$ the subsets of X of size n. For all colorings of $X^{(n)}$ with c different colours, there exists some infinite subset M of X such that the size n subsets of M all have the same colour.

In our case, $n = 2$, X is the set of the $\{g\,\vec{u}_i\}$ and c is the number of matrices, so $3^{\mathrm{ar}(g)^2}$

6.10 Related Works

In [Wah07], Wahlstedt defines a dependently typed programming language, for which he shows the weak normalization. His proof is similar to what is presented here, since he also defines an interpretation, then shows that all well-typed terms are in this interpretation if symbols of the signature are, and he finally obtains this from the well-foundation of a call relation, well-foundation proved by the size-change principle. Our works extends Wahlstedt's one in two directions: we are proving strong normalization (which is the termination of any reduction strategy) instead of weak normalization (existence of a normal form), and we do not require the rewriting system to be orthogonal (linear and non-overlapping), but only local confluence.

In [Bla05], Blanqui proposes a termination criterion for an extension of the Calculus of Constructions with rewriting. This setting is more general than the one we considered in this work, since it features polymorphism, making the interpretation much heavier to define. However, it requires a confluent object-level rewriting system and an orthogonal type-level rewriting system, whereas we only need local confluence. Furthermore, every symbol are equipped by a status, expliciting how to compare its arguments in recursive call. This is much less flexible than using dependency pairs, since the user does not have to declare the status of every symbol and a strict decrease must occur in all the calls.

Walukiewicz-Chrząszcz extends in [WC03] the Higher-Order Recursive Path Ordering (HORPO) [JR07] to the calculus of constructions, obtaining a termination criterion for this setting. However, she does not consider type-level rewriting.

A "framework" for higher-order static dependency pairs was proposed by Kop and Fuhs in [FK19] in a simply-typed setting. Just like us, they construct an interpretation, adapting reducibility candidates, to prove that, if all the variables of the right-hand side are accessible, the termination of the call relation induced by the dependency pairs implies the termination of the original rewriting system.

Abel and Altenkirch in [AA02] uses the size-change principle with the structural ordering [Coq92], to prove the strong normalization of a simply-typed predicative language with polymorphism, inductive types and functions defined by pattern-matching. Since they do not feature dependent types and have a well-identified notion of values, their interpretation is much simpler than ours.

Sized types [HPS96] include the checking for the structural descent in the type-checking, by annotating datatypes with a "size". The termination checking is then delegated to the type-checker, whichs ensures that the size goes down at every recursive call. Most of the sized type systems are not crafted for dependent setting [BGR08, AP16], however several version of them exist in a dependently typed setting [Xi02, GS10, AVW17, Bla18].

Chapter 7

SizeChange Tool: An Automatic Termination Prover for the λΠ-Calculus Modulo Rewriting

SizeChange Tool [Gen18] is a fully automated termination checker for the λΠ-calculus modulo rewriting, implementing the results presented in Chapter 6.

Its development became essential as various libraries were encoded in the logical framework Dedukti [ABC+19]. Indeed, to define in Dedukti the logic she wants to reason with, the user provides a set of rewriting rules. However, to ensure that the defined type system has good properties, like logical consistency or decidability, the rules must satisfy some properties: termination, confluence and type preservation.

Many criteria have been created to check termination of first-order rewriting. The dynamism of this research area is illustrated by the numerous tools participating in the various first-order categories of the termination competition [TC]. For higher-order rewriting, criteria have been crafted too, many of them can be found in [Kop12] and a category exists in the competition. However, one can deplore the small number of participants in this category: Only 2 in 2019 and 2020[1], including SizeChange Tool!

This lack of implementations is even more visible for rewriting with dependent types, for which criteria have been developed [Bla05, JL15], but as far as the author knows, none of them have been implemented.

Facing this situation, there were no alternative for a user trying to encode a new logic in Dedukti, but to do an ad-hoc, pen and paper proof of termination, or to check manually the existing criteria. The aim of this work is to provide a simple to use, fully automatic termination checker.

7.1 Implementation and Interaction with the Type Checker

SizeChange Tool takes as input Dedukti files or XTC files, the format of the termination competition [TC]. However, XTC does not include dependent types yet, hence we proposed a backward compatible extension of the format.

[1] There was a third participant in 2018, Second-Order Laboratory (SOL), developed by Hamana and Kikuchi [Ham19], but it only participated in the confluence competition the last two years.

SizeChange Tool performs all the checks stated in Theorem 6.9.1.

Theorem 6.9.1 $\rightsquigarrow = (\rightsquigarrow_\beta \cup \rightsquigarrow_{\mathcal{R}})$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if

- $\rightsquigarrow$ is locally confluent.

  This check is the only one left to the user. However, if she wants to check it automatically, Dedukti offers an export to the confluence competition.

- $\mathcal{F}_\mathcal{T} \cup \mathcal{F}_o$ is finite,

  This hypothesis is guaranteed, since one cannot declare infinitely many symbols in a finite Dedukti or XTC file.

- $\mathfrak{R}$ is AVO.

  This hypothesis is a double one, since to be AVO, one requires both accessibility of all variables used in the right-hand side, and typability of this right-hand side.

  1. The typability of the right-hand side of every rule is simply delegated to Dedukti, once the types of the variables have been inferred. Naturally, to fully benefit from the power offered by the type-checker of Dedukti, this inference of variable types is also delegated to Dedukti;
  2. *The accessibility condition* requires to have a pre-order on type constructors. The user is not asked to provide this order. While analyzing the rules, SizeChange Tool assumes that all variables used in the right-hand side are accessible and accumulates constraints of the form "Type constructor A is strictly greater than type constructor B" or "Type constructor A is greater or equal to type constructor B". With all those constraints, one construct a directed graph which nodes are type constructors and an edge between A and B means "A must be at least greater or equal than B". To check that this relation satisfies all the accumulated constraints, one simply checks that for every constraint "A is strictly greater than B" there is no arrow between A and B in the transitive closure of the graph.

- $\mathcal{R}$ is size-change terminating for $\triangleright_{acc}$.

  Size-change termination requires to analyze every rule in order to extract the dependency pairs. One has then to construct the call graph associated to this set of dependency pairs, meaning, that for each pair of argument (one originating from the left-hand side and the other one from the right-hand side), one must compare them using $\triangleright_{acc}$ (Definition 6.5.1).

  As explained in Section 6.5, $\triangleright_{acc}$ is an undecidable relation, so the subapproximation proposed in Definition 6.5.6 is used.

  To perform size-change termination checking, one must then compute the transitive closure of the call graph and verify the presence of a -1 on the diagonal of every idempotent matrix labeling a loop. The computation of the transitive closure and the analyze of the call graph has been implemented by Lepigre and Raffalli for the language PML_2 [Lep16]. SizeChange Tool reuses their work.

All this information is summarized in Figure 7.1, a visual description of the different steps performed by SizeChange Tool.

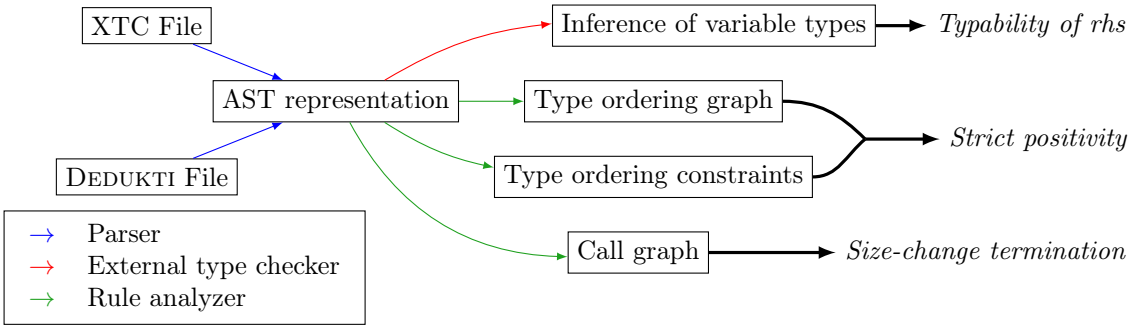


Figure 7.1: SizeChangeTool Workflow

One must note the red arrow in this workflow presentation, which highlights usage of the external type checker. This arises naturally, since the termination criterion of Theorem 6.9.1 is type-directed. To get this interaction between the type checker and the termination checker, one could have chosen to simply implement SizeChange Tool as a plugin in Dedukti, and to make a monolithic multipurpose project. However, since Dedukti is a *Logical Framework* designed for interoperability, having external tools, which do not rely on Dedukti's implementation and can easily be ported to other tools, was a more consistent choice.

Of course, even if it is a blue arrow, the parser of Dedukti files was not duplicated. The files are first parsed by Dedukti and then the abstract syntax tree is translated to the one internal to SizeChange Tool.

7.2 Examples

The reader of Chapter 6 might have been a bit frustrated by the fact that a termination criterion is presented, but no concrete example of what kind of problem it is able to deal with is shown. This section will try to remedy this issue, by presenting a few examples.

7.2.1 Strength of Size-Change Termination

It is not our purpose to develop an efficient first-order termination checker, since several of them already exist. However, we did not chose the size-change termination technique haphazardly. It is able to prove terminating some interesting examples other techniques cannot deal with.

For instance, it has no problem to prove the termination of non primitive recursive function and can deal with the combination of argument permutations and duplications, for which usual path ordering techniques, like Recursive Path Ordering, or Computability Path Ordering [BJR08] are unable to prove termination.

To illustrate this, one can study the definition of the Ackermann function, well-known to be not primitive recursive, and the function f, introduced by Thiemann and Giesl in [TG05] as an example of function which cannot be handled by RPO or CPO.

```
constant ℕ : ⋆.
constant 0 : ℕ.
constant s : ℕ ⇒ ℕ.

symbol Ackermann : ℕ ⇒ ℕ ⇒ ℕ.
[n]    Ackermann 0     n     ↪ s n
```

```
[m]    Ackermann (s m) 0     ↪ Ackermann m (s 0)
[m, n] Ackermann (s m) (s n) ↪ Ackermann m (Ackermann (s m) n).

symbol f : ℕ ⇒ ℕ ⇒ ℕ.
[]     f 0     0     ↪ 0.
[x,y]  f (s x) 0     ↪ f x (s x).
[x,y]  f x     (s y) ↪ f y x.
```

Since it is a simply-typed first-order example, all the variables are accessible, it suffices to consider the "flat" ordering where every type constructor is smaller or equal to all the others. All the right-hand sides are typable in this example, and the rewriting system is orthogonal[2] so locally confluent. The only thing remaining to check is size-change termination.

For the Ackermann function, the second rule and the third rule contain recursive calls. For the second rule, there is only one, and the associated matrix is $A = \begin{pmatrix} -1 & \infty \\ \infty & \infty \end{pmatrix}$. For the third one, there are two calls, one external and one internal. The matrices respectively associated to each of them are A and $B = \begin{pmatrix} 0 & \infty \\ \infty & -1 \end{pmatrix}$.

One can then compute $A^2 = AB = BA = A$ and $B^2 = B$. Those equalities ensure that the only two matrices to be considered are A and B and both of them have a -1 on the diagonal, hence, one can use the Theorem 6.9.1 to conclude that this definition of the Ackermann function is terminating.

Regarding the function f, there are two recursive calls, one in the second and one in the third line. The associated matrices are $C = \begin{pmatrix} -1 & 0 \\ \infty & \infty \end{pmatrix}$ and $D = \begin{pmatrix} \infty & 0 \\ -1 & \infty \end{pmatrix}$. When computing the smallest set of matrix containing C and D and closed by multiplication, one obtains $\left\{ \begin{pmatrix} -1 & 0 \\ \infty & \infty \end{pmatrix}, \begin{pmatrix} \infty & 0 \\ -1 & \infty \end{pmatrix}, \begin{pmatrix} -1 & -1 \\ \infty & \infty \end{pmatrix}, \begin{pmatrix} \infty & \infty \\ -1 & -1 \end{pmatrix}, \begin{pmatrix} -1 & \infty \\ \infty & -1 \end{pmatrix}, \begin{pmatrix} \infty & -1 \\ -1 & \infty \end{pmatrix} \right\}$.

Among those, the only idempotent ones are $\begin{pmatrix} -1 & -1 \\ \infty & \infty \end{pmatrix}$, $\begin{pmatrix} \infty & \infty \\ -1 & -1 \end{pmatrix}$ and $\begin{pmatrix} -1 & \infty \\ \infty & -1 \end{pmatrix}$, which all have a -1 on the diagonal, ensuring us that this function f is proved terminating using Theorem 6.9.1.

7.2.2 With Dependent Types

But our tool would not offer novelty if it was restricted to the examples presented in the previous section, since size-change termination is already used by state-of-the-art first-order termination prover. The true strength of SizeChange Tool is its ability to deal with dependent types.

Hence, let us present an example using dependent types. For this, we will choose the definition of a summation with variable arity.

To define such a function, one has to start by defining the type of this function, which will be `(n:Nat) ⇒ D n`, meaning that the first argument will be a natural number, telling us how many elements will be summed.

The definition of this type `D`, which was already introduced in Example 6.6.10, is

```
symbol D : ℕ ⇒ ⋆.
[]  D 0     ↪ ℕ.
[n] D (s n) ↪ ℕ ⇒ D n.
```

Since `D` is defined by rewriting rules, it is not a type constructor.

[2] A rewriting system is orthogonal if it is left-linear and there no overlap between two left-hand sides.

Once this type is defined, it is direct to say that the value of the empty sum is zero, that the summation of only one number outputs this number and that if one has several numbers to sum, it suffices to accumulate the partial result in the first argument.

```
symbol sum : (n: ℕ) ⇒ D n.
[]  sum 0         ↪ 0.
[]  sum (s 0)     ↪ λx, x.
[n] sum (s (s n)) ↪ λx, λy, sum (s n) (x + y).
```

Once again, this system is orthogonal, so locally confluent, and all the right-hand sides are well-typed. The two things to check is that n is accessible in the last rule, and that this rule verifies size-change termination.

Since n occurs as the first argument of s, one has to verifies that $\mathbb{N}$ is a type constructor and that $\mathbb{N}$ is smaller or equal to $\mathbb{N}$. Since those checks are direct, it only remains to check that the rule `[n] sum (s (s n)) ↪ λx, λy, sum (s n) (x + y)` is size-change terminating. This is not long, since it just means that one has to note that `s n` is a subterm of `s (s n)`.

As far as the author know, SizeChange Tool is the only tool able to automatically prove this example terminating.

7.3 Implementation Is Ahead of Theory

Until this point, we restricted ourselves to study rewriting rules with patterns containing no λ-abstractions (see Definition 4.1.2). However, both Dedukti and the Termination Competition allow a richer class of patterns. For instance, in Dedukti, the left-hand side of a rewriting rule must be a *Miller pattern headed by a symbol of the signature* [Mil91].

7.3.1 Higher-Order Matching

To define this new class of patterns, one must firstly define new notions of variables and terms, called *meta-variables* and *meta-terms*.

The definition of rewriting in this section is a form of *Algebraic Functional System with Meta-variable* (AFSM) introduced by Kop in [Kop12] and used to define static dependency pairs in [FK19]. AFSM's originate themselves from *Combinatory Reduction Systems* introduced by Klop [Klo80, KvOvR93]. However, in our setting all the functions are curried, meaning that we do not require them to be applied, all the symbols of the signature are considered of arity 0, and only the meta-variables can have a non-zero arity in our setting.

Definition 7.3.1 (Meta-variable)**.** *Let $\mathcal{M}$ be an infinite set of names disjoint of $\mathcal{V}$, $\mathcal{S}$ and $\mathcal{F}$.*

Definition 7.3.2 (Meta-term)**.** *A* meta-term *is defined by the grammar:*

$$M ::= M\,M \;\mid\; \lambda\,(x : M)\,.M \;\mid\; (x : M) \Rightarrow M \;\mid\; s \in \mathcal{S} \;\mid\; f \in \mathcal{F} \;\mid\; x \in \mathcal{V} \;\mid\; X[M^*] \text{ with } X \in \mathcal{M}$$

Contrary to what was done in Definition 3.1.2, we do not present a non-ambiguous grammar here. In case of ambiguity, the conventions for terms are reused: binders bind as far as possible, applications associate to the left and arrows to the right. Despite the fact that parentheses are only around the annotation of a variable in this grammar, it is required to add supplementary parentheses to make the intended priority of operations explicit.

We introduce a function MV to list the meta variables of a meta-term:

Definition 7.3.3 (Free meta-variables)**.** *The function* MV *is defined by:*

$$\mathrm{MV}(t\,u) = \mathrm{MV}(t) \cup \mathrm{MV}(u) \qquad \mathrm{MV}(\lambda\,(x : A)\,.t) = \mathrm{MV}(A) \cup \mathrm{MV}(t)$$
$$\mathrm{MV}((x : A) \Rightarrow B) = \mathrm{MV}(A) \cup \mathrm{MV}(B) \qquad \mathrm{MV}(a) = \emptyset \text{ if } a \in \mathcal{S} \cup \mathcal{F} \cup \mathcal{V}$$
$$\mathrm{MV}(X[\vec{t}]) = \{X\} \cup \bigcup_i \mathrm{MV}(t_i) \text{ if } X \in \mathcal{M}$$

Contrary to what was done for FV, since there are no binders for meta-variable, the meta-variables of a term is the union of the meta-variables of its subterms.

Definition 7.3.4 (Arity of a meta-variable)**.** *In the construction of a meta-term, each meta-variable comes with a finite list of meta-terms. The length of this list is called the* arity *of the meta-variable.*

Definition 7.3.5 (Closed meta-terms)**.** *A meta-term is said* closed *if it does not contain free variables (even if it contains meta-variables).*

Definition 7.3.6 (Miller Pattern)**.** *A* Miller pattern *is described by the syntactic category P below.*

$$\begin{aligned} P &::= f\,P^* \text{ with } f \in \mathcal{F} \;\mid\; O \\ N &::= f \in \mathcal{F} \;\mid\; x \in \mathcal{V} \\ O &::= \lambda x.M \;\mid\; X[x_1, \ldots, x_n] \text{ with } X \in \mathcal{M} \text{ and } x_i\text{'s are distinct variables} \\ M &::= N\,M^* \;\mid\; O \end{aligned}$$

In the previous definition, N is the syntactic category of meta-terms without sort or product, but also without applications, meta-variables and λ-abstractions. Those last two constructions are in their own syntactic class, namely the class O. The reason for isolating them is the class M, which specifies the syntactic restrictions on the application, namely that one cannot apply an abstraction (to avoid creating β-redices) or a meta-variable.

Let us first finish to define what a rewriting rule is in this new framework:

Definition 7.3.7 (Rewriting rule)**.** *A* rewriting rule *is an ordered pair $f\,\vec{l} \hookrightarrow r$ where*

- *the left-hand side $f\,\vec{l}$ is a closed pattern headed by a symbol of the signature,*
- *the right-hand side r is a meta-term;*

such that all meta-variables occurring in the right-hand side also occurs in the left-hand side, and such that all the occurrences, in both sides of the rule, of the same meta-variable always have the same arity.

This new definition of patterns (Definition 7.3.6) allows to "apply" a meta-variable to distinct bound variables. This means that one can have the rule

$$\texttt{[F]}\ \partial\ (\lambda\ \texttt{x, sin F[x]}) \hookrightarrow (\partial\ (\lambda(\texttt{x} : \mathbb{R}), \texttt{F[x]})) \times (\lambda(\texttt{x} : \mathbb{R}), \texttt{cos F[x]})$$

where ∂, $\mathbb{R}$, sin, cos and $\times$ are considered in the signature and the infix notation is used for $\times$. With this example, one can wonder if $\partial\,(\lambda x.\,(\sin\,x))$, $\partial\,(\lambda x.\,(\sin\,(2 \times x)))$ or $\partial\,(\lambda x.\,(\sin\,(x \times x)))$ are rewritten using this rule, and if they are, by what is replaced F in the right-hand side.

If one prefer to use a more technical vocabulary, the question is what is the definition of a unifier in this setting of meta-variables with strictly positive arity.

To define a substitution σ in this setting, one has to find a way to ensure that $X[t_1, \ldots, t_n]\sigma$ is a term if the t_i's are. To achieve this property, one defines the notion of pseudo-term, which is what should stand for $\sigma(X)$.

Definition 7.3.8 (Pseudo-term)**.** *A* pseudo-term *is of the form* $\underline{\lambda x_1, \ldots, x_n}.t$*, where* t *is a usual term.*

Now, let us define the meta-substitutions:

Definition 7.3.9 (Meta-substitution)**.** *A* meta-substitution σ *is a function from meta-variables to pseudo-terms, such that if* X *is of arity* n*, then* $\sigma(X)$ *is of the form* $\underline{\lambda x_1, \ldots, x_n}.t$*, where* t *is a usual term.*

Now that several structures, which use meta-variables, have been defined, we will extend the application of meta-substitutions to all those structures.

Definition 7.3.10 (Application of a meta-substitution to a meta-term)**.** *Let* σ *be a meta-substitution. Then:*

$$
\begin{aligned}
(t\,u)\,\sigma &= t\sigma\ (u\sigma) & (\lambda\,(x : A)\,.t)\,\sigma &= \lambda\,(x : A\sigma)\,.t\sigma \\
((x : A) \Rightarrow t)\,\sigma &= (x : A\sigma) \Rightarrow t\sigma & x\sigma &= x \text{ if } x \in \mathcal{V} \cup \mathcal{F} \cup \mathcal{S} \\
X[t_1, \ldots, t_n]\sigma &= u\left[{}^{t_1\sigma}/_{x_1}, \ldots, {}^{t_n\sigma}/_{x_n}\right] \text{ if } \sigma(X) = \underline{\lambda x_1, \ldots, x_n}.u
\end{aligned}
$$

Contrary to what is done in the case of usual substitution Definition 3.3.2, where $\sigma\,[x \mapsto x]$ is required to be used under binders, in the case of meta-substitutions, there cannot be any conflict between the variable names which are bound and the meta-variables which are substituted.

Rigorously, a pattern is not a meta-term, since the λ-abstractions are not annotated in patterns. However, the definition of a meta-substitution applied to a pattern is similar enough to be inferred by the reader.

The attentive reader might have noticed that applying such a meta-substitution to a pattern will create heterogeneous terms, where some λ-abstractions are annotated, and some others are not.

Definition 7.3.11 (Instance of a pattern)**.** *A term* t *is an* instance of the pattern p*, if there is a meta-substitution* σ *such that* $p\sigma \equiv t$*.*

Here $\equiv$ *is* α*-equivalence (Definition 3.2.5) enriched with the irrelevance of the type annotations which are "missing" in patterns:* $\lambda x.t \equiv \lambda\,(y : A)\,.u$ *if* x *does not occur in* u *and* $t \equiv u\left[{}^{x}/_{y}\right]$*.*

Just like in Definition 4.1.4, we define the rewriting relation to be the closure by context and substitution of the rewriting rules, using instantiation of patterns for the left-hand side rather than simple substitution.

Definition 7.3.12 (Rewriting Relation)**.** t reduces at the head *to* u *if there is a meta-substitution* σ *and a rule* $f\,\vec{l} \hookrightarrow r$ *such that* $\left(f\,\vec{l}\right)\sigma \equiv t$ *and* $r\sigma = u$*.*

The rewriting relation *is then the contextual closure of this reduction at the head.*

7.3.2 Adapting Accessibility

Now that we have a new definition of rewriting, it is possible to present exactly the criterion implemented in SizeChange Tool.

This requires to redefine two notions:

- which meta-variables are accessible in a Miller pattern and how to adapt AVO rules,
- how to extend $\triangleright_{acc}$ to compare meta-terms.

The purpose of accessibility is to identify some positions for which taking the subterm at this position preserves interpretability. Furthermore, since interpretations are reducibility candidates (Section 6.3), they are stable by reduction and contain all the normal neutral terms, so every variable. If $\lambda\,(x : A)\,.t \in [\![(x : A) \Rightarrow B]\!]$, this means that for every $u \in [\![A]\!]$, $(\lambda\,(x : A)\,.t)\ u \in [\![B\,[u/_x]]\!]$. Especially, since variables are in all the interpretations, $(\lambda\,(x : A)\,.t)\ x \in [\![B]\!]$, and by reduction, $t \in [\![B]\!]$.

Hence, it seems quite safe to modify accessible positions (Definition 6.7.1) and type inference (Definition 6.7.2) to state that if a λ-abstraction is accessible, so is its body. We denote the position of the body of an abstraction by the symbol $\bullet$, so the positions are not simply sequences of natural numbers anymore.

Since we are not only going through constructors, but also λ-abstractions, the function is not denoted AccPosCstr anymore, but simply AccPos.

Definition 7.3.13 (Accessible Positions Reworked)**.** *We define the function:*

$$\textit{If } \begin{cases} t = c\,u_1 \dots u_k \\ c \in \mathcal{C}_o \\ \Theta(c) = (y_1 : U_1) \Rightarrow \dots \Rightarrow (y_r : U_r) \Rightarrow C\,\vec{v} \\ \exists W_{k+1} \dots W_r, \vec{v}', T \rightsquigarrow^* (y_{k+1} : W_{k+1}) \Rightarrow \dots \Rightarrow (y_r : W_r) \Rightarrow C\,\vec{v}' \end{cases}$$

$$\textit{then:} \qquad \mathrm{AccPos} : (t, T) \mapsto \{\varepsilon\} \cup \bigcup_{i \in \mathrm{Acc}(c)} \left\{ i.p \;\middle|\; p \in \mathrm{AccPosCstr}\left(u_i, U_i \left[\overrightarrow{u/_y}\right]\right) \right\}$$

$$\textit{else: if} \quad \begin{cases} t = \lambda x.u \\ \exists A, U, T = (x : A) \Rightarrow U \end{cases}$$

$$\textit{then:} \qquad \mathrm{AccPos} : (t, T) \mapsto \{\varepsilon\} \cup \{\bullet.p \mid p \in \mathrm{AccPos}\,(u, U)\}$$

$$\textit{else:} \qquad \mathrm{AccPos} : (t, T) \mapsto \{\varepsilon\}$$

and:

$$\mathrm{AccPosHd}' : (f\,\vec{t}) \mapsto \bigcup_i \{i.x \mid x \in \mathrm{AccPos}(t_i, T_i)\} \;\; \textit{with } \Theta(f) = \overrightarrow{(x : T)} \Rightarrow U$$

For the inference of types, it is not always possible to infer the type of the body of an unannotated λ-abstraction. Hence, the new function to infer type, called InferType$'$, not only takes a term t and a position p, but also a supplementary argument T which is the type inferred for t.

Definition 7.3.14 (Type Inference Reworked)**.** *We now define the function:*

$$\mathrm{InferType} : (p, t, A) \mapsto \begin{cases} A & \textit{if } p = \varepsilon \\ \mathrm{InferType}(p', u_i, T_i \left[\overrightarrow{u/_y}\right]) & \textit{if } p = i.p', t = f\,\vec{u} \textit{ and } \Theta(f) = \overrightarrow{(y : T)} \Rightarrow U \\ \mathrm{InferType}(p', u, U) & \textit{if } p = \bullet.p', t = \lambda x.u \textit{ and } T = (x : V) \Rightarrow U \end{cases}$$

InferType$'$ is used to denote InferType with a dummy third argument. Indeed the third argument of InferType is a type only used when the position given as first argument is ε, so whenever one has the guarantee that the position studied is not the head of the term, this third argument can be removed.

We then can define *Accessible Meta-Variable Only* rules, analogously to what we did in Definition 6.7.3.

Definition 7.3.15 (AMVO rules)**.** *A rule $f\, l_1 \dots l_n \hookrightarrow r$ is* Accessible Meta-Variable Only (AMVO) *if*

- *there is a function $\phi : \mathrm{MV}(r) \to \mathrm{AccPosHd}'(f\,\vec{l})$ selecting an occurrence of each meta-variable, meaning that for all $X \in \mathrm{MV}(r)$, there are $\vec{u}$ such that $(f\,\vec{l})|_{\phi(X)} = X[\vec{u}]$.*
- *$\Delta_r \vdash r : T_r$, where*
  - *$\Delta_r = \left\{ X : \mathrm{InferType}'(\phi(X), f\,\vec{l}) \;\middle|\; X \in \mathrm{MV}(r) \right\}$ ordered by the alphabetical order on $\phi(x)$.*
  - *$T_r = U\left[\overrightarrow{l/_x}\right]$, where $\Theta(f) = (x_1 : T_1) \Rightarrow \dots \Rightarrow (x_n : T_n) \Rightarrow U$.*

This definition of *AMVO* rules, is very similar to what Fuhs and Kop called *Accessible Function Passing* (AFP) rules [FK19], however, since they are in a simply-typed context, they do not have to restrict themselves to the usage of symbols of a "smaller layer" to type-check the right-hand sides of rules, hence their definition of *AFP* rules is mostly the existence of the function ϕ in our definition of *AMVO* rules.

7.3.3 Adapting the Structural Order $\triangleright_{acc}$

We now have to adapt $\triangleright_{acc}$ (Definition 6.5.1), to allow ourself to select subterms which are under λ-abstractions. Furthermore, just like we applied subterms to invented terms in the interpretation of the expected type, we want to replace the bound variable of a meta-variable by meta-terms which are in the interpretation of their types.

This means that in a rule $f\,\vec{l} \hookrightarrow r$, if $\lambda y.t$ is accessible in $f\,\vec{l}$, then it possible to infer the type of y (it is the domain of the type inferred for $\lambda y.t$, which is of the shape $(y : A) \Rightarrow B$). Hence, if $X[\vec{y}]$ is accessible, it is possible to infer the type expected for each argument of X. For this, we define a function InferMetasHd, which takes as input a Miller pattern headed by a function of the signature, and InferMetas, which takes not only a Miller pattern, but also the inferred type and a set with the types of bound variables. Both functions outputs a set of ordered pairs, with a meta-variable name, and the list of the types inferred for its arguments. The length of this list is equal to the arity of the meta-variable. Sets of couples are seen as functions, in the set-theoretic tradition, whenever it alleviates notations.

Definition 7.3.16 (Type inference for arguments of a meta-variable)**.** *For $f \in \mathcal{F}$, with $\Theta(f) = \overrightarrow{(x : T)} \Rightarrow U$, we define:*

$$\mathrm{InferMetasHd} : (f\,\vec{l}) \mapsto \bigcup_i \mathrm{InferMetas}(l_i, T_i\left[\overrightarrow{l/_x}\right], \emptyset)$$

and

$$\mathrm{InferMetas} : (t, T, S) \mapsto \begin{cases} \bigcup_{i \in \mathrm{Acc}(c)} \mathrm{InferMetas}(l_i, T_i\left[\overrightarrow{l/_x}\right], S) & \text{if } t = c\,\vec{l}, c \in \mathcal{C}_o \text{ and } \Theta(c) = \overrightarrow{(x : T)} \Rightarrow U \\ \mathrm{InferMetas}(u, B, S \cup (x, A)) & \text{if } t = \lambda x.u \text{ and } T = (x : A) \Rightarrow B \\ \{(X, map(S, [\vec{y}])\} & \text{if } t = X[\vec{y}] \end{cases}$$

where map is the function which takes as input a function f and a list l and outputs the list of the result of the application of f to all the elements of l, $\mathrm{map}(f, [a_1, \ldots, a_n]) = [f(a_1), \ldots, f(a_n)]$. *It can also be found in the Example 5.1.7.*

We now can instantiate the meta-variables occurring in the bodies of λ-abstractions, with terms in the interpretation of the expected types:

Definition 7.3.17 (Instance of a meta-term)**.** *Given a rule $f\,\vec{l} \hookrightarrow r$, we define the function* Instances*, which takes a Miller pattern and outputs a set of Miller pattern by:*

$$\begin{aligned}
\mathrm{Instances}(\lambda x.t) &= \{\lambda x.u \mid u \in \mathrm{Instances}(t)\} \\
\mathrm{Instances}(f\,\vec{t}) &= \{f\,\vec{u} \mid \textit{for all } i, u_i \in \mathrm{Instances}(t_i)\} \\
\mathrm{Instances}(x\,\vec{t}) &= \{x\,\vec{u} \mid \textit{for all } i, u_i \in \mathrm{Instances}(t_i)\} \\
\mathrm{Instances}(X[\vec{y}]) &= \left\{ X[\vec{u}] \;\middle|\; \textit{for all } i, u_i \in \left[\!\left[\pi_i(\mathrm{InferMetasHd}(f\,\vec{l})(X)) \right]\!\right] \right\}
\end{aligned}$$

We now can define $\blacktriangleright_{acc}$ similarly to what was done for $\triangleright_{acc}$, but extending the possibility to invent arguments to the case of meta-variables.

Definition 7.3.18. *Given a rule $f\,\vec{l} \hookrightarrow r$, we have that*

- $\left(c\,t_1 \ldots t_{\mathrm{ar}(c)}\right) \blacktriangleright_{acc} (t_i\,\vec{u})$, *if* $c \in \mathcal{C}_o$, $\Theta(c) = \overrightarrow{(x : T)} \Rightarrow U$, $i \in \mathrm{Acc}(c)$, $T_i = \overrightarrow{(y : V)} \Rightarrow W$ *and* $\left[\overrightarrow{u/_y}\right] \vDash \overrightarrow{(y : V)}$,
- $(\lambda x.t) \blacktriangleright_{acc} u$ *if* $u \in \mathrm{Instances}(t)$ *for the rule* $f\,\vec{l} \hookrightarrow r$.

Just like we restricted $\triangleright_{acc}$ in Definition 6.5.6, for decidability, we restrict ourselves to use symbols of the signature which are not defined by rewriting rules and variables when inventing a term u in an interpretation.

Now, we can state the analogous of Theorem 6.9.1 in this extended setting:

Conjecture 7.3.19 (Implemented Criterion)**.** $\rightsquigarrow = (\rightsquigarrow_\beta \cup \rightsquigarrow_{\mathcal{R}})$ *terminates on terms typable in* $\lambda\Pi$ *modulo rewriting if*

- $\rightsquigarrow$ *is locally confluent,*
- $\mathcal{F}_\mathcal{T} \cup \mathcal{F}_o$ *is finite,*
- $\mathfrak{R}$ *is AMVO,*
- $\mathfrak{R}$ *is size-change terminating for* $\blacktriangleright_{acc}$.

7.4 Comparison with other tools

As far as the author knows, there are no other termination checker combining dependent types and non-orthogonal rewriting rules. However, dropping one of these features and restricting ourselves to simply-typed higher-order rewriting systems or to dependently-typed orthogonal systems permits comparison with existing tools.

For simply-typed systems, the termination competition [TC] proposes the category "higher-order rewriting union beta". In 2019, there were only two tools competing in this category: SizeChange Tool and Wanda [Kop]. Wanda uses multiple techniques to prove termination: dependency pairs, polynomial interpretations, HORPO... [Kop12]. Unsurprisingly, the

sole criterion used in SizeChange Tool cannot prove as many examples as this wide range of techniques.

However, on the bench of the competition, SizeChange Tool is 11 times faster than Wanda. The speed of SizeChange Tool permits it to show in less than 0.1 second termination of examples for which Wanda requires several seconds, sometimes even more than a minute. The very low time consumption of the presented criterion suggests that Wanda would improve significantly its efficiency by adding our technique to its toolbase.

If we restrict ourselves to orthogonal systems, it is then possible to compare our technique to the ones implemented in Coq and Agda. Coq essentially implements a higher-order version of primitive recursion [Gim94], with various extension of the guard condition, like [Sac11, Bou12], whereas Agda uses subterm criterion (a criterion very similar to size-change termination) [Abe98]. Hence, Coq cannot handle function definitions with permuted arguments in function calls, which is not a problem for Agda and SizeChange Tool. Agda recently added the possibility of declaring rewriting rules but this feature is highly experimental and no check is performed on the rules. In particular, Agda termination checker does not handle rewriting rules.

7.5 Limitations and Improvements of SizeChange Tool

Even if the main motivation of the development of SizeChange Tool was not to compete with Wanda on simply-typed rewriting systems, it is instructive to analyze the files on which Wanda obtains results, whereas SizeChange Tool does not.

Hence, we took the benchmark of the 2019 termination competition and analyzed all the files which are rewriting systems with Miller patterns for which SizeChange Tool outputs "MAYBE" and Wanda "YES", meaning that Wanda was able to prove the system terminating and not SizeChange Tool.

There are 78 such files. Among them, we distinguished 3 categories:

- first-order difficulty: Those are files for which the only difficulties SizeChange Tool is facing are inability to prove termination of the first-order subset of the rules;
- Accessibility issues: Those are files for which the accessibility criterion provided in Section 6.1 is too restrictive;
- Subterms limitations: Those are the remaining files, for which the necessity to have a decrease in a subterm was a too strong requirement.

7.5.1 Having a First-Order Backend

Regarding the files of the "Higher-order Rewriting Union Beta" section of the termination competition, C. Kop observed: "about half the benchmarks now do little more than test the strength of the first-order back-end that some higher-order tools use." [Kop19]. The study of the results of the 2019 Termination Competition [TC] confirms the affirmation of Kop, since among the 87 problems proved terminating by Wanda and not by SizeChange Tool, 9 do not fit into our definition of patterns (Definition 7.3.6), and 38 of the 78 remaining tests are refused by SizeChange Tool only because of the impossibility to prove the termination of a first-order set of rules using dependency pairs with the size-change termination processor.

Hence, following the approach adopted by Wanda, one could also just study truly higher-order rules, use a state-of-the-art first-order prover for the remaining rules and then rely on a modularity theorem, like the ones of [JO97, FK11], to conclude. Such a modular checking would

definitely improve the performance of SizeChange Tool. However, it would require to extend the modularity criteria to our setting with dependent types.

One could even imagine to go further in the direction of modularization, and to delegate the simply-typed rules to a prover like Wanda, and only use SizeChange Tool to study the dependently-typed rules. But a modularity result for this situation does not exist and is still to craft.

7.5.2 About Logic Encodings

As already mentioned several times above, the main aim of Dedukti is to allow the user to define logics. Hence, one could legitimately expects from a termination checker for Dedukti to be able to prove some logic encodings terminating.

Currently, Theorem 6.9.1 allows to deal with the encoding of simple types, like `Set`, `arrow` and `El` in Example 5.1.7, or the encoding in Cousineau and Dowek's style [CD07] of the PTS $\lambda^{\rightarrow}$ (Example 3.4.2). It must be noted that $\lambda^{\rightarrow}$ is finitely sorted, hence there is no reason to use the enrichment of the encoding proposed by Assaf and presented in Section 5.3.

Even worse than that, this enrichment would not represent faithfully the finitely sorted PTS $\lambda^{\rightarrow}$. Indeed, since there is a unique function `code` in this extension, which constructs the sort above its input, the term `code (code (code (code (code T))))` is well-typed, hence there are always infinitely many sorts in this version of the encoding, even if finitely many exists in the PTS we are translating.

To encode the $\lambda^{\rightarrow}$-calculus, one simply needs 6 symbols and 2 rules:

```
constant K : ⋆.
constant T : ⋆.

symbol eps : T ⇒ ⋆.
symbol eta : K ⇒ ⋆.

constant t : K.
[] eta t ↪ T.

constant Pitt : T ⇒ T ⇒ T.

[a, b] eps (Pitt a b) ↪ eps a ⇒ eps b.
```

The two sorts $\star$ and $\Box$ are often called Type and Kind, and are in this encoding represented by the letters `T` and `K`.

Since there are two spaces of codes, one for each sort of the $\lambda^{\rightarrow}$, there are two decoder functions, `eta` for kinds and `eps` for types. Furthermore, to represent the inhabitation Type : Kind (i.e. the axiom $\star : \Box$), a code for Type in the space `K` is declared. It is the symbol `t`.

Finally, the product of $\lambda^{\rightarrow}$ is declared, together with the rewriting rule to decode it.

This system is locally confluent since it is orthogonal. It uses a finite number of symbols.

The only dependency pairs are:

```
eps (Pitt a b) > eps a
eps (Pitt a b) > eps b
```

And we directly see that, in both dependency pairs, the first argument of `eps` on the right is a strict subterm of the one on the left.

Thus, to conclude that this system is terminating, thanks to Theorem 6.9.1, the only question is whether `a` and `b` are accessible variables. Since `T` is a type constructor smaller or equal to `T`, it is easily seen that the accessibility condition is fulfilled.

However, our criterion would require a few enrichments to handle richer logics.

The main issue the user of Theorem 6.9.1 faces, when studying higher-order rewriting rules, is that the accessibility criterion is way too restrictive. Indeed, in the 2019 Termination Competition, 23 problems proved terminating by WANDA (out of 78) are not handled by SIZECHANGE TOOL, because of positivity issues.

This issue also occurs when one adds dependent types in the calculus one is encoding and trying to prove the termination of. Here is the encoding of the calculus $\lambda\Pi$, in the $\lambda\Pi$-calculus modulo rewriting:

```
constant K : ⋆.
constant T : ⋆.

symbol eps : T ⇒ ⋆.
symbol eta : K ⇒ ⋆.

constant t : K.
[] eta t ↪ T.

constant Pitt : (a : T) ⇒ (eps a ⇒ T) ⇒ T.
constant Pitk : (a : T) ⇒ (eps a ⇒ K) ⇒ K.

[a, b] eps (Pitt a b) ↪ (x : eps a) ⇒ eps (b x).
[a, b] eta (Pitk a b) ↪ (x : eps a) ⇒ eta (b x).
```

The dependency pairs are:

```
eps (Pitt a b) > eps a
eps (Pitt a b) > eps (b x)
eta (Pitk a b) > eps a
eta (Pitk a b) > eta (b x)
```

And it would not be a trouble to use the Size-Change Termination criterion to prove it terminating, if the variable `b` was accessible. But, since `eta` is a defined symbol in this example, `eta a ⇒ T` is not a frozen type, so the second argument of `Pitt` is not accessible.

To handle this system, the definable types would have to be included in the order on type constructors introduced in Condition 6.1.2. But, types headed by the same defined symbol can have different interpretation, since they might reduce to types of very different shape, some of them might produce arrows, or different type constructors. Hence, doing this would require to interleave the definition of the interpretation of type values Definition 6.2.1, with the interpretation of the other types of Section 6.2.2.

Just like the proof of the extension of Theorem 6.9.1 to higher-order patterns (Conjecture 7.3.19), such a definition of interpretations which includes definable types in the ordering of type constructors is left as future work.

Chapter 8

η-conversion

8.1 Extending Conversion

Many proof assistants implement, among other conversion rules, the η rule, which states that if f is a function, $f \equiv_\eta \lambda x.f\ x$.

At first sight, this conversion might look quite harmless, and one can hope to just add the corresponding rewriting rule. However, this relation is an important issue for translation of logical systems in $\lambda\Pi$-modulo rewriting. Indeed, the contraction rule cannot be stated. The main issue is that the rule defining η-reduction is not headed by a symbol of the signature. Since such a left-hand side is not "algebraic" [BJO99], the study of the meta-theory of the system is deeply modified. For termination, the interpretation we defined in Section 6.2 relies on the fact that all the redices are headed by an application, and allowing some of them to be headed by an abstraction would require to completely rethink the definition of this interpretation. Furthermore, $\beta\eta$ is not confluent on untyped λ-terms and one has to restrict herself to some typed system to recover the Church-Rosser property in this case, as shown by Geuvers [Geu92].

Even in a framework where the left-hand sides of rewriting rules can be headed by an abstraction, it is often impossible to write the rule `[f]λx,f x ↪ f` for the η-contraction, since it would require to match on the fact that $f\ x$ is an application, which would be "meta-matching". In Klop's CRS [Klo80] and in Saillard's definition of $\lambda\Pi$ modulo [Sai15], to avoid this "meta-matching", the matching of this kind of patterns is done modulo β, meaning that $\lambda x.f\ x$, matches any abstraction, not only the ones where the body is the application of a term in which the bound variable does not occur to the bound variable. In our definition of higher-order pattern (Definition 7.3.6), one would have to write `[F]λx,F[x]`, and it also matches all the abstractions, and not simply those of the shape we are interested in. Furthermore, one would be in trouble when trying to write the right-hand side, since `F` is of arity one, so it must be given an argument to be valid. Lastly, to preserve typing, the expansion rule has to match on the type of a variable, and is not syntax-directed anymore.

Another natural solution could be to define $\lambda\Pi$-modulo rewriting as a logical framework with η hard coded in the conversion (just like β). But this is a path *logical frameworks* want to avoid. Indeed, if η is hard coded, it is impossible to have a shallow encoding of λ-calculi without η-conversion.

One could expect that η-expanding every term during the translation phase, could allow us to completely ignore η-conversion in the $\lambda\Pi$-calculus modulo rewriting. But this is not possible with dependent types, since it might happen than an η-long term has a non-η-long type. A situation that often breaks *type preservation of the translation.*

Example 8.1.1. *To illustrate this, we start by defining a type* `D n`*, already introduced in Example 6.6.10 and Section 7.2.2, whose number of arrows depends on a natural number, with a constructor* `d` *for this type.*

```
symbol D : ℕ ⇒ ⋆.
[]  D 0     ↪ ℕ.
[n] D (s n) ↪ ℕ ⇒ D n.

constant d : (n : ℕ) ⇒ D n.
```

We then define a new type `E n` *depending on* `D` *and a constructor* `e` *for this new type.*

```
symbol E : (n : ℕ) ⇒ D n ⇒ ⋆.     symbol e : (n : ℕ) ⇒ E n (d n).
```

Now, the term `e 1` *is η-long and has type* `E 1 (d 1)`*, but, in absence of η-conversion, it does not have the type* `E 1 (λ x, d 1 x)` *which is the η-long form of the type.*

8.2 The Time-Bomb Symbol

To overcome this issue, we propose to postpone η-expansion until the type is fully instantiated. For this, we introduce a symbol ηE in the translation, which purpose is to tag with their types the subterms which may become η-expansible. This symbol acts as a "time-bomb". It stays in the term as long as the type annotation is not explicit enough to know whether η-expansion will be required, and when this information becomes available, it triggers the eta-expansion of the term and the erasing of itself. To do so, some rewriting rules pattern match on this type annotation to decide when and how the expansion can be performed.

Our goal is to translate a terminating full PTS enriched with η-conversion, hence, we consider the type system of Definition 3.4.9, with the conversion rule enriched to include both β and η-conversion.

$$(\text{conv'}) \quad \frac{\Gamma \vdash_\eta t : A \quad \Gamma \vdash_\eta B : s}{\Gamma \vdash_\eta t : B} \; A \equiv_{\beta\eta} B$$

Since we are doing translation between type systems, it is required to distinguish the terms and judgements of $\lambda\Pi$ modulo rewriting from the ones of the source PTS with η-conversion. For this, the turnstile symbol for the latter are indexed by η. Furthermore, the notation of the binders is slightly modified: the type annotations of λ-abstractions are superscript and the products are denoted by a simple arrow.

So $\Gamma \vdash \lambda\,(x : A)\,.t : (x : A) \Rightarrow B$ in the source PTS is the analogous of $\Gamma \vdash_\eta \lambda x^A.t : (x : A) \to B$ in its extension with η.

In the following, we reuse and extend the encoding of *Pure Type Systems* introduced by Cousineau and Dowek [CD07] and enriched by Assaf [Ass15], which was presented in Section 5.3.

ηE annotates terms with their types. To do so, it takes as arguments a sort, a code of type in this sort and the term to annotate. The rules state that η-expansion is the identity for inhabitant of sorts (rule ηS), and generates λ's for inhabitants of products (rule ηP). Furthermore, a rule states that η-expansion is an idempotent operation (rule ηI).

Definition 8.2.1 (Eta-expansion rewriting rules)**.**

```
symbol ηE : (s : 𝒮) ⇒ (A : Univ s) ⇒ Lift s A ⇒ Lift s A.
"ηS" [t]          ηE _ (code _)       t ↪ t.
"ηP" [a,b,A,B,t]  ηE _ (prod a b A B) t ↪
                     λ(x : Lift a A), ηE b (B (ηE a A x)) (t (ηE a A x)).
"ηI" [a,A,t]      ηE _ _ (ηE a A t)     ↪ ηE a A t.
```

8.3 Soundness of the Encoding

8.3.1 Adapting the Type System

To prove that adding those annotations in the encoding enriches the conversion enough to simulate η-equality, we will also add those annotations in the system we are translating, just like what is done in [GN91, DHW93].

Performing η-expansion can be required for variables or if an application instantiates a type, allowing it to reduce to a product. Hence, one would like to add those tags on the variables and application rules. Hence, one could imagine having the rules:

$$\text{(var')} \quad \frac{\Gamma \vdash_\eta A : s_i}{\Gamma, x : A \vdash_\eta x^A : A}\, x \notin \mathrm{dom}(\Gamma) \qquad \text{(app')} \quad \frac{\Gamma \vdash_\eta t : (x : A) \to B \quad \Gamma \vdash_\eta u : A}{\Gamma \vdash_\eta (t\,u)^{B\left[u/x\right]} : B\left[u/x\right]}$$

But those rules do not have the property that if a term is well-typed, its subterms are well-typed with smaller trees, because of the substitution performed on B.

Fortunately, Barthe, Hatcliff and Sørensen [BHS01] introduced an induction principle for PTS embeddable in $\mathcal{C}^\infty$ (see Example 3.4.3), that can be used to ensure this property.

Theorem 8.3.1 (Induction Principle of [BHS01, Thm. 19])**.** *The relation $\blacktriangleleft$ defined by the following rules is well-founded for any PTS embeddable in $\mathcal{C}^\infty$:*

$$\begin{array}{rr}
(\Gamma; M) \blacktriangleleft (\Gamma; M\,N) & (\Gamma; N) \blacktriangleleft (\Gamma, M\,N) \\
(\Gamma, x : A; M) \blacktriangleleft (\Gamma; \lambda x^A.M) & (\Gamma; A) \blacktriangleleft (\Gamma; \lambda x^A.M) \\
(\Gamma, x : A; B) \blacktriangleleft (\Gamma; (x : A) \to B) & (\Gamma; A) \blacktriangleleft (\Gamma; (x : A) \to B) \\
(\Gamma; A) \blacktriangleleft (\Gamma, x : A, \Gamma'; M) & (\Gamma, \Gamma'; M) \blacktriangleleft (\Gamma, x : A, \Gamma'; M)
\end{array}$$

$(\Gamma; M) \blacktriangleleft (\Gamma; N)$ *if $\Gamma \vdash N : M$, M is normal and N is a variable, an application or an abstraction.*

This principle ensures us that, if we annotate the applications with normal forms, the subterm property is verified, leading to:

$$\text{(app")} \quad \frac{\Gamma \vdash_\eta t : (x : A) \to B \quad \Gamma \vdash_\eta u : A}{\Gamma \vdash_\eta (t\,u)^{B\left[u/x\right]\Downarrow} : B\left[u/x\right]}$$

Thanks to this induction principle, for PTS embeddable in $\mathcal{C}^\infty$, the set of provable sequents is not altered by the addition of those annotations. In [BHS01], they annotate variables and applications using the identity function, to construct "affiliated terms", so t labeled with A has the shape $(\lambda x^A.x)\,t$ in their setting. For their system, they prove that for any provable sequent in the source PTS, there is an "affiliated" counterpart which is provable, if all the identity functions introduced are themselves typable. In [DHW93], they restrict their study to PTS of the λ-cube, for which they annotate variables and application with their types, called "marked terms", and prove that for any provable sequent in the source PTS, there is a "marked" counterpart which is provable.

Now that we have modified a little the notion of terms, η-equivalence becomes:

Definition 8.3.2 (Eta-Conversion)**.** η-conversion *is the closure by reflexivity, symmetry and context of the η equivalence:*

$$\lambda x^A.(t\,x^A)^B \equiv_\eta t$$

8.3.2 Translation

Definition 8.3.3 (Translation of Typing with $\vdash_\eta$)**.** *Given an annotated well-typed term t in a terminating Full Pure Type System, with the rules (var') and (app'') and the conversion enriched with η, we translate t by:*
$\left\|x^A\right\| = \eta E\ |s_A|_S\ \|A\|\,\mathtt{x}$; $\quad \|s\| = \mathtt{code}\ |s|_S$; $\quad \left\|(t\,u)^A\right\| = \eta E\ |s_A|_S\ \|A\|\ (\|t\|\ \|u\|)$;
$\left\|\lambda x^A.t\right\| = \lambda(\mathtt{x}\ :\ \mathtt{Lift}\ \ |s_A|_S\ \|A\|).\|t\|$;
$\|(x : A) \to B\| = \mathtt{prod}\,|s_A|_S\ |s_B|_S\ \|A\|\ (\lambda(\mathtt{x}\ :\ \mathtt{Lift}\ |s_1|_S\ \|A\|).\|B\|)$;
s_A and s_B are respectively the sorts of A and B, and $|.|_S$ is the translation of sorts.

Since we are only considering full PTS, which are functional, the type of any term is unique modulo conversion [GN91], especially, "the" sort of a type is well-defined, since it is really unique.

One can naturally extend this translation to the contexts, with

Definition 8.3.4 (Context Translation)**.** *If $\Gamma = y_1 : A_1, \ldots, y_n : A_n$, then its translation is* $\|\Gamma\| = y_1 : \mathtt{Lift}\ |s_{A_1}|_S\ \|A_1\|, \ldots, y_n : \mathtt{Lift}\ |s_{A_n}|_S\ \|A_n\|$.

The translation of sorts remains abstract in this chapter, since all the results presented here are independent of the details of this translation. However, it will be made explicit, when it will be required, in Chapter 9. One only assumes three properties of this translation:

Assumption 8.3.5.

- *If s is a sort, then* $\vdash |s|_S : \mathrm{Sort}$,
- *If $s_1 : s_2$ is an axiom of the PTS, then* $\mathtt{axiom}\ |s_1|_S \downarrow |s_2|_S$,
- *If (s_1, s_2, s_3) is a rule of the PTS, then* $\mathtt{rule}\ |s_1|_S\ |s_2|_S \downarrow |s_3|_S$.

8.3.3 Key Lemmas

The correctness of our translation relies on the preservation of conversion. This result comes from the three following lemmas:

Lemma 8.3.6 (Translation are η-long)**.** *If $\Gamma \vdash_\eta t : A$, then* $\eta E\ |s_A|_S\ \|A{\Downarrow}\|\ \|t\| \leftrightsquigarrow^* \|t\|$.

Lemma 8.3.7 (Substitution)**.** *If t is well-typed in the context $\Gamma, x_1 : A_1, \ldots, x_n : A_n, \Gamma'$ and if $\Gamma \vdash_\eta u_1 : A_1, \ldots, \Gamma \vdash_\eta u_n : A_n$ then* $\|t\| \left[\|u_i\|/_{x_i}\right]_{i\in\{1,\ldots,n\}} \leftrightsquigarrow^* \left\|t\left[u_i/_{x_i^{A_i}}\right]_{i\in\{1,\ldots,n\}}\right\|$.

Lemma 8.3.8 (Reduction)**.** *If $\Gamma \vdash_\eta t : A$ and $t \rightsquigarrow_{\beta\eta} t'$, then* $\|t\| \leftrightsquigarrow^* \|t'\|$.

We prove those three lemmas, in this order, by a mutual induction on the combination of the subterm ordering and reduction on a multiset of terms (this multiset is of size at most 2), called "measure" in the proofs.

- For Lemma 8.3.6, the measure is the multiset $\{\!|A, t|\!\}$;
- For Lemma 8.3.7, the measure is the multiset $\left\{\!\!\left|t, t\left[u_i/_{x_i^{A_i}}\right]_{i\in\{1,\ldots,n\}}\right|\!\!\right\}$;
- For Lemma 8.3.8, the measure is the multiset $\{\!|t|\!\}$.

Proof of Lemma 8.3.6. We use the multiset $\{\!|A, t|\!\}$ as the measure. If the normal form of A is a sort, then its translation is $(\mathtt{code}\ |s|_S)$ and one can conclude using the rule ηS. Otherwise, we proceed by case on t:

- If $t = x^B$, then $\eta E\ |s_A|_S\ \|A\Downarrow\|\ \|t\| = \eta E\ |s_A|_S\ \|A\Downarrow\|\ (\eta E\ |s_B|_S\ \|B\|\ x) \rightsquigarrow_{\eta I} \|t\|$.
- If $t = (u\,v)^B$, then it is again a direct consequence of the rule ηI
- If $t = \lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u$, with u not a λ-abstraction.
  Since the rule $(abstr)$ states that whenever introducing a λ, the type is a product, and since the β-reducts of products are still products, there is a C such that: $A\Downarrow = (x_1 : B_1\Downarrow) \to \dots \to (x_n : B_n\Downarrow) \to C$. We denote by s_i the sort of $(x_i : B_i\Downarrow) \to \dots \to (x_n : B_n\Downarrow) \to C$. We have:

$$
\begin{aligned}
&\eta E\ |s_A|_S\ \|A\Downarrow\|\ \|t\| \\
&=\ \eta E\ |s_A|_S\ [\mathrm{prod}\ |s_{B_1}|_S\ |s_2|_S\ \|B_1\Downarrow\|\ (\lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\Downarrow\|). \\
&\quad \dots \mathrm{prod}\ |s_{B_n}|_S\ |s_C|_S\ \|B_n\Downarrow\|\ (\lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\Downarrow\|).\ \|C\|) \dots)] \\
&\quad [\lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\|) \dots \lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\|).\ \|u\|] \\
&\rightsquigarrow_{\eta P}\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\Downarrow\|).\eta E\ |s_2|_S\ [(\lambda x_1 \dots \|C\|)(\eta E\ |s_1|_S\ \|B_1\Downarrow\|\ x_1)] \\
&\quad [(\lambda x_1 \dots \|u\|)(\eta E\ |s_{B_1}|_S\ \|B_1\Downarrow\|\ x_1)] \\
&\rightsquigarrow^2_{\beta}\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\Downarrow\|).\eta E\ |s_2|_S\ (\mathrm{prod}\ |s_{B_2}|_S\ |s_3|_S\ \|B_2\Downarrow\| \dots \|C\|)\,\sigma\,(\lambda x_2 \dots \|u\|)\sigma \\
&\text{with } \sigma = \left[\eta E\ |s_{B_1}|_S\ \|B_1\Downarrow\|\ x_1/_{x_1}\right]
\end{aligned}
$$

$$
\begin{aligned}
&(\rightsquigarrow_{\eta P} \rightsquigarrow^2_{\beta})^{n-1}\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\Downarrow\|) \dots \lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\Downarrow\|).\eta E\ |s_C|_S\ \|C\|\,\tau\ \|u\|\,\tau \\
&\text{with } \tau = \left[\eta E\ |s_{B_i}|_S\ \|B_i\Downarrow\|\ x_i/_{x_i}\right]_{i\in\{1,\dots,n\}} \\
&(\leftrightsquigarrow^*_{Lem.8.3.7})^2\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\Downarrow\|) \dots \lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\Downarrow\|).\eta E\ |s_C|_S\ \|C\tau'\|\ \|u\tau'\| \\
&\text{with } \tau' = \left[x_i^{B_i\Downarrow}/_{x_i^{B_i}}\right]_{i\in\{1,\dots,n\}} \\
&\leftrightsquigarrow^*_{IH}\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\Downarrow\|) \dots \lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\Downarrow\|).\ \|u\tau'\| \\
&(\leftrightsquigarrow^*_{Lem.8.3.8})^n\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\|) \dots \lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\|).\ \|u\tau'\| \\
&(\leftrightsquigarrow^*_{Lem.8.3.8})^*\ \lambda(x_1 : \texttt{Lift}\ |s_{B_1}|_S\ \|B_1\|) \dots \lambda(x_n : \texttt{Lift}\ |s_{B_n}|_S\ \|B_n\|).\ \|u\| \\
&=\ \left\|\lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u\right\|
\end{aligned}
$$

To conclude, we have to show that all the calls to one of the lemmas we are currently proving are really done with a strictly smaller measure. The measure of the current call is $\left\{\!\left\{A, \lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u\right\}\!\right\}$.

- Let us start by the call to Lemma 8.3.7 with measure $\left\{\!\left\{C, C\left[\overrightarrow{x^{B_i\Downarrow}/_{x_i^{B_i}}}\right]\right\}\!\right\}$. C is a subterm of the normal form of A, hence, it is strictly smaller than A, and since C is in normal form, so are all its annotations, so $C = C\left[\overrightarrow{x^{B_i\Downarrow}/_{x_i^{B_i}}}\right]$. Hence, this call is performed with a strict decrease of the measure.
- There is a second call to Lemma 8.3.7, this time with measure $\left\{\!\left\{u, u\left[\overrightarrow{x^{B_i\Downarrow}/_{x_i^{B_i}}}\right]\right\}\!\right\}$. u is a subterm of $\lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u$ and $u\left[x^{B_i\Downarrow}/_{x_i^{B_i}}\right]_{i\in\{1,\dots,n\}}$ is a reduct of u, so the measure is strictly decreasing when performing this call.

- Then there is a recursive call to Lemma 8.3.6, with measure $\left\{\!\!\left\{ C\left[\overrightarrow{x^{B_i\Downarrow}/x_i^{B_i}}\right], u\left[\overrightarrow{x^{B_i\Downarrow}/x_i^{B_i}}\right] \right\}\!\!\right\}$, and it has already been explained why those two terms are strictly smaller than A and $\lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u$ respectively.

- There are then calls to Lemma 8.3.8 with the measures $\{\!|B_i|\!\}$, which are strictly smaller than $\lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u$, since they are strict subterms of it.

- Finally, there is a call to Lemma 8.3.8 with the measure $\{\!|u|\!\}$, which is a strict subterm of $\lambda x_1^{B_1} \dots \lambda x_n^{B_n}.u$. □

Proof of Lemma 8.3.7. There, the measure is $\left\{\!\!\left\{ t, t\left[u_i/x_i^{A_i}\right]_{i\in\{1,\dots,n\}} \right\}\!\!\right\}$. Depending on the shape of t, we have:

- If t is a sort, the substitution does not have any impact.

- If $t = x_i^{A_i}$, $\|t\| = \eta E\ |s_{A_i}|_S\ \|A_i\|\ x_i$, so $\|t\|\left[\overrightarrow{\|u_i\|/x_i}\right] = \eta E\ |s_{A_i}|_S\ \|A_i\|\ \|u_i\|$. In this case, the measure is $\left\{\!\!\left\{ x_i^{A_i}, u_i \right\}\!\!\right\}$. By Lemma 8.3.8, $\|A_i\| \rightsquigarrow^* \|A_i\Downarrow\|$, and this lemma can be used, since A_i is a strict subterm of $x_i^{A_i}$. $\{\!|A_i, u_i|\!\}$ is strictly smaller than $\left\{\!\!\left\{ x_i^{A_i}, u_i \right\}\!\!\right\}$, since A_i is a strict subterm of $x_i^{A_i}$. So, one can conclude by Lemma 8.3.6 that $\|t\|\left[\overrightarrow{\|u_i\|/x_i}\right] \rightsquigarrow^* \|u_i\|$.

- If $t = y^B$ with $y \notin \{x_i\}_i$, then $\|t\| = \eta E\ |s_B|_S\ \|B\|\ y$,so

$$\begin{aligned}\|t\|\left[\overrightarrow{\|u_i\|/x_i}\right] &= \eta E\ |s_B|_S\ \|B\|\left[\overrightarrow{\|u_i\|/x_i}\right] y \\ &\rightsquigarrow^*_{IH} \eta E\ |s_B|_S\ \left\|B\left[\overrightarrow{u_i/x_i^{A_i}}\right]\right\| y \\ &= \left\|y^{B\left[\overrightarrow{u_i/x_i}\right]}\right\| = \left\|t\left[\overrightarrow{u_i/x_i}\right]\right\|.\end{aligned}$$

- If $t = \lambda y^B.v$, then $\|t\| = \lambda(y : \texttt{Lift}\ |s_B|_S\ \|B\|).\|v\|$, so

$$\begin{aligned}\|t\|\left[\overrightarrow{\|u_i\|/x_i}\right] &= \lambda(y : \texttt{Lift}\ |s_B|_S\ \|B\|\left[\overrightarrow{\|u_i\|/x_i}\right]).\|v\|\left[\overrightarrow{\|u_i\|/x_i}\right] \\ &\rightsquigarrow^*_{IH} \lambda(y : \texttt{Lift}\ |s_B|_S\ \left\|B\left[\overrightarrow{u_i/x_i}\right]\right\|).\left\|v\left[\overrightarrow{x_i/u_i}\right]\right\| = \left\|(\lambda y^B.v)\left[\overrightarrow{x_i/u_i}\right]\right\|\end{aligned}$$

The other cases are similar to the previous two. □

Proof of Lemma 8.3.8. We use $\{\!|t|\!\}$ as the measure. If the reduction is not at the head of t, then the result follows by the induction hypothesis.

Otherwise, the reduction occurs at the head of the term. It can be either η or β reduction.

(η) Then $t = \lambda x^A.(u\,x^A)^B$. Since B is the annotation of an application, the rule (app'') ensures us that it is in normal form. u is either a variable, an application or a λ-abstraction, in every case $\|t\| = \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\|u\|\ (\eta E\ |s_A|_S\ \|A\|\ x))$.

- If $u = y^C$, then $C\Downarrow = (x : A\Downarrow) \to B$.

$$\begin{aligned}\|u\| &= \eta E\ |s_C|_S\ \|C\|\ y\\ &\leftsquigarrow^*_{IH}\ \eta E\ |s_C|_S\ \|(x : A\Downarrow) \to B\|\ y\end{aligned}$$

C is a strict subterm of $\lambda x^A.(y^C\ x^A)^B$

$$= \eta E\ |s_C|_S\ (\mathrm{prod}\ |s_A|_S\ |s_B|_S\ \|A\Downarrow\|\ (\lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\ \|B\|))\ y$$

$\rightsquigarrow_{\eta P}\rightsquigarrow_\beta\leftsquigarrow^*_{Lemma 8.3.7}\ \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\eta E\ |s_B|_S\ \|B\|\ (y\,(\eta E\ |s_A|_S\ \|A\Downarrow\|\ x))$

When we instantiate $\|t\|$ with this reduct of $\|u\|$, we get:

$$\begin{aligned}\|t\| &\rightsquigarrow_\beta\ \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\\ &\qquad (\eta E\ |s_B|_S\ \|B\|\ \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right]\ (y\,(\eta E\ |s_A|_S\ \|A\Downarrow\|\ (\eta E\ |s_A|_S\ \|A\|\ x))))\\ &\rightsquigarrow^2_{\eta I}\ \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right]\ (y\,(\eta E\ |s_A|_S\ \|A\|\ x))\\ (\leftsquigarrow^*_{IH})^2\ &\lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\eta E\ |s_B|_S\ \|B\|\ \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right]\ (y\,(\eta E\ |s_A|_S\ \|A\Downarrow\|\ x))\end{aligned}$$

A is a strict subterm of $\lambda x^A.(y^C\ x^A)^B$

$$= \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\eta E\ |s_B|_S\ \|B\|\ \left[\left\|x^A\right\|/_x\right]\ (y\,(\eta E\ |s_A|_S\ \|A\Downarrow\|\ x))$$

$$\leftsquigarrow^*_{Lem.8.3.7}\ \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\eta E\ |s_B|_S\ \left\|B\left[x^A/_{x^A}\right]\right\|\ (y\,(\eta E\ |s_A|_S\ \|A\Downarrow\|\ x))$$

B is a strict subterm of $\lambda x^A.(y^C\ x^A)^B$ and $B\left[x^A/_{x^A}\right] = B$

$$= \|u\|$$

- If $u = (v\,w)^{(x:A\Downarrow)\to B}$.

$$\begin{aligned}\|u\| &= \eta E\ \left|s_{(x:A\Downarrow)\to B}\right|_S\ \|(x : A\Downarrow) \to B\|\ (\|v\|\ \|w\|)\\ &\rightsquigarrow_{\eta P}\ \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\eta E\ |s_B|_S\ \|B\|\ (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A\Downarrow\|\ x))\end{aligned}$$

Instantiating $\|t\|$ with this reduct of $\|u\|$ gives:

$$\begin{aligned}\|t\| \rightsquigarrow_\beta\ &\lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ (\eta E\ |s_B|_S\ \|B\|\ \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right]\\ &\qquad (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A\Downarrow\|\ (\eta E\ |s_A|_S\ \|A\|\ x))))\end{aligned}$$

since v and w do not contain x free.

$$\rightsquigarrow_{\eta I}\ \lambda(x : \mathtt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \left[\eta E\ |s_A|_S\ \|A\|\ x/_x\right]\ (\|v\|\ \|w\|\ (\eta E\ |s_A|_S\ \|A\|\ x))$$

$(\leftsquigarrow^*_{IH})^2\ \leftsquigarrow^*_{Lem.8.3.7}\ \|u\|$

with the same explanations as in the previous case.

- If $u = \lambda y^C.v$, then, typability ensures that $A \leftsquigarrow^* C$, so $C\Downarrow = A\Downarrow$ and $\|u\| = \lambda(y :$

$\texttt{Lift}\ |s_C|_S\ \|C\|).\ \|v\|$. Hence,

$$\begin{aligned}
\|t\| &\rightsquigarrow_\beta \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \|v\|\ \left[(\eta E\ |s_A|_S\ \|A\|\ x)/_y\right] \\
&= \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \|v\|\ \left[\|x^A\|/_y\right] \\
&\leftrightsquigarrow^*_{Lem.8.3.7} \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).\eta E\ |s_B|_S\ \|B\|\ \left\|v\left[x^A/_{y^C}\right]\right\|
\end{aligned}$$

$((\lambda y^C.v)\, x^A)^B$ is a subterm of t, so v is a subterm of t and $v\left[x^A/_{y^C}\right]$ is a subterm of a reduct of t.

$$\leftrightsquigarrow^*_{Lem.8.3.6} \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).\ \left\|v\left[x^A/_{y^C}\right]\right\|$$

$((\lambda y^C.v)\, x^A)^B$ is a subterm of t, so B is a subterm of t and $v\left[x^A/_{y^C}\right]$ is a subterm of a reduct of t. Furthermore, B is in normal form, since it is the annotation of an application.

$$\begin{aligned}
&\leftrightsquigarrow^*_{IH} \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\Downarrow\|).\ \left\|v\left[x^A\Downarrow/_{y^C}\right]\right\| \\
&= \lambda(x : \texttt{Lift}\ |s_A|_S\ \|C\Downarrow\|).\ \left\|v\left[x^C\Downarrow/_{y^C}\right]\right\| \\
&\leftrightsquigarrow^*_{IH} \lambda(x : \texttt{Lift}\ |s_A|_S\ \|C\|).\ \left\|v\left[x^C/_{y^C}\right]\right\| \\
&=_\alpha \|u\|
\end{aligned}$$

(β) Then $t = ((\lambda x^A.v)\, w)^B$ and $t' = v\,[w/_x]$. We have :

$$\begin{aligned}
\|t\| &= \eta E\ |s_B|_S\ \|B\|\ ((\lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).\ \|v\|)\ \|w\|) \\
&\rightsquigarrow_\beta \eta E\ |s_B|_S\ \|B\|\ \|v\|\ \left[\|w\|/_x\right] \\
&\leftrightsquigarrow^*_{Lem.8.3.7} \eta E\ |s_B|_S\ \|B\|\ \|v\,[w/_{x^A}]\|
\end{aligned}$$

v and $v\,[w/_{x^A}]$ are respectively subterm and reduct of t, hence Lemma 8.3.7 applies.

$\leftrightsquigarrow^*_{Lem.8.3.6}\ \|v\,[w/_{x^A}]\|$

B and $v\,[w/_{x^A}]$ are respectively subterm and reduct of t, hence Lemma 8.3.6 applies. □

8.3.4 Soundness Result

Preservation of the conversion (Lemma 8.3.8) is the key element to prove the soundness of the translation. However, we have chosen in Chapter 5 to present the type system with a joinability relation (Definition 4.1.8), rather than the convertibility one (Definition 4.1.10), since it is closer to what is really implemented and it was useful to prove the consistency of $\lambda\Pi$-modulo rewriting (Corollary 5.2.5). This choice means that it is required not only to check that the extremity of the conversion has a sort, but one has to check this property for every peak occurring in the conversion (see Remark 4.1.11). We will assume that this property is verified for the convertibility deduced from Lemma 8.3.8, so a conversion in the PTS with η is translated in $\lambda\Pi$-modulo rewriting simply by a sequence of conversions. With this hypothesis, we can conclude that our translation preserves typing:

Theorem 8.3.9 (Soundness of the translation)**.** *Let* Π *be a proof of* $\Gamma \vdash_\eta t : A$ *in the type system* $\vdash_\eta$ *with* $(conv')$, (var') *and* (app'').

Assuming that if $T \leftrightsquigarrow^*_{\beta\eta} U$, $\Gamma \vdash_\eta T : s$ *and* $\Gamma \vdash_\eta U : s$ *then there is a sequence* $T_2, \ldots, T_n$ *of terms of* $\lambda\Pi$*-modulo rewriting such that* $\|T\| \downarrow T_2 \downarrow \ldots \downarrow T_n \downarrow \|U\|$, *and* $\|\Gamma\| \vdash T_i : \mathtt{Univ}\, s$ *for all* i.

Assuming that the translation of sorts verifies the Assumption 8.3.5, either A *is a sort and* $\|\Gamma\| \vdash \|t\| : \mathtt{Univ}\ |A|_S$, *or there is a* s *such that* $\Gamma \vdash_\eta A : s$ *and* $\|\Gamma\| \vdash \|t\| : \mathtt{Lift}\ |s|_S\ \|A\|$.

Proof. In all this proof, one will intensively use Assumption 8.3.5, referred by "hyp. $|.|_S$".

Before doing the proof, let us enlighten a few useful subproofs:

$$U = \dfrac{\dfrac{}{\vdash \mathtt{Univ} : (s : \mathrm{Sort}) \Rightarrow \star}\,(\mathrm{sig}) \qquad \dfrac{\text{hyp. } |.|_S}{\vdash |s|_S : \mathrm{Sort}}}{\vdash \mathtt{Univ}\ |s|_S : \star}\,(\mathrm{app''})$$

$$L = \dfrac{\dfrac{}{\vdash \mathtt{Lift} : (s : \mathrm{Sort}) \Rightarrow \mathtt{Univ}\, s \Rightarrow \star}\,(\mathrm{sig}) \qquad \dfrac{\text{hyp. } |.|_S}{\vdash |s|_S : \mathrm{Sort}}}{\vdash \mathtt{Lift}\ |s|_S : \mathtt{Univ}\ |s|_S \Rightarrow \star}\,(\mathrm{app''})$$

$$P_1 = \dfrac{\dfrac{}{\vdash \mathrm{prod} : \ldots}\,(\mathrm{sig}) \qquad \dfrac{\text{hyp. } |.|_S}{\vdash |s|_S : \mathrm{Sort}}}{\vdash \mathrm{prod}\ |s|_S : (s_2 : \mathrm{Sort}) \Rightarrow (A : \mathtt{Univ}\ |s|_S) \Rightarrow (\mathtt{Lift}\ |s|_S\ A \Rightarrow \mathtt{Univ}\, s_2) \Rightarrow \mathtt{Univ}\,(\mathtt{rule}\ |s|_S\ s_2)}\,(\mathrm{app''})$$

$$P_2 = \dfrac{\dfrac{P_1}{\vdash \mathrm{prod}\ |s|_S : \ldots} \qquad \dfrac{\text{hyp. } |.|_S}{\vdash |s_2|_S : \mathrm{Sort}}}{\vdash \mathrm{prod}\ |s|_S\ |s_2|_S : (A : \mathtt{Univ}\ |s|_S) \Rightarrow (\mathtt{Lift}\ |s|_S\ A \Rightarrow \mathtt{Univ}\ |s_2|_S) \Rightarrow \mathtt{Univ}\,(\mathtt{rule}\ |s|_S\ |s_2|_S)}\,(\mathrm{app''})$$

$$E = \dfrac{\dfrac{}{\vdash \eta E : (s : \mathrm{Sort}) \Rightarrow (A : \mathtt{Univ}\, s) \Rightarrow \mathtt{Lift}\, s\, A \Rightarrow \mathtt{Lift}\, s\, A}\,(\mathrm{sig}) \qquad \dfrac{\text{hyp. } |.|_S}{\vdash |s|_S : \mathrm{Sort}}}{\vdash \eta E\ |s|_S : (A : \mathtt{Univ}\ |s|_S) \Rightarrow \mathtt{Lift}\ |s|_S\ A \Rightarrow \mathtt{Lift}\ |s|_S\ A}\,(\mathrm{app''})$$

Now that the preliminaries are set up, let us dot the proof by induction on Π:

- If $\Pi = \dfrac{}{\vdash_\eta s_1 : s_2}$, then we have to prove $\vdash \mathtt{code}\, |s_1|_S : \mathtt{Univ}\ |s_2|_S$.

$$S = \dfrac{\dfrac{}{\vdash \mathtt{code} : (s : \mathrm{Sort}) \Rightarrow \mathtt{Univ}\,(\mathtt{axiom}\, s)}\,(\mathrm{sig}) \qquad \dfrac{\text{hyp. } |.|_S}{\vdash |s_1|_S : \mathrm{Sort}}}{\vdash \mathtt{code}\, |s_1|_S : \mathtt{Univ}\,(\mathtt{axiom}\, |s_1|_S)}\,(\mathrm{app''})$$

  Now one can use the hypothesis that $\mathtt{axiom}\ |s_1|_S \downarrow |s_2|_S$, since $s_1 : s_2$ is an axiom of the PTS, in order to use the conversion rule and conclude:

$$\dfrac{\dfrac{P_1}{\vdash \mathtt{code}\, |s_1|_S : \mathtt{Univ}\,(\mathtt{axiom}\, |s_1|_S)} \qquad \dfrac{U}{\vdash \mathtt{Univ}\ |s_2|_S : \star} \qquad \dfrac{\text{Assumption 8.3.5}}{\mathtt{axiom}\ |s_1|_S \downarrow |s_2|_S}}{\vdash \mathtt{code}\, |s_1|_S : \mathtt{Univ}\ |s_2|_S}\,(\mathrm{conv'})$$

- If $\Pi = \dfrac{\Gamma \vdash_\eta A : s_2 \quad \Gamma, x : A \vdash_\eta B : s_2}{\Gamma \vdash_\eta (x : A) \to B : s_3}$, one has by induction hypothesis $\|\Gamma\| \vdash \|A\| : \mathtt{Univ}\ |s_1|_S$ and $\|\Gamma\|, x : \mathtt{Lift}\ |s_1|_S\ \|A\| \vdash \|B\| : \mathtt{Univ}\ |s_2|_S$.

  We want $\|\Gamma\| \vdash \mathrm{prod}\ |i|_S\ |j|_S\ \|A\|\ (\lambda(x : \mathtt{Lift}\ |s_1|_S\ \|A\|).\ \|B\| : |s_3|_S$.

  It is direct using P_2, the same conversion rule as before and the induction hypotheses.

  Here, it requires $\mathtt{rule}\ |s_1|_S\ |s_2|_S \downarrow |s_3|_S$.

- The cases of abstraction and weakening are also direct.

- The $(conv')$ case is a consequence of the hypothesis that if $T \leftrightsquigarrow^*_{\beta\eta} U$ and T and U are both typable, then there is a sequence of typable $T_2, \ldots, T_n$ such that $\|T\| \downarrow T_1 \downarrow \ldots \downarrow T_n \downarrow \|U\|$ and of 8.3.8.

- In the (app'') case: $\Pi = \dfrac{\Gamma \vdash_\eta t : (x : A) \to B \quad \Gamma \vdash_\eta u : A}{\Gamma \vdash_\eta (t\,u)^{B[u/x]\Downarrow_\beta} : B\,[u/_x]}$

  By induction hypothesis, $\|\Gamma\| \vdash \|t\| : \mathtt{Lift}\ |s'|_S\ \|(x : A) \to B\|$, hence one has $\|\Gamma\| \vdash \|t\| : \mathtt{Lift}\ |s'|_S\ (\mathrm{prod}\ |s_A|_S\ |s_B|_S\ \|A\|\ (\lambda x.\ \|B\|))$. By conversion, one gets $\|\Gamma\| \vdash \|t\| : (x : \mathtt{Lift}\ |s_A|_S\ \|A\|) \Rightarrow \mathtt{Lift}\ |s_B|_S\ \|B\|$. The other induction hypothesis is $\|\Gamma\| \vdash \|u\| : \mathtt{Lift}\ |s_A|_S\ \|A\|$.

  Hence, one can apply the (app) rule and get $\|\Gamma\| \vdash \|t\|\ \|u\| : (\mathtt{Lift}\ |s_B|_S\ \|B\|)\left[\|u\|/_x\right]$. Thanks to the substitution lemma (Lemma 8.3.7), one can apply the $(conv')$ rule and gets that

  $$\|\Gamma\| \vdash \|t\|\ \|u\| : \mathtt{Lift}\ |s_B|_S\ \|(B\,[u/_x])\|\,.$$

  One can now conclude by introducing the ηE symbol using the piece of proof tree E, in order to type $\left\|(t\,u)^{B[u/x]\Downarrow_\beta}\right\| = \eta E\ |s_B|_S\ \left\|(B\,[u/_x])\Downarrow_\beta\right\|\ (\|t\|\ \|u\|)$. One must note that to introduce this symbol, one has to use the induction hypothesis on $B\,[u/_x]\Downarrow_\beta$, which is smaller than $t\,u$ thanks to the induction principle of Barthe, Hatcliff and Sørensen (Theorem 8.3.1) [BHS01].

- The remaining (var') case is analogous to the last steps of the (app'') case. □

It is not straightforward to adapt the proofs of Lemma 8.3.6, Lemma 8.3.7 and Lemma 8.3.8 in order to prove that "if $T \leftrightsquigarrow^*_{\beta\eta} U$, $\Gamma \vdash_\eta T : s$ and $\Gamma \vdash_\eta U : s$ then there is a sequence $T_2, \ldots, T_n$ of terms of $\lambda\Pi$-modulo rewriting such that $\|T\| \downarrow T_2 \downarrow \ldots \downarrow T_n \downarrow \|U\|$, and $\|\Gamma\| \vdash T_i : \mathtt{Univ}\ s$ for all i", because this would require to include to have typability hypotheses in Lemma 8.3.6, Lemma 8.3.7 and Lemma 8.3.8 and to prove them mutually with Theorem 8.3.9.

However, in the results of the translation of the Agda's standard library, I did not encountered a translation failing to type check because of a convertibility issue between terms featuring the ηE-symbol. Hence, I conjecture that it should be possible to suppress this hypothesis.

Conjecture 8.3.10 (Soundness of the translation)**.** *Let Π be a proof of $\Gamma \vdash_\eta t : A$ in the type system $\vdash_\eta$ with $(conv')$, (var') and (app''). Assuming that the translation of sorts verifies the Assumption 8.3.5, either A is a sort and $\|\Gamma\| \vdash \|t\| : \mathtt{Univ}\ |A|_S$, or there is a s such that $\Gamma \vdash_\eta A : s$ and $\|\Gamma\| \vdash \|t\| : \mathtt{Lift}\ |s|_S\ \|A\|$.*

Chapter 9

Universe Polymorphism

It is quite common to enrich PTS with *Universe Polymorphism* [HP91], which consists in allowing the user to quantify over *universe levels*, allowing to declare simultaneously a symbol for several sorts. For instance, if the sorts are $\{\mathrm{Set}_i \mid i \in \mathbb{N}\}$, then one want to declare `List` in $\forall \ell, (A : \mathrm{Set}_\ell) \to \mathrm{Set}_\ell$. Indeed, just like polymorphism was used to avoid declaring a type of lists for each type of elements, one wants to avoid one declaration of a new type of lists for each universe level.

We present here a definition of *universe polymorphism* inspired by the one given by Sozeau and Tabareau [ST14] for the proof assistant Coq. In this setting, the context contains three lists: a list Σ called signature, a list Θ of level variables, and a list Γ called local context. Both Σ and Γ contain pairs of a variable name and a type, but the variables in Γ can contain free level variables (those occurring in Θ), whereas all the level variables are bound by a prenex quantifier $\forall$ in the signature Σ. Unlike [ST14], we do not need to store constraints between universe levels, since those constraints are related to cumulativity, a feature we are not trying to encode here.

In his PhD thesis [Fér20], Férey propose an encoding of the universe polymorphism of Coq in the $\lambda\Pi$-calculus modulo rewriting. Contrary to what is presented here, his work includes cumulativity, breaking unicity of type, hence his translation translate terms of the calculus of constructions with cumulative universe polymorphism to a untyped setting (called the universe of codes), and then an operator selects in which universe each instance lives.

9.1 Uniform Universe-Polymorphic Pure Type System

In this chapter, we consider a set $\mathbb{L}$ of levels and a finite set $\mathcal{H}$ of sort constructors. Then the sorts are the ordered pairs $\{r_\ell\}_{r\in\mathcal{H},\ell\in\mathbb{L}}$.

Definition 9.1.1 (Uniform Universe-Polymorphic Full PTS)**.** *We assume functionality and totality of $\mathcal{A}$ and $\mathcal{R}$. And we also assume a form of uniformity in the hierarchy: for all $r \in \mathcal{H}$, there is a unique $r' \in \mathcal{H}$, such that for all $\ell \in \mathbb{L}$, there is a unique $\ell' \in \mathbb{L}$, such that $(r_\ell, r'_{\ell'}) \in \mathcal{A}$ and for all $r^{(1)}, r^{(2)} \in \mathcal{H}$, there is a unique $r^{(3)} \in \mathcal{H}$, such that for all $\ell_1, \ell_2, \in \mathbb{L}$, there is a unique $\ell_3 \in \mathbb{L}$ such that $(r^{(1)}_{\ell_1}, r^{(2)}_{\ell_2}, r^{(3)}_{\ell_3}) \in \mathcal{R}$.*

We denote by $\bar{\mathcal{A}}$ the function $\left\{(r, r') \in \mathcal{H}^2 \mid \exists \ell, \ell', (r_\ell, r'_{\ell'}) \in \mathcal{A}\right\}$ and for all r by $\mathcal{A}_r$ the function $\left\{(\ell, \ell') \in \mathbb{L}^2 \mid \exists r', (r_\ell, r'_{\ell'}) \in \mathcal{A}\right\}$.

Analogously $\bar{\mathcal{R}}$ is the function $\left\{(r^{(1)}, r^{(2)}, r') \in \mathcal{H}^3 \;\middle|\; \exists \ell_1, \ell_2, \ell', (r^{(1)}_{\ell_1}, r^{(2)}_{\ell_2}, r'_{\ell'}) \in \mathcal{R}\right\}$ and for all $(r^{(1)}, r^{(2)})$, $\mathcal{R}_{r^{(1)}, r^{(2)}}$ is the function $\left\{(\ell_1, \ell_2, \ell') \in \mathbb{L}^3 \;\middle|\; \exists r', (r^{(1)}_{\ell_1}, r^{(2)}_{\ell_2}, r'_{\ell'}) \in \mathcal{R}\right\}$.

Definition 9.1.2 (Universe Levels with Variables)**.** *Given a list of variable names Θ, we define the set of universe levels over Θ as the γ such that $\Theta \vdash_{UP} \gamma \,\mathrm{isLvl}$ where:*

$$\textit{(lvl)}\quad \frac{}{\Theta \vdash_{UP} \ell \ \mathrm{isLvl}}\ \ell \in \mathbb{L} \qquad \textit{(}\mathbb{L}\textit{var)}\quad \frac{}{\Theta \vdash_{UP} i \ \mathrm{isLvl}}\ i \in \Theta$$

$$\textit{(}\mathbb{L}\mathcal{A}\textit{)}\quad \frac{\Theta \vdash_{UP} \gamma \ \mathrm{isLvl}}{\Theta \vdash_{UP} \mathcal{A}_r(\gamma) \ \mathrm{isLvl}} \qquad \textit{(}\mathbb{L}\mathcal{R}\textit{)}\quad \frac{\Theta \vdash_{UP} \gamma_1 \ \mathrm{isLvl} \quad \Theta \vdash_{UP} \gamma_2 \ \mathrm{isLvl}}{\Theta \vdash_{UP} \mathcal{R}_{rr'}(\gamma_1, \gamma_2) \ \mathrm{isLvl}}$$

We have extended the function $\mathcal{A}_r$ and $\mathcal{R}_{rr'}$ to be applied not only to concrete levels, but also to variables, and recursively to terms constructed from it.

The set $\mathbb{L}^+_\Theta$ contains all those new levels. To ensure that the levels are really "new", one must not allow the application of $\mathcal{A}_r$ and $\mathcal{R}_{rr'}$ to elements of $\mathbb{L}$.

Definition 9.1.3 (Uninstantiated Level Expressions)**.** *Let $\mathbb{L}^+_\Theta$ be the smallest subset such that:*

$$\mathbb{L}^+_\Theta = \Theta \cup \left\{ \mathcal{A}_r(l) \;\middle|\; r \in \mathcal{H}, l \in \mathbb{L}^+_\Theta \right\} \cup \left\{ \mathcal{R}_{rr'}(l_1, l_2) \;\middle|\; r, r' \in \mathcal{H}, (l_1, l_2) \in (\mathbb{L} \cup \mathbb{L}^+_\Theta)^2 \setminus \mathbb{L}^2 \right\}.$$

Hence, $\mathbb{L}^+_\Theta$ is the free algebra constructed with the $\mathcal{A}_r$, $\mathcal{R}_{rr'}$, $\ell \in \mathbb{L}$ and $i \in \Theta$, quotiented by the definition of the functions $\mathcal{A}_r$ and $\mathcal{R}_{rr'}$, when applied to concrete levels ($\ell \in \mathbb{L}$).

If γ is a concrete level, $\mathcal{A}_r(\gamma)$ designates the only concrete γ' level such that $(r_\gamma, r_{\gamma'}) \in \mathcal{A}$, whereas if $\gamma \in \mathbb{L}^+_\Theta$, it designates a new level we just introduced. With those levels, one can define a new PTS with the hierarchy of levels enriched to $\mathbb{L} \cup \mathbb{L}^+_\Theta$. So, if i is a level variable, $\mathcal{A}_r(i)$ is a new identifier to designate a level. The value of the application of the function $\mathcal{A}_r$ to the variable i is nothing else than $\mathcal{A}_r(i)$ itself.

Definition 9.1.4 (Typing Rules of Uniform Universe-Polymorphic Full PTS)**.** *The typing rules are:*

$$\textit{(ax)}\quad \frac{\Theta \vdash_{UP} \gamma \ \mathrm{isLvl}}{[]; \Theta; [] \vdash_{UP} r_\gamma : r'_{\mathcal{A}_r(\gamma)}}\ (r, r') \in \bar{\mathcal{A}} \qquad \textit{(sig)}\quad \frac{\Sigma; \Theta; [] \vdash_{UP} A : r_\gamma}{\Sigma, x : \forall\Theta.A; \Theta'; [] \vdash_{UP} x : \forall\Theta.A}\ x \notin \mathrm{dom}(\Sigma, \Gamma)$$

$$\textit{(var)}\quad \frac{\Sigma; \Theta; \Gamma \vdash_{UP} A : r_\gamma}{\Sigma; \Theta; \Gamma, x : A \vdash_{UP} x : A}\ x \notin \mathrm{dom}(\Sigma, \Gamma) \qquad \textit{(conv)}\quad \frac{\Sigma; \Theta; \Gamma \vdash_{UP} t : A \quad \Sigma; \Theta; \Gamma \vdash_{UP} B : r_\gamma}{\Sigma; \Theta; \Gamma \vdash_{UP} t : B}\ A \downarrow_\beta B$$

$$\textit{(abs)}\quad \frac{\Sigma; \Theta, \Gamma \vdash_{UP} (x : A) \to B : r_\gamma \quad \Sigma; \Theta; \Gamma, x : A \vdash_{UP} t : B}{\Sigma; \Theta; \Gamma \vdash_{UP} \lambda x^A.t : (x : A) \to B}$$

$$\textit{(app)}\quad \frac{\Sigma; \Theta; \Gamma \vdash_{UP} t : (x : A) \to B \quad \Sigma; \Theta; \Gamma \vdash_{UP} u : A}{\Sigma; \Theta; \Gamma \vdash_{UP} t\, u : B\left[u/_x\right]}$$

$$\textit{(inst)}\quad \frac{\Sigma; \Theta; \Gamma \vdash_{UP} t : \forall[i_1, \ldots, i_n].A \quad \Theta \vdash_{UP} \gamma_1 \ \mathrm{isLvl} \quad \ldots \quad \Theta \vdash_{UP} \gamma_n \ \mathrm{isLvl}}{\Sigma; \Theta; \Gamma \vdash_{UP} t[\gamma_1, \ldots, \gamma_n] : A\left[\gamma_k/_{i_k}\right]_k}$$

$$\textit{(prod)}\quad \frac{\Sigma; \Theta; \Gamma \vdash_{UP} A : r_\gamma \quad \Sigma; \Theta; \Gamma, x : A \vdash_{UP} B : r'_{\gamma'}}{\Sigma; \Theta; \Gamma \vdash_{UP} (x : A) \to B : r''_{\mathcal{R}_{s,s'}(\gamma,\gamma')}}(r, r', r'') \in \bar{\mathcal{R}}$$

$$\textit{(ctx-weak)}\quad \frac{\Sigma; \Theta; \Gamma \vdash_{UP} A : r_\gamma \quad \Sigma; \Theta; \Gamma \vdash_{UP} t : B}{\Sigma; \Theta; \Gamma, x : A \vdash_{UP} t : B}\ x \notin \Sigma, \Gamma$$

$$\textit{(sig-weak)}\quad \frac{\Sigma; \Theta; [] \vdash_{UP} A : r_\gamma \quad \Sigma; \Theta'; [] \vdash_{UP} t : B}{\Sigma, x : \forall\Theta.A; \Theta'; \Gamma \vdash_{UP} t : B}\ x \notin \Sigma, \Gamma$$

In all those typing rules, $r, r' \in \mathcal{H}$ and $i, x \in \mathcal{V}$.

Just like what we did in the previous chapter, the products are denoted with a simple arrow and the annotations of λ-abstractions are superscript, to distinguish between the terms of a universe-polymorphic PTS and the one of $\lambda\Pi$-modulo rewriting.

One typical case of use is to have only one hierarchy: $\mathcal{H} = \{\text{Set}\}$ and to use natural numbers for levels: $\mathbb{L} = \mathbb{N}$. But we do not want to restrict ourselves to have only one hierarchy, since some proof assistants feature several. For instance, in Agda and Coq, there are two hierarchies, called Set and Prop, and Type and SProp respectively.

The two rules modifying the signature Σ (sig) and ($sig - weak$) allows to completely renew the set Θ of names of local level variables. Changing this set during the proof is not necessary, however, without this renewal of Θ, all the symbols in the signature would have been quantified over the same set Θ, no matter which variables occur really in it.

The universe polymorphism we are interested in is purely prenex. Furthermore, universally quantified types are not typed themselves and are only inhabited by variables. This form of universe polymorphism only provides ease of use, but it does not allow to prove more statements, meaning that it does not compromise the consistency of the logic.

Let $P = (\mathbb{L}, \mathcal{H}, \mathcal{A}, \mathcal{R})$ be a uniform universe polymorphic full PTS and Θ be a subset of $\mathcal{V}$.

To show this consistency, one can construct a new PTS $(\mathcal{S}^\Theta, \mathcal{A}^\Theta, \mathcal{R}^\Theta)$ simply by adding a brand new level for every expression containing a level variable. So we will close the set Θ of variables by the application of the functions $\mathcal{A}_r$ and $\mathcal{R}_{rr'}$.

Definition 9.1.5 (PTS with the Added Levels)**.** *Let P^Θ be the PTS:*

$$\mathcal{S}^\Theta = \left\{ r_l \;\middle|\; r \in \mathcal{H}, l \in \mathbb{L} \cup \mathbb{L}^+_\Theta \right\}; \qquad \mathcal{A}^\Theta = \mathcal{A} \cup \left\{ \left(r_l, r'_{\mathcal{A}_r(l))} \right) \;\middle|\; (r, r') \in \bar{\mathcal{A}}, l \in \mathbb{L}^+_\Theta \right\}$$

$$\mathcal{R}^\Theta = \mathcal{R} \cup \left\{ \left(r_{l_1}, r'_{l_2}, r''_{\mathcal{R}_{ss'}(l_1,l_2))} \right) \;\middle|\; (r, r', r'') \in \bar{\mathcal{R}}, (l_1, l_2) \in (\mathbb{L} \cup \mathbb{L}^+)^2 \setminus \mathbb{L}^2 \right\}$$

The embedding of this newly-constructed PTS in the original one is defined just by interpreting level variables. Then, using this interpretation of the variables, one can mimic the proofs done using universe polymorphism in the original PTS.

However, in the universe-polymorphic type system (Definition 9.1.4) the variables of the signature can be instantiated with levels, and be of the form $y[l_1, \ldots, l_n]$. In order to avoid technical but quite meaningless renaming of all the variables, when proving that the same proof tree can be transformed to eliminate universe polymorphism, we will do the choice to enrich the set of variables with all those instances.

Definition 9.1.6 (Enlarged Set of Variables)**.**

$$\mathcal{V}^+ = \mathcal{V} \cup \left\{ y[l_1, \ldots, l_n] \;\middle|\; y \in \mathcal{V}, n \in \mathbb{N}, (l_1, \ldots, l_n) \in (\mathbb{L} \cup \mathbb{L}^+_\mathcal{V})^n \right\}$$

With those definitions, it becomes quite straightforward to prove that adding a purely prenex form of universe polymorphism, as in Definition 9.1.4, is conservative over the underlying PTS, in the sense that one can prove syntactically the same conclusion in the PTS as in the universe polymorphic version, and in a context which is quite similar, since it contains the concatenation of instances of the variables which were originally universally quantified (those in the "signature" Σ) and of the "local context" Γ.

Proposition 9.1.7 (Conservativity of the universe polymorphism)**.**

a. There is an embedding from P^Θ to the underlying PTS of P.

b. If $\Sigma; \Theta; \Gamma \vdash_{\mathcal{UP}} t : A$ in P and A is not a universal quantification, then there is a $\bar{\Sigma} \subset \left\{ x[l_1, \ldots, l_n] : A' \;\middle|\; x : \forall[y_1, \ldots, y_n].A \in \Sigma, A' = A \left[l_i / y_i \right]_{i=1\ldots n} \text{ and all } l_i \in \mathbb{L} \cup \mathbb{L}^+_\Theta \right\}$ such that $\bar{\Sigma}, \Gamma \vdash_{P^\Theta} t : A$ using the enriched set of variables $\mathcal{V}^+$.

Proof sketch.

a. The embedding consists in just choosing a level for each variable in Θ.

b. Since A is not a universal quantification, in the proof of $\Sigma;\Theta;\Gamma \vdash_{\text{UP}} t : A$, all the (sig) are followed directly by an arbitrary number of weakenings and an application of the rule $(inst)$. The weakenings can be anticipated and to create a proof in P^{Θ}, the (sig) and $(inst)$ are compressed in a single introduction of a variable of $\bar{\Sigma}$. □

In a PTS, if $\Gamma \vdash t : A$, then there is a sort s such that $A = s$ or $\Gamma \vdash A : s$. In a full PTS, $\mathcal{A}$ is a total function, hence, all sorts inhabit a sort, allowing us to refer to s as "the sort of A". However, in the presentation of universe polymorphism of Definition 9.1.4, this property is lost because universally quantified types have no type. To overcome this issue, we assign artificially a type to those quantified types, using a brand new sort Sort_ω, which is not typable, is the type of no sort and over which one cannot quantify. Its only purpose is to make "the sort of A" well-defined whenever A is inhabited. It must be noted that Sort is not in $\mathcal{H}$ and ω is not a level.

9.2 Encoding Universe-Polymorphic PTS

To encode *Uniform Universe-Polymorphic Full PTS*, one introduces a symbol `sortOmega` and a quantification symbol $\forall_{\mathbb{L}}$ which takes as first argument the family of sorts (indexed with its level) in which the term will live once instantiated with its level. The definition of the decoding function `Lift` is enriched with a new rule, specifying its behaviour when applied to a $\forall_{\mathbb{L}}$.

Definition 9.2.1 (Encoding of Universal Quantification on Levels)**.**

```
constant sortOmega : 𝒮.
constant ∀𝕃 : (f : (𝕃 ⇒ 𝒮)) ⇒ ((l : 𝕃) ⇒ Univ (f l)) ⇒ Univ sortOmega.
Lift _ (∀𝕃 f t) ↪ (l : 𝕃) ⇒ Lift (f l) (t l).
```

For instance, the encoding of $\forall \ell, \text{Set}_\ell$ is `∀𝕃 (λ l, axiom (set l)) (λ l, code (set l))`, if `set` is a sort constructor in the encoding. And its decoding (when applying `Lift sortOmega`) reduces, as expected, to `(l:𝕃) ⇒ Univ (set l)`.

Example 9.2.2. *Consider the system* $\mathcal{H} = \{r, \rho\}$, $\mathcal{A} = \{(A_i, r_{ax_A(i)}) \mid A \in \mathcal{H}\}$ *and* $\mathcal{R} = \{(A_i, B_j, B_{ru(i,j)}) \mid A, B \in \mathcal{H}\}$, *with* ax_r, ax_ρ *and* ru *three functions remaining abstract here.* ru *could be indexed by two sort hierarchies, for ease of readability, we have chosen not present such a general case.*

We first introduce one symbol for each sort constructor (i.e. each element of $\mathcal{H}$*):*

```
constant r : 𝕃 ⇒ 𝒮.                    constant ρ : 𝕃 ⇒ 𝒮.
```

We then define a new function for the axioms of each hierarchy:

```
symbol axiom : 𝒮 ⇒ 𝒮.
symbol ax_r : 𝕃 ⇒ 𝕃.                    symbol ax_ρ : 𝕃 ⇒ 𝕃.
[i] axiom (r i) ↪ r (ax_r i).           [i] axiom (ρ i) ↪ r (ax_ρ i).
```

And a new function for each pair of sort constructors in the hierarchy associated to a rule in $\mathcal{R}$*:*

```
symbol rule : 𝒮 ⇒ 𝒮 ⇒ 𝒮. symbol ru : 𝕃 ⇒ 𝕃 ⇒ 𝕃.
[i,j] rule (r i) (r j) ↪ r (ru i j).
[i,j] rule (r i) (ρ j) ↪ ρ (ru i j).
[i,j] rule (ρ i) (r j) ↪ r (ru i j).
[i,j] rule (ρ i) (ρ j) ↪ ρ (ru i j).
```

Just like in this example, a constant of type $\mathbb{L} \Rightarrow \mathcal{S}$ and one of type $\mathcal{S} \Rightarrow \mathcal{S}$ are added in the signature for each element of $\mathcal{H}$, and symbols of type $\mathcal{S} \Rightarrow \mathcal{S} \Rightarrow \mathcal{S}$ are added for each element of $\mathcal{H}^2$.

We can use those new symbols to define the translation function.

Definition 9.2.3 (Translation of Typing with $\vdash_{\text{UP}}$)**.** *We translate well-typed terms of a Universe Polymorphic Full Pure Type System by:*

$\|x\| = \mathtt{x}$; $\quad \|r_\ell\| = \mathtt{code}\ |r_\ell|_S$; $\quad \|t\,u\| = \|t\|\ \|u\|$;

$\left\|\lambda x^A.t\right\| = \lambda(\mathtt{x}\ \mathtt{:}\ \mathtt{Lift}\ \ |s_A|_S\ \|A\|).\|t\|$;

$\|(x : A) \to B\| = \mathtt{prod}\,|s_A|_S\ |s_B|_S\ \|A\|\ (\lambda(\mathtt{x}\ \mathtt{:}\ \mathtt{Lift}\ |s_A|_S\ \|A\|).\|B\|)$;

$\|\forall[\ell_1, \ldots, \ell_n], A\| = \forall_{\mathbb{L}}\ (\lambda(\ell_1 : \mathbb{L}).\mathtt{sortOmega})\ (\lambda(\ell_1 : \mathbb{L}).\forall_{\mathbb{L}}\ \ldots\ (\lambda(\ell_n : \mathbb{L}).|s_A|_S)\ (\lambda\ell_n : \mathbb{L}.\|A\|)\ldots)$;

$\|A[\gamma_1, \ldots, \gamma_n]\| = \|A\|\ |\gamma_1|_L \ldots |\gamma_n|_L$.

The translation of sorts is $|\mathrm{Sort}_\omega|_S = \mathtt{sortOmega}$, $|r_\gamma|_S = \mathtt{r}\ |\gamma|_L$.

And the translation of levels is $|i|_L = i$ *if* $i \in \mathcal{V}$;

$|\mathcal{A}_r(\ell)|_L = \mathtt{ax\_r}\ |\ell|_L$ *and* $|\mathcal{R}_{rr'}(\ell_1, \ell_2)|_L = \mathtt{ru\_rr'}\ |\ell_1|_L\ |\ell_2|_L$.

Wherever they are used, s_A *and* s_B *are respectively the sorts of* A *and* B.

It can be noted that the translation $|\ell|_L$ for $\ell \in \mathbb{L}$ is not given, since in general the number of level is infinite, hence, we do not want to introduce one new symbol per level. This translation must be crafted case by case, we propose in Section 9.4 such a translation of sorts, for the hierarchy of $\mathcal{P}^\infty$ (it is the set of natural numbers with the operations max and succ, see Example 3.4.3). Furthermore, with universe polymorphism, universe levels are open terms, hence, convertibility between universe levels is now an issue. Fortunately, it is the last one, since once this issue is overcome, the encoding has one of the expected properties: we type check at least as much terms as in the original system.

To state this, we start with two useful lemmas:

Lemma 9.2.4 (Substitution and conversion)**.**

a. If x *is a free variable in* t *such that* t *and* $t\,[u/_x]$ *are well-typed,* $\|t\,[u/_x]\| = \|t\|\left[\|u\|/_x\right]$;

b. If ℓ *is a level variable in* t *such that* t *and* $t\,[u/_\ell]$ *are well-typed,* $\|t\,[u/_x]\| = \|t\|\left[|u|_L/_x\right]$;

c. If $t \rightsquigarrow_\beta u$*, then* $\|t\| \rightsquigarrow_\beta \|u\|$.

Proof. a and b are proved by induction on the the term t. c is because a β-redex is translated as a β-redex. □

The proof of this property is only sketched. In Chapter 8, we provided detailed proofs of analogous lemmas, which were much more involved than those ones, because of the type annotations added via the symbol ηE.

Lemma 9.2.5 (Shape-preservation of type)**.**

a. If s *is a sort,* $\mathtt{Lift}\,|\mathcal{A}(s)|_S\,\|s\| \rightsquigarrow^* \mathtt{Univ}\,|s|_S$,

b. If $(x : A) \to B$ *is of sort* s, $\mathtt{Lift}\,|s|_S\,\|(x : A) \to B\| \rightsquigarrow^* (x : \mathtt{Lift}\,|s_A|_S\,\|A\|) \Rightarrow \mathtt{Lift}\,|s_B|_S\,\|B\|$;

c. $\mathtt{Lift\ sortOmega}\,\|\forall\,[\ell_i]_i\,, A\| \rightsquigarrow^* (\ell_1 : \mathbb{L}) \Rightarrow \ldots \Rightarrow (\ell_n : \mathbb{L}) \Rightarrow \|A\|$.

Here again, s_A *and* s_B *designate the sorts of* A *and* B.

Proof. The three rules (the two ones in Section 5.3 and the one in Definition 9.2.1) on `Lift` are crafted to ensure those properties. □

To state properly the Correctness Theorem, one first has to define the translation of contexts:

Definition 9.2.6 (Context Translation)**.** *If* $\Sigma = x_1 : T_1, \dots, x_k : T_k$, $\Theta = i_1, \dots, i_m$ *and* $\Gamma = y_1 : A_1, \dots, y_n : A_n$, *then their translation is* $\|\Sigma; \Theta; \Gamma\| = x_1 : \texttt{Lift}\ \ |s_{T_1}|_S\ \|T_1\|, \dots, x_k : \texttt{Lift}\ \ |s_{T_k}|_S\ \|T_k\|, i_1 : \mathbb{L}, \dots, i_m : \mathbb{L}, y_1 : \texttt{Lift}\ |s_{A_1}|_S\ \|A_1\|, \dots, y_n : \texttt{Lift}\ |s_{A_n}|_S\ \|A_n\|$;
where the s_{T_i}*'s are the sorts of the* T_i*'s.*

If T_i is a universal quantification on universe level, $|s_{T_i}|_S = \texttt{sortOmega}$.

9.3 Soundness of the Encoding

Lemma 9.3.1 (Soundness of the encoding of levels)**.** *We assume that for all* $\ell \in \mathbb{L}$, $\vdash_{\lambda\Pi} |\ell|_L : \mathbb{L}$. *If* $\Theta \vdash_{\mathcal{UP}} \gamma$ isLvl, *then* $\|[]; \Theta; []\| \vdash_{\lambda\Pi} |\gamma|_L : \mathbb{L}$.

Proof. By induction of the derivation:

(lvl) This is the hypothesis.

($\mathbb{L}$var) Since $\|[]; [i_1, \dots, i_n]; []\|$ is of the shape $i_1 : \mathbb{L}, \dots, i_n : \mathbb{L}$, the rule (*var*) of $\lambda\Pi$-modulo rewriting can be applied to obtain this conclusion.

($\mathbb{L}\mathcal{A}$) and ($\mathbb{L}\mathcal{R}$) By induction hypothesis, we know that the arguments are of type $\mathbb{L}$, and the types of the functions `ax_r` and `ru_rr'` ensure that they output a term of type level.

□

Theorem 9.3.2 (Soundness)**.** *Given a correct criterion for equality of levels (i.e. if* $\ell_1 = \ell_2$, *then* $|\ell_1|_L \downarrow |\ell_2|_L$*), for a Universe-Polymorphic Full Pure Type System* P, *if* $\Sigma; \Theta; \Gamma \vdash_{\mathcal{UP}} t : A$, *then* $\|\Sigma; \Theta; \Gamma\| \vdash_{\lambda\Pi} \|t\| : \texttt{Lift}\ |s|_S\ \|A\|$, *where* s *is the sort of* A.

Proof. By induction on the derivation. The Lemma 9.3.1 treats the rules for which the conclusion is of the shape $\Theta \vdash_{\mathcal{UP}} \gamma$ isLvl. We then consider the 10 remaining cases:

(var) By induction hypothesis, $\|\Sigma; \Theta; \Gamma\| \vdash_{\lambda\Pi} \|A\| : \texttt{Univ}\ |r_\gamma|_S$. Hence $\|\Sigma; \Theta; \Gamma\| \vdash_{\lambda\Pi} \texttt{Lift}\ |r_\gamma|_S\ \|A\| : \star$, so one can introduce a variable of this type.

(ax) The translation of r_γ is $\texttt{code}\ (\texttt{r}\ |\gamma|_L)$ which lives in $\texttt{Univ}\ (\texttt{r'}\ (\texttt{ax\_r}\ |\gamma|_L))$, which is the reduct of the translation as a type of $r'_{\mathcal{A}_r(\gamma)}$.

(abs) By induction hypothesis, $\|\Sigma; \Theta; \Gamma\|, \texttt{x} : \texttt{Lift}\ |s_A|_S\ \|A\| \vdash_{\lambda\Pi} \|t\| : \texttt{Lift}\ |s_B|_S\ \|B\|$, hence, one has that $\|\Sigma; \Theta; \Gamma\| \vdash_{\lambda\Pi} \lambda(x : \texttt{Lift}\ |s_A|_S\ \|A\|).t : (x : \texttt{Lift}\ |s_A|_S\ \|A\|) \Rightarrow \texttt{Lift}\ |s_B|_S\ \|B\|$, which is the reduct of the translation as a type of $(x : A) \to B$. The other induction hypothesis, $\|\Sigma; \Theta; \Gamma\| \vdash_{\lambda\Pi} \|(x : A) \to B\| : \texttt{Univ}\ |r_\gamma|_S$, ensures us that $\texttt{Lift}\ |r|_S\ \|(x : A) \to B\|$ lives in $\star$, allowing us to use the (*conv*) rule.

(app) By induction hypothesis and Lemma 9.2.5, one can use the rule (*app*) of $\lambda\Pi$-modulo rewriting on the translation of t and the translation of u. The result lives in the translation of $B\,[u/_x]$ thanks to Lemma 9.2.4 (a).

(conv) This is a direct consequence of Lemma 9.2.4 (c) and the induction hypotheses.

(sig) By induction hypothesis, $\|\Sigma; \Theta; []\| \vdash_{\lambda\Pi} \|A\| : \texttt{Univ}\ |r_\gamma|_S$. Hence, one can use the (prod) rule of $\lambda\Pi$-modulo rewriting to move all the $i : \mathbb{L}$ from the context to the term. By Lemma 9.2.5, the product obtained is convertible with $\|\forall\Theta.A\|$, hence one can introduce a variable of this type. One must then use the weakening rule to reintroduce the variables of type $\mathbb{L}$ corresponding to the Θ'.

(inst) Lemma 9.2.5 tells us that, after conversion, the induction hypothesis is $\|\Sigma;\Theta;\Gamma\| \vdash_{\text{UP}} \|A\| : (\ell_1 : \mathbb{L}) \Rightarrow \ldots \Rightarrow (\ell_n : \mathbb{L}) \Rightarrow \|X\|$, hence, we can apply $\|A\|$ to the γ_i's without typing issues.

(prod) By induction hypothesis, we have $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi} \|A\| : \texttt{Univ}\ \|r_\gamma\|$ and also $\|\Sigma;\Theta;\Gamma, x : A\| \vdash_{\lambda\Pi} \|B\| : \texttt{Univ}\ \left\|r'_{\gamma'}\right\|$, so $\|\Sigma;\Theta;\Gamma\|, x : \texttt{Lift}\ |r_\gamma|_S\ \|A\| \vdash_{\lambda\Pi} \|B\| : \texttt{Univ}\ \left\|r'_{\gamma'}\right\|$ and we can conclude by introducing the lambda and applying the constant `prod`.

(ctx-weak) As before, we have $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi} \|A\| : \texttt{Univ}\ \|r_\gamma\|$, so $\|\Sigma;\Theta;\Gamma\| \vdash_{\lambda\Pi} \texttt{Lift}\ |r_\gamma|_S\ \|A\| : \star$, and one can weaken on a variable of this type.

(∀weak) Like for the (sig) rule, one can empty the context of the variables of type $\mathbb{L}$ by applying the rule (prod) of $\lambda\Pi$-modulo rewriting. Then, one can weaken on a variable of this type and variables of type $\mathbb{L}$ to translate the Θ'. □

9.4 Instantiating the Encoding

Now, we will more specifically focus on a specific hierarchy of levels, where $\mathbb{L} = \mathbb{N}$, all the $\mathcal{A}_r$ are the successor function and all $\mathcal{R}_{rr'}$ are the maximum function. This is the predicative hierarchy of $\mathcal{P}^\infty$ (Example 3.4.3), used in Agda for instance.

The grammar of universe levels we are interested in is: $t, u \in \mathcal{L} ::= x \in \mathcal{V} \mid \texttt{0}_\mathbb{L} \mid \texttt{s}_\mathbb{L}\, t \mid \texttt{max}_\mathbb{L}\, t\, u$:

```
constant 𝕃 : TYPE.                    symbol 0𝕃 : 𝕃.
symbol s𝕃 : 𝕃 ⇒ 𝕃.                    symbol max𝕃 : 𝕃 ⇒ 𝕃 ⇒ 𝕃.
```

The question which arises in the translation is to have a convergent rewriting system such that for all t and u in $\mathcal{L}$:

$$t\Downarrow = u\Downarrow \text{ if and only if } \forall \sigma : \mathcal{V} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$$

where $[\![\_]\!]_\_ : \mathcal{L} \to (\mathcal{V} \to \mathbb{N}) \to \mathbb{N}$ is the obvious interpretation in $\mathbb{N}$:

$$[\![\texttt{0}_\mathbb{L}]\!]_\sigma = 0 \quad [\![x]\!]_\sigma = \sigma(x), \text{ if } x \in \mathcal{V} \quad [\![\texttt{s}_\mathbb{L}\, t]\!]_\sigma = [\![t]\!]_\sigma + 1 \quad [\![\texttt{max}_\mathbb{L}\, t\, u]\!]_\sigma = \max([\![t]\!]_\sigma, [\![u]\!]_\sigma)$$

Since max is associative and commutative (AC), we will propose an encoding having a weak version of this property: $t\Downarrow \equiv_{AC} u\Downarrow$ if and only if $\forall \sigma : \mathcal{V} \to \mathbb{N}, [\![t]\!]_\sigma = [\![u]\!]_\sigma$.

Since $[\![\texttt{s}_\mathbb{L}\, (\texttt{max}_\mathbb{L}\, t\, u)]\!] = [\![\texttt{max}_\mathbb{L}\, (\texttt{s}_\mathbb{L}\, t)\, (\texttt{s}_\mathbb{L}\, u)]\!]$, one can consider having a `Max` acting on a set of terms, which do not contain $\texttt{max}_\mathbb{L}$.

Furthermore, we have for all n the equality $[\![\texttt{max}_\mathbb{L}\, (\texttt{s}^n_\mathbb{L}\, x)\, x]\!] = [\![\texttt{s}^n_\mathbb{L}\, x]\!]$. To avoid declaring this rule infinitely often (once for every n), we add addition to our encoding. However, since we introduce addition only to encode iteration of the application of $\texttt{s}_\mathbb{L}$, we do not define this addition between two levels, but only between a ground natural number and a level. Furthermore, $[\![\texttt{max}_\mathbb{L}\, (\texttt{s}^n_\mathbb{L}\, x)\, (\texttt{s}^m_\mathbb{L}\, 0)]\!] = [\![\texttt{s}^n_\mathbb{L}\, x]\!]$, if $m \leqslant n$. Hence, the symbol `Max` will also collect the value of the smallest possible ground natural number that the result can be.

Hence, in our encoding, the normal forms are the `Max` $i\ \{j_k + x_k\}_{k \leqslant n}$ where:
(1) $x_1, \ldots, x_n$ are distinct variables,
(2) for all $k \leqslant n$, $i \geqslant j_k$.
(3) $i, j_1, \ldots, j_n$ are ground natural numbers,
A separate type $\mathbb{N}$, containing only ground natural numbers, is declared, to avoid confusion with levels.

```
constant ℕ : TYPE.        constant 0ℕ : ℕ.      constant sℕ : ℕ ⇒ ℕ.
definition 1ℕ := sℕ 0ℕ.

symbol maxℕ : ℕ ⇒ ℕ ⇒ ℕ.
[x]   maxℕ x      0ℕ      ↪ x.
[y]   maxℕ 0ℕ     y       ↪ y.
[x,y] maxℕ (sℕ x) (sℕ y) ↪ sℕ (maxℕ x y).

infix +ℕ : ℕ ⇒ ℕ ⇒ ℕ.
[y]   0ℕ      +ℕ y ↪ y.
[x,y] (sℕ x) +ℕ y ↪ sℕ (x +ℕ y).
```

Sets can be empty, singleton, or union of sets. This union operator is an associative and commutative symbol. Furthermore, since singletons are of the form $\{i + x\}$, the constructor of singletons is denoted $\oplus$.

```
symbol ∅ : LSet.
infix ⊕ : ℕ ⇒ 𝕃 ⇒ LSet.
infix ac ∪ : LSet ⇒ LSet ⇒ LSet.
[x] x ∪ ∅ ↪ x.
```

Since constraint (3) is guaranteed by typing, it remains to implement the two constraints (1) and (2) presented in the description of the normal form above:

- The only non-left-linear rule of the encoding eliminates redundancies, ensuring that all variables in the normal forms are distinct, in order to satisfy the invariant (1).

  ```
  [i,j,l] (i ⊕ l) ∪ (j ⊕ l) ↪ (maxℕ i j) ⊕ l.
  ```

- Intuitively, to flatten the entanglement of max and plus, we would like to have a rule stating that $a + \max(b, c) = \max(a + b, a + c)$.

  However, to fulfill constraint (2), we added the invariant that the first argument of `Max` is larger than all the first arguments of the $\oplus$ occurring directly under it. Hence, we do not declare the expected computation rule of $\oplus$, but enforce this computation to be performed under a `Max`.

  Furthermore, to enforce typing distinction between $\mathbb{L}$ and `LSet`, we introduce an auxiliary function, `mapPlus`, mapping `(i ⊕ _)` to all the elements of a set.

  ```
  symbol mapPlus : ℕ ⇒ LSet ⇒ LSet.
  [i]       mapPlus i ∅          ↪ ∅.
  [i,j,l]   mapPlus i (j ⊕ l)    ↪ (i +ℕ j) ⊕ l.
  [i,l1,l2] mapPlus i (l1 ∪ l2)  ↪ (mapPlus i l1) ∪ (mapPlus i l2).

  symbol Max : ℕ ⇒ LSet ⇒ 𝕃.
  [x]           Max 0ℕ   (0ℕ ⊕ x)             ↪ x.
  [i,j,k,l]     Max i    (j ⊕ Max k l)        ↪
                                  Max (maxℕ i (j +ℕ k)) (mapPlus j l).
  [i,j,k,l,tl] Max i  ((j ⊕ Max k l) ∪ tl) ↪
                             Max (maxℕ i (j +ℕ k)) ((mapPlus j l) ∪ tl).
  ```

And finally we give rewriting rules for the symbols of the syntax:

```
[]    0𝕃       ↪ Max 0ℕ ∅.
[x]   s𝕃 x     ↪ Max 1ℕ (1ℕ ⊕ x).
[x,y] max𝕃 x y ↪ Max 0ℕ ((0ℕ ⊕ x) ∪ (0ℕ ⊕ y)).
```

This encoding is not confluent, as the following example illustrates:

```
Max i (j ⊕ (Max k (j2 ⊕ (Max k2 l))))
 ⇝out Max (maxℕ i (j +ℕ k)) (mapPlus j (j2 ⊕ (Max k2 l)))
  ⇝ Max (maxℕ i (j +ℕ k)) ((j +ℕ j2) ⊕ (Max k2 l))
  ⇝ Max (maxℕ (maxℕ i (j +ℕ k)) (j +ℕ j2 +ℕ k2)) (mapPlus (j +ℕ j2) l)
 ⇝in Max i (j ⊕ (Max (maxℕ k (j2 +ℕ k2)) (mapPlus j2 l)))
  ⇝ Max (maxℕ i (j +ℕ (maxℕ k (j2 +ℕ k2)))) (mapPlus j (mapPlus j2 l))
```

But this is not an issue, since we are only interested in reducts of elements of the syntax, meaning that all the variables are of type $\mathbb{L}$.

Proposition 9.4.1. *The absence of variable of type* $\mathbb{N}$ *or* `LvlSet` *ensures that all normal forms are of the shape* `Max i` $((j_1 \oplus x_1) \cup \ldots \cup (j_n \oplus x_n))$ *where:*
(1) $x_1, \ldots, x_n$ *are distinct variables,*
(2) for all $k \leqslant n$, $i \geqslant j_k$.

Proof. Since there are no variables of type $\mathbb{N}$ and `LSet`, the function $\mathtt{max}_{\mathbb{N}}$, $+_{\mathbb{N}}$ and `mapPlus` are totally defined and cannot occur in the normal forms.

Hence, normal forms contain only $0_{\mathbb{N}}$, $\mathtt{s}_{\mathbb{N}}$, `Max`, $\emptyset$, $\oplus$ and $\cup$. Among those symbols, the only constructor of a $\mathbb{L}$ is `Max`, hence every level is either a variable or headed by `Max`.

If a normal form of type $\mathbb{L}$ contains a `Max`, there is one at the head. Hence the normal forms are of the form `Max n s` with `n` a closed natural number and `s` a `LSet`. If there are more than one `Max`, it means that the `LSet s` contains a level which is not a variable. This one is headed by `Max`, so one of the rewriting rules regarding the interaction between `Max` and $\oplus$ can be applied.

Hence all normal forms are either a variable or of the form `Max n s`, with `n` closed natural and `s` a `LSet` where all levels are variable. The non-linear rule ensures us that the variables are all distinct.

One can check that the invariant that every natural which is the first argument of a $\oplus$ is smaller or equal to the first argument of the `Max` directly above the $\oplus$ is preserved by every rule and verified by the reducts of the syntax.

So, we can conclude that the normal forms have the shape announced:
`Max i` $((j_1 \oplus x_1) \cup \ldots \cup (j_n \oplus x_n))$ where:
(1) $x_1, \ldots, x_n$ are distinct variables,
(2) for all $k \leqslant n$, $i \geqslant j_k$. □

To check that a term cannot have two distinct normal forms, the definition of the interpretation is extended to the symbols we introduced and one can verify that all the rules preserve the interpretation and that all the terms of the shape we described have a different interpretation.

Let $\sigma : \mathcal{V} \to \mathbb{N}$, we will define $[\![\_]\!]^{\mathbb{L}}_{\sigma}$, $[\![\_]\!]^{\mathbb{N}}$ and $[\![\_]\!]^{S}_{\sigma}$ for all terms of type $\mathbb{L}$, $\mathbb{N}$ and `LSet` respectively.

Regarding the symbols constructing terms of type $\mathbb{N}$, we will define $[\![\_]\!]^{\mathbb{N}}_{\sigma} : \Lambda \to \mathbb{N}$ straightforwardly by:

$$[\![0_{\mathbb{N}}]\!]^{\mathbb{N}} = 0 \qquad [\![1_{\mathbb{N}}]\!]^{\mathbb{N}} = 1$$
$$[\![\mathtt{s}_{\mathbb{N}}\, t]\!]^{\mathbb{N}} = [\![t]\!]^{\mathbb{N}}_{\sigma} + 1 \qquad [\![\mathtt{max}_{\mathbb{N}}\, t\, u]\!]^{\mathbb{N}} = \max([\![t]\!]^{\mathbb{N}}, [\![u]\!]^{\mathbb{N}})$$
$$[\![t +_{\mathbb{N}} u]\!]^{\mathbb{N}} = [\![t]\!]^{\mathbb{N}} + [\![u]\!]^{\mathbb{N}}$$

The rewriting rules defining $\mathtt{max}_{\mathbb{N}}$ and $+_{\mathbb{N}}$ preserves the interpretation, since 0 is neutral for the max operator on $\mathbb{N}$.

The interpretation of natural numbers does not require an interpretation of variable σ, since all the terms of type $\mathbb{N}$ are ground.

For terms of type `LSet`, one could expect to interpret them as set of natural numbers. But the symbol $\cup$ is misleading, since it cannot be interpreted by the union of sets, because of the rule:

```
[i,j,l] (i ⊕ l) ∪ (j ⊕ l) ↪ (max_ℕ i j) ⊕ l.
```

Hence, we will interpret the terms of type `LSet` by natural numbers. $[\![\_]\!]^S_\sigma : \Lambda \to (\mathbb{N} \cup \{-\infty\})$ is:

$$[\![\emptyset]\!]_\sigma = -\infty \qquad [\![t \oplus u]\!]_\sigma = [\![t]\!]^{\mathbb{N}}_\sigma + [\![u]\!]^{\mathbb{L}}_\sigma$$
$$[\![t \cup u]\!]_\sigma = \max([\![t]\!]^S_\sigma, [\![u]\!]^S_\sigma) \qquad [\![\texttt{mapPlus}\, t\, u]\!]_\sigma = [\![t]\!]^{\mathbb{N}}_\sigma + [\![u]\!]^S_\sigma$$

And the interpretation $[\![\_]\!]^{\mathbb{L}}_\sigma$ is as expected:

$$[\![\texttt{Max}\, t\, u]\!]^{\mathbb{L}}_\sigma = \max([\![t]\!]^{\mathbb{N}}_\sigma, [\![u]\!]^S_\sigma)$$
$$[\![x]\!]^{\mathbb{L}}_\sigma = \sigma(x) \text{ if } x \text{ is a variable}$$

Once again, it is straightforward to check that all the rewriting rules preserve the interpretation.

Proposition 9.4.2. *The absence of variable of type $\mathbb{N}$ or `LvlSet` ensures the uniqueness of normal forms (modulo AC) of all terms of type $\mathbb{L}$.*

Proof. For this, one just has to show that all the terms of the shape we described have a different interpretation. Let

$$t = \texttt{Max i}((j_1 \oplus x_1) \cup \ldots \cup (j_n \oplus x_n) \cup (l_1 \oplus y_1) \cup \ldots \cup (l_m \oplus y_m))$$
$$u = \texttt{Max a}((b_1 \oplus x_1) \cup \ldots \cup (b_n \oplus x_n) \cup (c_1 \oplus z_1) \cup \ldots \cup (c_p \oplus z_p))$$

be two normal forms of type $\mathbb{L}$. So:

(1) $x_1, \ldots, x_n, y_1, \ldots, y_m, z_1, \ldots, z_p$ are distinct variables,

(2) for all k, $i \geqslant j_k$, $i \geqslant l_k$, $a \geqslant b_k$ and $a \geqslant c_k$, .

For terms of type $\mathbb{N}$, we identify the term and its interpretation as a natural number, o we omit the $[\![\_]\!]^{\mathbb{N}}$.

Let $\sigma : \mathcal{V} \to \mathbb{N}$, we have $[\![t]\!]^{\mathbb{L}}_\sigma = \max(i, j_1 + \sigma(x_1), \ldots, j_n + \sigma(x_n), l_1 + \sigma(y_1), \ldots, l_m + \sigma(y_m))$ and $[\![u]\!]^{\mathbb{L}}_\sigma = \max(a, b_1 + \sigma(x_1), \ldots, b_n + \sigma(x_n), c_1 + \sigma(z_1), \ldots, c_p + \sigma(z_p))$.

We assume that for all σ, $[\![t]\!]^{\mathbb{L}}_\sigma = [\![u]\!]^{\mathbb{L}}_\sigma$.

Especially, for $\sigma : x \in \mathcal{V} \mapsto 0$, we have $\max(i, j_1, \ldots, j_n, l_1, \ldots, l_m) = \max(a, b_1, \ldots, b_n, c_1, \ldots, c_p)$. Hence by condition (2), $i = a$.

Let $M = \max(i, j_1, \ldots, j_n, l_1, \ldots, l_m, a, b_1, \ldots, b_n, c_1, \ldots, c_p)$ be the max of all the natural numbers involved in the normal forms t and u.

For $k \in [\![1, n]\!]$, let σ_k be defined by $\sigma_k(x_k) = M$ and $\sigma_k(z) = 0$ if $z \neq x_k$, we have that $[\![t]\!]^{\mathbb{L}}_{\sigma_k} = j_k + M$ and $[\![u]\!]^{\mathbb{L}}_{\sigma_k} = b_k + M$, so $j_k = b_k$.

If $m \geqslant 1$, we define τ by $\tau(y_1) = M + 1$ and $\tau(z) = 0$ if $z \neq y_1$, we have $[\![t]\!]^{\mathbb{L}}_\tau = l_1 + M + 1$ and $[\![u]\!]^{\mathbb{L}}_\tau = a$, hence it is not possible to have $[\![t]\!]^{\mathbb{L}}_\tau = [\![u]\!]^{\mathbb{L}}_\tau$ if $m \geqslant 1$. Hence, $m = 0$ and all the variables which occur in t also occur in u. By symmetry, $p = 0$.

Hence, if two normal forms of type $\mathbb{L}$ are distinct modulo AC, there is an interpretation of variable σ able of distinguishing between them. □

Chapter 10

Agda2Dedukti: A Translator of Agda Programs to Dedukti

Agda [NAD+05, Nor07] is a dependently-typed programming language, based on an extension of Martin-Löf type theory, Luo's Unifying Theory of dependent Types [Luo94, Chapter 9], which allows the direct writing of programs (which can be seen as proofs) using the ability to declare function by dependent pattern matching. Agda features both universe polymorphism and η-conversion.

Since it features both ingredients introduced in Chapters 9 and 8, developing a prototypical translator [CG19] from Agda to Dedukti was a natural goal.

However, Agda offers its users a logic much richer than a universe polymorphic pure type system with η-conversion. First of all, Agda permits to declare inductive types and then to define functions using dependent pattern-matching on the constructors of this type. This behaviour can easily be replicated in Dedukti, by declaring new symbols for inductive types, constructors and functions and rewriting rules for each case of the dependent pattern matching. Just like sorts and products have an encoded and a decoded version, linked by the application of the function `Term`, the type has two translation, one as a code and one decoded, linked by a rewriting rule enriching the definition of `Term`. Analogously, one rewriting rule is added to enrich the definition of ηE.

Example 10.0.1. *The* Agda *declaration of the addition of natural numbers:*

```
data Nat : Set where                    _+_ : Nat → Nat → Nat
  zero : Nat                            zero  + m = m
  suc  : (n : Nat) → Nat                suc n + m = suc (n + m)
```

is translated in Dedukti *by:*

```
constant TYPE__Nat : TYPE.              constant Nat : Univ (set 0).
[] Lift _ Nat ↪ TYPE__Nat.
[t] ηE _ Nat t ↪ t.

constant Nat__zero: Lift (set 0) Nat.
constant Nat__suc: Lift (set 0) (prod (set 0) (set 0) Nat (λ n, Nat)).

symbol {|_+_|} : Lift (set 0) (prod (set 0) (set 0) Nat
                  (λ _0, prod (set 0) (set 0) Nat (λ _1, Nat))).
[m]   {|_+_|} Nat__zero    m ↪ m.
```

```
[m,n] {|_+_|} (Nat__suc n) m ↪ Nat__suc ({|_+_|} n m).
```

We can observe that `Nat` *in* Agda *became* `TYPE__Nat` *and* `Nat` *in* Dedukti*, and two rules have been added: one to state that* `TYPE__Nat` *is the decoding of* `Nat` *and the other to extend the definition of* ηE*.*

Each declaration of a new type consists in adding a new constructor to the type `Univ s`. The new rules on ηE and `Term` are here to ensure that the pattern-matching on this type remains exhaustive, in order to completely get rid of administrative encoding operators in the normal forms of values.

One can note that the enrichment of the functions `Term` and ηE are left to the will of the author of the translation. This proves to be a good feature, since the η-conversion of Agda does not restrict to product types, but also concerns records (η-conversion of records is also sometimes called "surjective pairing" and means that if t lives in $\sum_{x:A} B$, then t and $(\mathrm{fst}\, t, \mathrm{snd}\, t)$ are convertible). This does not require to introduce a new symbol for this enrichment of the conversion, but just to define adequate rules on ηE.

Example 10.0.2. *The declaration of this record:*

```
record r : Set1 where                    constructor cons
  field A : Set                          field b : A
```

is translated by:

```
constant TYPE__r : ⋆.
constant r : Univ (set (s 0)).

constant r__cons : Lift (set (s 0)) (prod (set (s 0)) (set (s 0))
          (code (set 0)) (λ A, prod (set 0) (set (s 0)) A (λ b, r))).
symbol r__A : Lift (set (s 0))
              (prod (set (s 0)) (set (s 0)) r (λ r, code (set 0))).
symbol r__b : Lift (set (s 0))
              (prod (set (s 0)) (set 0) r (λ r, r__A r)).

[] Lift _ r ↪ TYPE__r.
[y] ηE _ r y ↪ r__cons (r__A y) (ηE 0 (r__A y) (r__b y)).

[A,b] r__A (r__cons A b) ↪ A.
[A,b] r__b (r__cons A b) ↪ ηE 0 A b.
```

The rule to define the η*-expansion of an element of* `r` *states that if* y *is of type* `r`*, then* $y \equiv \{a = y.a; b = y.b\}$*.*

This prototypical translator is available at `https://github.com/Deducteam/Agda2Dedukti`, the directory `theory/` contains the encoding presented in Chapters 8 and 9. It is able to translate and type-check 162 files of Agda's standard library [DDA20], out of 590.

10.1 Future Work

We presented in those last three chapters a correct encoding of universe polymorphism in $\lambda\Pi$-modulo rewriting, meaning that every term typable in the original system is translated to a typable term. We also presented a rewriting system to decide equality in the max-plus algebra, which is a common universe algebra (and especially is the one used by Agda).

Furthermore, we proposed an operator ηE to encode shallowly a type-directed rule, like η-conversion, since the translation of an application really involves the application of the translation of a term to the other one, reducing the interleaving between the computation steps coming from the original system and the steps related to the encoding.

Finally, we applied those results to the practical case of the translation of the proof system Agda, which offers, among others, the features we targeted, allowing us to provide Dedukti users with more than 500 declarations of types, constructors or functions, originating from Agda's standard library.

We proved that translations of well-typed terms remain typable in our encoding. However, it could be that our encoding is over-permissive and type-checks much more terms than the original system. Hence, one could envision a conservativity theorem, stating that if the translation of a type is inhabited, then the type is also inhabited in the original system. For implementability purposes, we have chosen an encoding with finitely many symbols. Such a theorem has only been proved [CD07, Ass15], for encodings of PTS with as many symbols as sorts, axioms and rules. Extending those theorems to our setting is a short-term goal.

Regarding the implementation, making the translator more complete is naturally an objective, however, it involves more theoretical problems, which are long run research programs. For instance, how size types or co-inductive types can be encoded in the $\lambda\Pi$-calculus modulo rewriting is not known yet.

Now that proofs have been translated to the logical framework Dedukti, they can be analyzed, and (when it is possible) exported to other proof assistants, like what was done with proofs originating from the arithmetic library of Matita [Thi18].

Nomenclature

$\overset{\varepsilon}{\rightsquigarrow}_\beta$	Head β-reduction
$\rightsquigarrow_\beta$	β-reduction
$\leftrightsquigarrow^*_\beta$	Reflexive symmetric transitive closure of $\rightsquigarrow_\beta$
$\Rightarrow$	Symbol of functional types
$\equiv_\alpha$	Relation of α-equivalence
$[\![m, n[\![$	$\{k \in \mathbb{N} \mid m \leqslant k < n\}$
$\leftrightsquigarrow^*$	Convertibility relation
$\downarrow$	Joinability relation
$\preccurlyeq$	Well-founded pre-order on type symbols
$\hookrightarrow$	Pairing operator of rewriting rules
$\mathcal{A}$	Axioms of a PTS
$\mathrm{Acc}(f)$	Set of accessible position under function f.
$\mathrm{ar}(f)$	Arity of the symbol f
$\mathcal{C}^\infty$	PTS of specification $\mathcal{S} = \{*_i \mid i \in \mathbb{N}\}; \mathcal{A} = \{(*_i, *_{i+1})\}; \mathcal{R} = \{(*_i, *_j, *_k) \mid j \geqslant 1 \text{ and } k = \max(i, j)\} \cup \{(*_i, *_0, *_0)\}$
DB	Set of terms with De Bruijn indices
Δ	Lambda term $\lambda\,(x : A)\,.x\,x$
dom	Domain of a context
dom	Domain of a substitution
ε	The empty word
$\mathcal{F}$	Set of functions symbols
$f[A]$	If A is a subset of the definition set of f, $f[A] = \{f(x) \mid x \in A\}$
$f\vert_A$	Function f restricted to the set A
$\mathcal{F}_o$	Objects of the signature

$\mathrm{FrozTyp}_{\preccurlyeq C}$	Frozen types with respect to the type constructor C
$\mathcal{F}_{\mathcal{T}}$	Type families of the signature
FV	Function extracting the set of free variables
$\mathcal{I}_C$	Pre-interpretation of the type value $C\,\vec{t}$
id	Identity function
Λ	Set of terms
λ	Binders for functional terms
$\lambda^{\rightarrow}$	PTS of specification $(\mathcal{S} = \{\star, \Box\}\,; \mathcal{A} = \{\star : \Box\}\,; \mathcal{R} = \{(\star, \star, \star)\})$
$\lambda\Pi$	PTS of specification $(\mathcal{S} = \{\star, \Box\}\,; \mathcal{A} = \{\star : \Box\}\,; \mathcal{R} = \{(\star, \star, \star), (\star, \Box, \Box)\})$
$\mathcal{M}$	Set of meta-variable names
MV	Function extracting the set of meta variables
NF	Set of terms in normal form
$\mathcal{N}_o$	Set of neutral objects
$\mathcal{N}_{\mathcal{T}}$	Set of neutral type families
$\mathcal{P}$	Powerset function
$\mathcal{P}_F$	Function associating to a set the set of its finite subsets
π_k	k^{th} projection of a tuple
$\mathcal{P}^{\infty}$	PTS of specification $\mathcal{S} = \{*_i \mid i \in \mathbb{N}\}\,; \mathcal{A} = \{(*_i, *_{i+1})\}\,; \mathcal{R} = \{(*_i, *_j, *_k) \mid k = \max(i, j)\}$
$\mathfrak{R}$	Set of rewriting rules
$\mathcal{R}$	Rules of a PTS
$\mathcal{S}$	Set of sort names of a PTS
size	Function outputting the size of terms
SN	Set of strongly normalizing terms
T	Set of abstract terms
$t\,\sigma$	Application of the function induced by the substitution σ to the term t
Θ	Typing map. The function giving the type of the signature symbols
$\mathcal{V}$	Set of variables names
$\mathcal{V}al_T$	Set of type values
$\mathcal{X}$	Set of names

Bibliography

[AA02] Andreas Abel and Thorsten Altenkirch. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, 12(01), 2002.

[ABC+19] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory. 2019.

[Abe98] Andreas Abel. foetus – Termination Checker for Simple Functional Programs. Programming Lab Report, 1998.

[Acz77] Peter Aczel. An Introduction to Inductive Definitions. *Studies in logic and the foundations of mathematics*, 90:739–782, 1977.

[AG00] Thomas Arts and Jürgen Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[ALSU86] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 1986.

[AP16] Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26(e2), 2016.

[Ass15] Ali Assaf. *A Framework for Defining Computational Higher-Order Logics*. PhD thesis, École polytechnique, France, 2015.

[AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. Normalization by Evaluation for Sized Dependent Types. 2017.

[Bar81] Henk P. Barendregt. *The Lambda-Calculus, Its Syntax and Semantics*. Amsterdam, New York, and Oxford, 1981.

[Bar92] Henk P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[Bar99] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7 - Denis Diderot, France, 1999.

[Bar07] Bruno Barras. Coq-Contribs/PTS. `https://github.com/coq-contribs/pts`, 2007.

[BCH12] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus Modulo as a Universal Proof Language. *PxTP*, page 16, 2012.

[BGH19] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting. In *4th International Conference on Formal Structures for Computation and Deduction*, pages 9:1–9:21, Dortmund, Germany, 2019.

[BGR08] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In *22nd International Conference on Computer Science Logic*, volume 5213 of *Lecture Notes in Computer Science*, 2008.

[BHS01] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. An Induction Principle for Pure Type Systems. *Theoretical Computer Science*, 266(1-2):773–818, 2001.

[BJO99] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The Calculus of Algebraic Constructions. *10th International Conference on Rewriting Techniques and Applications (RTA)*, LNCS 1631, 1999.

[BJR08] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The Computability Path Ordering: The End of a Quest. In Michael Kaminski and Simone Martini, editors, *Computer Science Logic*, volume 5213, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Bla01] Frédéric Blanqui. *Type Theory and Rewriting.* Phd, Université Paris XI, Orsay, France, 2001.

[Bla05] Frédéric Blanqui. Definitions by Rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[Bla06] Frédéric Blanqui. Higher-Order Dependency Pairs. In *Eighth International Workshop on Termination - WST 2006*, Seattle, United States, August 2006.

[Bla18] Frédéric Blanqui. Size-based termination of higher-order rewriting. *Journal of Functional Programming*, 28(e11), 2018. 75 pages.

[Bla20] Frédéric Blanqui. Type Safety of Rewrite Rules in Dependent Types. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 167 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Bou12] Pierre Boutillier. A relaxation of Coq's guard condition. In *JFLA - Journées Francophones des langages applicatifs - 2012*, pages 1 – 14, Carnac, France, 2012.

[CD07] Denis Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007.

[CDT20] The Coq Development Team. Coq. `https://coq.inria.fr/`, 1984-2020.

[CG19] Jesper Cockx and Guillaume Genestier. Agda2dedukti. `https://github.com/Deducteam/Agda2Dedukti`, 2019.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report RR-0530, Inria, May 1986.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. *J. Symb. Log.*, 5(2):56–68, 1940.

[Coq86] Thierry Coquand. An Analysis of Girard's Paradox. Technical Report RR-0531, INRIA, May 1986.

[Coq92] Thierry Coquand. Pattern Matching with Dependent Types. In *Proceedings of the International Workshop on Types for Proofs and Programs*, 1992.

[CR36] Alonzo Church and John Barkley Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[Cur30] Haskell B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52(3):509–536, 1930.

[DDA20] Nils Anders Danielsson, Matthew Daggitt, and Guillaume Allais. Agda standard library. `https://github.com/agda/agda-stdlib`, 2010-2020.

[Ded20] Deducteam. Dedukti. `https://deducteam.github.io/`, 2011-2020.

[DHK03] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *J. Autom. Reasoning*, 31(1):33–72, 2003.

[DHW93] Gilles Dowek, Gérard Huet, and Benjamin Werner. On the Definition of the Eta-long Normal Form in Type Systems of the Cube. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 115–130, 1993.

[Fér20] Gaspard Férey. *Higher-Order Confluence and Universe Embedding in the Logical Framework*. PhD thesis, École Normale Supérieure Paris-Saclay, France, 2020.

[FK11] Carsten Fuhs and Cynthia Kop. Harnessing First Order Termination Provers Using Higher Order Dependency Pairs. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*. Springer, Berlin, Heidelberg, 2011.

[FK19] Carsten Fuhs and Cynthia Kop. A Static Higher-Order Dependency Pair Framework. In Luís Caires, editor, *Programming Languages and Systems*, volume 11423, pages 752–782. Springer International Publishing, 2019.

[Gen18] Guillaume Genestier. Sizechangetool. `https://github.com/Deducteam/SizeChangeTool`, 2018.

[Geu92] Herman Geuvers. The Church-Rosser Property for beta-eta-reduction in Typed lambda-Calculi. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92)*, pages 453–460. IEEE Computer Society, 1992.

[GH17] Herman Geuvers and Tonny Hurkens. Deriving Natural Deduction Rules from Truth Tables. In Sujata Ghosh and Sanjiva Prasad, editors, *Logic and Its Applications - 7th Indian Conference, ICLA*, volume 10119 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2017.

[Gim94] Eduardo Giménez. Codifying Guarded Definitions with Recursive Schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Paris VII, France, 1972.

[GLT88] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[GN91] Herman Geuvers and Mark-Jan Nederhof. Modular Proof of Strong Normalization for the Calculus of Constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[Göd58] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280 – 287, 1958.

[GS10] Benjamin Grégoire and Jorge Luis Sacchini. On Strong Normalization of the Calculus of Constructions with Type-Based Termination. In Christian G. Fermüller and Andrei Voronkov, editors, *17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 6397 of *Lecture Notes in Computer Science*, pages 333–347, 2010.

[HA28] David Hilbert and Wilhelm Ackermann. Grundzüge der theoretischen Logik. 1928.

[HAB+15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexei Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A Formal Proof of the Kepler Conjecture. *ArXiv*, abs/1501.02155, 2015.

[Hal05] Thomas Hales. A Proof of the Kepler Conjecture. *Annals of Mathematics*, 162:1065–1185, 2005.

[Ham19] Makoto Hamana. How to prove decidability of equational theories with second-order computation analyser SOL. *Journal of Functional Programming*, 29, 2019.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HP91] Robert Harper and Randy Pollack. Type Checking with Universes. *Theoretical Computer Science*, 89:107–136, 1991.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 01 1996.

[HR10] Pierre Hyvernat and Christophe Raffalli. Improvements on the "Size Change Termination Principle" in a Functional Language. In *11th International Workshop on Termination*, 2010.

[Hyv14] Pierre Hyvernat. The Size-Change Termination Principle for Constructor Based Languages. *Logical Methods in Computer Science*, 10(1), 2014.

[JL15] Jean-Pierre Jouannaud and Jian-Qi Li. Termination of Dependently Typed Rewrite Rules. In *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications*, Leibniz International Proceedings in Informatics 38, 2015.

[JO97] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.

[JR07] Jean-Pierre Jouannaud and ALbert Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 54(1):1–48, 2007.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Utrecht Universiteit, NL, 1980. Published as Mathematical Center Tract 129.

[Kop] Cynthia Kop. Wanda. `http://wandahot.sourceforge.net/`.

[Kop12] Cynthia Kop. *Higher Order Termination*. PhD thesis, VU University Amsterdam, 2012.

[Kop19] Cynthia Kop. Mail to the termtools list: higher-order union beta category in the tpdb, 19 March, 2019.

[Kri93] Jean-Louis Krivine. *Lambda-Calculus, Types and Models*. Ellis Horwood series in computers and their applications. Masson, 1993.

[KS07] Keiichirou Kusakari and Masahiko Sakai. Enhancing Dependency Pair Method using Strong Computability in Simply-Typed Term Rewriting Systems. *Applicable Algebra in Engineering Communication and Computing*, 18(5):407–431, 2007.

[KvOvR93] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory Reduction Systems: Introduction and Survey. *Theoretical Computer Science*, 121:279–308, 1993.

[KvR12] Cynthia Kop and Femke van Raamsdonk. Dynamic Dependency Pairs for Algebraic Functional Systems. *Logical Methods in Computer Science*, 8(2), 2012.

[Lep16] Rodolphe Lepigre. The PML_2 Language: Proving Programs in ML, 2016.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, 2001.

[Log] Logipedia. `http://www.logipedia.science`.

[Luo94] Zhaohui Luo. *Computation and Reasoning - a Type Theory for Computer Science*, volume 11 of *International series of monographs on computer science*. Oxford University Press, 1994.

[Men87] Paul F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, United States, 1987.

[Mil91] Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.*, 1(4):497–536, 1991.

[NAD+05] Ulf Norell, Andreas Abel, Niels Anders Danielsson, Makoto Takeyama, and Catarina Coquand. Agda. `https://github.com/agda/agda`, v1.0 : 1999, v2.0 : 2005.

[New42] Maxwell H. A. Newman. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics*, 43, Number 2:223–243, 1942.

[Nip91] Tobias Nipkow. Higher-Order Critical Pairs. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, 1991.

[Nor07] Ulf Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. Phd, Chalmers University of Technology, Gothenburg, Sweden, 2007.

[PPM90] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. technical report CMU-CS-89-209.

[Ram30] Frank P. Ramsey. On a Problem of Formal Logic. *Proc. London Math. Soc.*, 30:264–286, 1930.

[Sac11] Jorge Luis Sacchini. *On Type-Based Termination and Dependent Pattern Matching in the Calculus of Inductive Constructions*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2011.

[Sai15] Ronan Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, Mines ParisTech, France, 2015.

[Sch24] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Math. Ann.*, 92:305–316, 1924.

[Sel98] Jonathan P. Seldin. Excluded Middle without Definite Descriptions in the Theory of Constructions. *Proc. of the 1st Montreal Workshop on Programming Language Theory*, 1998.

[Sel08] Peter Selinger. Lecture Notes on the Lambda Calculus. *CoRR*, 2008.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 499–514. Springer, 2014.

[Tai67] William W. Tait. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic*, 32(02):198–212, 1967.

[Tar28] Alfred Tarski. Un théorème sur les fonction d'ensembles. *Annales de la société Polonaise de Mathématiques*, 6:133–134, 1928.

[TC] Termination Competition. `http://termination-portal.org/wiki/Termination_Competition`.

[TG05] René Thiemann and Jürgen Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering Communication and Computing*, 16(4):229–270, 2005.

[Thi07] Rene Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen University, 2007.

[Thi18] François Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP*, pages 57–71, 2018.

[vOvR93] Vincent van Oostrom and Femke van Raamsdonk. Comparing Combinatory Reduction Systems and Higher-Order Rewrite Systems. In *Proceedings of the 1st International Workshop on Higher-Order Algebra, Logic and Term Rewriting*, Lecture Notes in Computer Science 816, 1993.

[Wah07] David Wahlstedt. *Dependent Type Theory with First-Order Parameterized Data Types and Well-Founded Recursion.* PhD thesis, Chalmers University of Technology, Sweden, 2007.

[WC03] Daria Walukiewicz-Chrząszcz. Termination of Rewriting in the Calculus of Constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.

[WCC08] Daria Walukiewicz-Chrzaszcz and Jacek Chrzaszcz. Consistency and Completeness of Rewriting in the Calculus of Constructions. *Logical Methods in Computer Science*, Volume 4, Issue 3, September 2008.

[Xi02] Hongwei Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.

Index

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Terminaison en présence de types dépendants et encodage par réécriture d'une théorie des types extensionelle avec polymorphisme d'univers

Mots clés : Théorie des types, Réécriture, Cadre logique, Terminaison, Eta-conversion, Polymorphisme d'univers

Résumé : Dedukti est un cadre logique dans lequel l'utilisateur encode la théorie qu'il souhaite utiliser à l'aide de règles de réécriture. Pour garantir la décidabilité du typage, il faut s'assurer que le système de réécriture utilisé est terminant.
Après avoir rappelé les propriétés des systèmes de types purs et leur extension avec de la réécriture, un critère de terminaison pour la réécriture d'ordre supérieur avec types dépendants est présenté. Il s'agit d'une extension de la notion de paires de dépendances au cas du lambda-pi-calcul modulo réécriture. Ce résultat se décompose en deux théorèmes principaux. Le premier stipule que la bonne fondaison de la relation d'appel définie à partir des paires de dépendances implique la normalisation forte du sytème de réécriture.
Le second résultat de cette partie décrit des conditions décidables suffisantes pour pouvoir utiliser le premier théorème. Cette version décidable du critère de terminaison est implémenté dans un outil appelé "SizeChange Tool".
La seconde partie de cette thèse est consacrée à l'utilisation du cadre logique Dedukti pour encoder une théorie des types riche. Nous nous intéressons plus particulièrement à la traduction d'un fragment d'Agda incluant deux fonctionnalités très répandues : l'extension de la conversion avec la règle eta et le polymorphisme d'univers.
Une fois encore, ce travail possède un versant théorique, avec des encodages prouvés corrects de ces deux fonctionalités dans le lambda-pi-calcul modulo réécriture, ainsi qu'une implémentation prototypique de traducteur entre Agda et Dedukti.

Title : Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting

Keywords : Type theory, Rewriting, Logical Framework, Termination, Eta-Conversion, Universe Polymorphism

Abstract : Dedukti is a logical framework in which the user encodes the theory she wants to use via rewriting rules. To ensure the decidability of typing, the rewriting system must be terminating.
After recalling some properties of pure type systems and their extension with rewriting, a termination criterion for higher-order rewriting with dependent types is presented. It is an extension of the dependency pairs to the lambda-pi-calculus modulo rewriting. This result features two main theorems. The first one states that the well-foundedness of the call relation defined from dependency pairs implies the strong normalization of the rewriting system.
The second result of this part describes decidable sufficient conditions to use the first one. This decidable version of the termination criterion is implemented in "SizeChange Tool".
The second part of this thesis is dedicated to the use of the logical framework Dedukti to encode a rich type theory. We are interested in a fragment of the logic beyond Agda which includes two widely used features: extension of conversion with the eta rule and universe polymorphism.
Once again, this work includes a theoretical part, with correct encodings of both features in the lambda-pi-calculus modulo rewriting, and a prototypical translator from Agda to Dedukti.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France