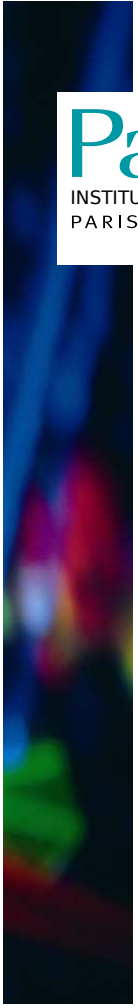
Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice

Ronan Saillard



ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 : Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps réel, robotique et automatique »

présentée et soutenue publiquement par

Ronan SAILLARD

le 25 septembre 2015

Vérification de typage pour le $\lambda\Pi$-Calcul Modulo : théorie et pratique

Typechecking in the $\lambda\Pi$-Calculus Modulo: Theory and Practice

Directeur de thèse : **Pierre JOUVELOT**
Maître de thèse : **Olivier HERMANT**

Jury

M. Andreas ABEL, Senior Lecturer, Chalmers and Gothenburg University	Rapporteur
M. Bruno BARRAS, Chargé de recherche, INRIA	Examinateur
M. Olivier HERMANT, Chargé de recherche, MINES ParisTech	Examinateur
M. Pierre JOUVELOT, Maître de recherche, MINES ParisTech	Examinateur
Mme Delia KESNER, Professeur, Université Paris Diderot	Présidente
Mme Brigitte PIENTKA, Associate Professor, McGill University	Rapporteur

T H È S E

MINES ParisTech
Centre de recherche en informatique
35 rue Saint-Honoré, 77305 Fontainebleau Cedex, France

Acknowledgements

I would like to thank Andreas Abel and Brigitte Pientka for the hard work of reviewing my manuscript, Bruno Barras and Delia Kesner for accepting to be part of my jury and Olivier Hermant and Pierre Jouvelot for guiding me for the last three years.

Many thanks also to all my colleagues at MINES ParisTech and at Inria: François, Corinne, Fabien, Claude, Benoit, Laurent, Claire, Pierre B., Pierre G., Pierre W., Florian, Nelson, Vivien, Dounia, Karel, Emilio, Arnaud, Imré, Catherine L., Ali, Raphaël, Frédéric G., Pierre H., Gilles, Frédéric B., Bruno B., Guillaume B^{I}., Guillaume B^{Y}., Virginie, Kailiang, Simon, Catherine D., Jean-Pierre, Robert, Vaston, David, Alejandro, Cecilia, Gaétan, Hermann, Hugo, Pierre N. and Benoit V.

Contents

List of Figures

Introduction

(English version follows.)

L'histoire commence par le développement d'un programme informatique nommé DEDUKTI. DEDUKTI est un vérificateur de preuves, c'est-à-dire un outil capable de vérifier automatiquement la validité d'une preuve mathématique.

Qu'est ce qu'une preuve ? Une preuve est une justification de la vérité d'une proposition. La preuve de la mortalité de Socrate en est un exemple classique :

Socrate est un homme ; tout homme est mortel ; donc, Socrate est mortel.

Dans cette preuve, on trouve les énoncés de deux hypothèses (*Socrate est un homme* et *tout homme est mortel*) et de la conclusion (*Socrate est mortel*). Le mot *donc* précise que la conclusion se déduit logiquement des hypothèses. Cependant, la nature de cette étape déductive reste implicite.

Les mathématiciens s'intéressent depuis longtemps à la notion de preuve et de raisonnement logique. La *théorie de la preuve* est la branche des mathématiques qui s'intéresse aux preuves en tant qu'objets mathématiques. Ce domaine a été particulièrement actif depuis le début du 19e siècle, et en particulier depuis la publication par Gottlob Frege de son *Begriffsschrift* (1879). Giuseppe Peano, Bertrand Russell, Richard Dedekind, David Hilbert, Kurt Gödel et Gerhard Gentzen, pour n'en citer que quelques uns, ont chacun apporté une importante contribution au domaine pendant la première moitié du 20e siècle. Ces travaux ont donné naissance à de nombreuses notions formelles de preuve. On peut, par exemple, voir une preuve comme un arbre étiqueté par des propositions.

$$\dfrac{\dfrac{\forall x, \mathbf{Homme}(x) \implies \mathbf{Mortel}(x)}{\mathbf{Homme}(\mathit{Socrate}) \implies \mathbf{Mortel}(\mathit{Socrate})}\ \textit{(Inst.)} \qquad \mathbf{Homme}(\mathit{Socrate})}{\mathbf{Mortel}(\mathit{Socrate})}\ \textit{(M-P)}$$

Cet arbre est une représentation formelle de la preuve de la mortalité de Socrate donnée plus haut. Comme le langage naturel est souvent ambigu, on écrit les propositions dans un langage plus précis et proche du vernaculaire mathématique. En haut, aux feuilles de l'arbre, $\forall x, \mathbf{Homme}(x) \implies \mathbf{Mortel}(x)$ et $\mathbf{Homme}(\mathit{Socrate})$ sont les hypothèses (ou axiomes) de la preuve ; la première signifie, *pour tout x, si x est un homme, alors x est mortel*, ou, plus simplement, *tout homme est mortel* ; la seconde signifie *Socrate est un homme*. En bas, la racine de l'arbre, $\mathbf{Mortel}(\mathit{Socrate})$, est la conclusion de la preuve ; elle signifie que *Socrate est mortel*.

L'arbre contient aussi deux nœuds internes, *(Inst.)* et *(M-P)*, qui correspondent à des étapes du raisonnement logique. Le premier nœud, *(Inst.)*, est une instance de

la règle d'*instanciation*. Cette règle dit que, pour tout objet o, il est logiquement valide de déduire $P(o)$ à partir de la proposition $\forall x.P(x)$, où $P(x)$ est une proposition dépendant d'une variable x, et $P(o)$ est la proposition $P(x)$ où l'on a remplacé les occurrences de la variable x par o. Ici la proposition $P(x)$ est $\mathbf{Homme}(x) \implies \mathbf{Mortel}(x)$ et l'objet o est *Socrate*. Le second nœud, *(M-P)*, est une instance de la règle du *Modus Ponens*. Cette règle dit que, pour toutes propositions P et Q, on peut déduire Q des deux propositions P et $P \implies Q$. Ici, on déduit $\mathbf{Mortel}(Socrate)$ à partir de $\mathbf{Homme}(Socrate)$ et $\mathbf{Homme}(Socrate) \implies \mathbf{Mortel}(Socrate)$.

Il est évident que cette preuve formelle de la mortalité de Socrate est beaucoup plus détaillée et précise que la preuve informelle avec laquelle on a commencé. Une preuve formelle permet donc d'avoir un plus grand degré de confiance que n'importe quelle autre notion de preuve puisqu'elle réduit tout à la validité d'un (petit) ensemble d'axiomes et de règles de déduction. Bien sur, en contrepartie, une preuve formelle est aussi plus fastidieuse à écrire puisque chaque étape du raisonnement logique doit être explicitée.

En choisissant le langage des propositions, les règles de déduction ainsi que les axiomes, on peut définir plusieurs types de logiques : classique, constructive, minimale, linéaire, modale, temporelle, etc.

Un vérificateur de preuves, c'est quoi ? Si on fixe l'ensemble des axiomes et des règles de déduction que l'on a le droit d'utiliser dans une preuve, la vérification de la validité d'une preuve formelle peut être automatisée. En effet, une preuve est valide si c'est un arbre dont les feuilles sont des axiomes, dont la racine est la conclusion et dont chaque nœud interne est une instance d'une règle de déduction. Ceci nous ouvre donc la voie à la vérification automatique de preuves par des programmes. On appelle ces programmes des vérificateurs de preuve.

Le premier vérificateur de preuve, *Automath* [NGdV94], a été conçu par Nicolaas Govert de Bruijn à la fin des années 60. Depuis, de nombreux vérificateurs ont été développés. Parmi les plus connus on peut citer *Agda* [BDN09], *Coq* [CDT], *Isabelle* [NWP02], *PVS* [ORS92], *Nuprl* [Kre], *Twelf* [PS99] ou encore *Beluga* [Pie10].

La preuve de programmes Parce que les preuves formelles sont très détaillées et peuvent être automatiquement vérifiées, elles permettent un haut niveau de confiance. La preuve de programmes est un important domaine d'application pour les preuves formelles. Le but est de prouver qu'un programme informatique correspond bien à sa spécification, autrement dit, qu'il ne contient pas de *bogues*. Ces *méthodes formelles* sont déjà utilisées dans l'industrie, pour prouver que des systèmes critiques ne contiennent pas d'erreurs, par exemple dans les domaines du transport (avion, train, métro) ou de la sécurité informatique (cryptographie, protocoles).

Comment fonctionne un vérificateur de preuves ? La majorité des vérificateurs de preuves s'appuient sur une correspondance forte entre preuves formelles et programmes fonctionnels que l'on appelle la *correspondance de Curry-Howard* ou *interprétation formule/type*.

Cette correspondance permet de remplacer la vérification qu'une preuve correspond à une proposition par la vérification qu'un programme fonctionnel a un type donné, l'intuition étant que les règles de déduction peuvent être vues comme des règles de typage pour programmes fonctionnels.

Par exemple, la règle du *Modus Ponens*, que l'on a déjà évoquée,

$$\frac{P \Longrightarrow Q \qquad P}{Q}$$

correspond à la règle de typage de l'application d'une fonction f à un argument a en programmation fonctionnelle, un style de programmation inspiré du λ-calcul développé par Alonzo Church.

$$\frac{f : A \longrightarrow B \qquad a : A}{f\ a : B}$$

Cette règle permet, à partir d'une fonction de type $A \longrightarrow B$ (c'est-à-dire une fonction qui prend un argument de type A et produit un élément de type B) et d'un argument de type A, de construire l'application $(f\ a)$ de type B. Par exemple, si 42 est de type `Entier` (le type des entiers naturels) et `EstPair` est une fonction de type `Entier` $\longrightarrow$ `Booleen`, c'est-à-dire une fonction des entiers vers les booléens (le type de *vrai* et *faux*), alors l'expression (`EstPair` 42) est de type `Booleen`.

On peut remarquer qu'il y a une correspondance (à un renommage près : P en A, Q en B et $\Longrightarrow$ en $\longrightarrow$) entre les propositions de la règle de *Modus Ponens* et les types de la règle de typage de l'application. Cette correspondance peut être étendue à d'autres règles de déduction et de typage, construisant ainsi une connexion très forte entre les systèmes logiques et les systèmes de types. De plus, grâce à cette connexion, on peut voir un programme de type A comme une preuve de la proposition correspondant à A. À partir de cette idée, les vérificateurs de preuves modernes, implémentent généralement à la fois un système logique et un langage de programmation.

DEDUKTI implémente une variante de la *correspondance formule/type*, la *correspondance jugement/type* [HHP93]. L'idée de base est la même : on réduit un problème de vérification de preuve à un problème de vérification de type. Par contre on abandonne la correspondance entre les propositions et les types. Les propositions ainsi que les règles de déduction et axiomes sont traduits en programmes en utilisant des encodages spécifiques. Choisir un encodage permet de choisir une logique. Ainsi, DEDUKTI est un vérificateur de preuves universel car il est indépendant de la logique considérée. On appelle un tel vérificateur un *logical framework* (cadre logique).

DEDUKTI est utilisé comme *back-end* par de nombreuses implémentations d'encodages.

- CoqInE [BB12] (Coq In dEdukti) produit des preuves DEDUKTI à partir de preuves *Coq* [CDT].
- Holide [AB14] (HOL In DEdukti) produit des preuves DEDUKTI à partir de preuves *HOL* [Har09] au format *Open Theory* [Hur11].
- Focalide [Cau] (FoCaLize In DEdukti) produit des fichiers DEDUKTI à partir des développements *FoCaLize* [HPWD].
- Krajono [Ass] (*Pencil* en espéranto) produit des preuves DEDUKTI à partir de preuves *Matita* [ARCT11].
- iProver Modulo [Bur13] est une extension de *iProver* [Kor08], un prouveur automatique de théorèmes fondé sur la méthode de résolution, permettant le support de la déduction modulo et produisant des preuves DEDUKTI.

- Zenon Modulo [DDG+13] est une extension de *Zenon* [BDD07], un prouveur automatique de théorèmes fondé sur la méthode des tableaux, avec du typage et de la déduction modulo, produisant des preuves DEDUKTI.

Quelle est la particularité de DEDUKTI ? Les vérificateurs de preuves fondés sur la *correspondance de Curry-Howard* implémentent en même temps un système logique et un langage de programmation. Cela signifie que l'on peut calculer *avec* les preuves (puisque ce sont aussi des programmes), mais cela permet aussi de calculer *dans* les preuves. En effet, les vérificateurs de preuves identifient généralement les propositions qui sont identiques à un calcul près. Par exemple, la proposition *2+2 = 4* est identifiée à la proposition *4=4*, puisque le résultat du calcul *2+2* est *4*. Cela veut dire que prouver que 2+2 est égal à 4 se réduit à un simple calcul et à l'utilisation du principe de réflexivité de l'égalité :

$$\dfrac{\dfrac{\forall x.x = x}{4 = 4}\ \textit{(Inst.)}}{2+2=4}\ \textit{(Calcul)}$$

La particularité de DEDUKTI est de permettre de facilement étendre cette notion de calcul grâce à l'ajout de règles de réécriture. Par exemple, l'addition *d'entiers de Peano* peut être calculée grâce aux règles suivantes :

$$\begin{array}{ll} n+0 & \hookrightarrow n \\ n+(S\ m) & \hookrightarrow S\ (n+m) \end{array}$$

Un entier de Peano est un entier représenté par un mot de la forme $S\ (\ldots(S\ 0))$. Le symbole 0 signifie zéro et le symbole S (successeur) incrémente un entier de un. L'entier 2 est donc représenté par $S\ (S\ 0)$ et 4 par $S\ (S\ (S\ (S\ 0)))$. La première règle de réécriture dit que, pour tout n, $n+0$ vaut n. La seconde règle dit que, pour tout n et m, $n+(m+1)$ vaut $(n+m)+1$.

Les règles de réécriture de DEDUKTI permettent donc d'avoir un contrôle précis sur la notion de calcul.

Une brève histoire de DEDUKTI Le projet DEDUKTI a été initié par Dowek comme un vérificateur de preuves/types fondé sur le $\lambda\Pi$-Calcul Modulo [CD07], un formalisme à base de types dépendants (c'est-à-dire des types dépendant de valeurs) et de règles de réécriture. Une première version a été développée par Boespflug [Boe11], puis par Carbonneaux [BCH12]. Cette première version implémentait une architecture logicielle originale. La vérification de type se passait en deux étapes : d'abord le problème était lu par le programme qui générait un vérificateur dédié ; dans un second temps, le code généré était compilé et exécuté pour obtenir le résultat. Cette architecture permettait d'implémenter les lieurs en utilisant la *syntaxe abstraite d'ordre supérieure* [PE88] et la réduction en utilisant la *normalisation par évaluation* [BS91]. De plus, le calcul implémenté était *sans contexte* [Boe11]. Plusieurs variantes de cette première version ont existé mettant en œuvre différents langages de programmation : d'abord une version *Haskell* générant du code *Haskell*, ensuite une version *Haskell* générant du code *Lua*, et enfin, une version *C* générant du code *Lua*.

Pour pallier des problèmes de performance et de passage à l'échelle, nous avons développé une nouvelle version en *OCaml* implémentant une architecture plus standard (une seule étape). Cette thèse a pour but de décrire les fondations théoriques de cette nouvelle version, étendue, de DEDUKTI.

De quoi parle cette thèse ? Au fil du temps, l'implémentation de DEDUKTI a évolué : des fonctionnalités ont été ajoutées ; plus de systèmes de réécriture ont été supportés, etc. D'un autre côté, la définition du $\lambda\Pi$-Calcul Modulo a peu évolué depuis l'article original de Cousineau et Dowek [CD07]. De plus, ce premier papier ne se concentre pas sur l'étude du $\lambda\Pi$-Calcul Modulo, mais cherche plutôt à motiver son utilisation à travers un exemple détaillé, l'encodage des *Systèmes de Types Purs Fonctionnels*. Par conséquent, il y avait un décalage entre le $\lambda\Pi$-Calcul Modulo de Cousineau et Dowek et le calcul implémenté dans DEDUKTI. En particulier :

- la façon de vérifier le bon typage des règles de réécriture dans DEDUKTI etait plus générale que la définition initiale du $\lambda\Pi$-Calcul Modulo, notamment parce que DEDUKTI procède itérativement : les règles préalablement ajoutées au système sont utilisées pour typer les nouvelles ;
- les conditions nécessaires à la décidabilité de la vérification de type et nécessaires à la correction de l'algorithme de vérification de type étaient mal connues ;
- peu d'outils théoriques permettaient de vérifier que ces conditions étaient vérifiées pour un système de réécriture donné.

Cette thèse répond à ces problèmes spécifiques de trois façons :

- en proposant une nouvelle version du $\lambda\Pi$-Calcul Modulo facilement comparable avec son implémentation dans DEDUKTI ;
- en effectuant une étude théorique détaillée de ce nouveau calcul, en se concentrant en particulier sur les conditions nécessaires au système de réécriture rendant la vérification de type décidable et plus généralement permettant au système de type de *bien se comporter* ;
- et en donnant des critères effectifs permettant de garantir ces conditions.

Résumé et contributions de la thèse

- Le **chapitre 1** rappelle certaines notions classiques de théorie de la réécriture concernant les systèmes de réduction abstraits, la réécriture du premier ordre, le λ-calcul et leur combinaison qui seront utilisées dans les chapitres suivants. On s'intéresse en particulier aux résultats de confluence.
- Le **chapitre 2** donne une nouvelle présentation du $\lambda\Pi$-Calcul Modulo correspondant au calcul implémenté par DEDUKTI. Cette nouvelle version améliore celle de Cousineau et Dowek de deux manières. D'abord on définit une notion de réécriture sur les termes sans aucune notion de typage. Ceci permet de rendre sa comparaison avec son implémentation plus directe. Ensuite, on explicite et on clarifie le typage des règles de réécriture.

  On procède à une étude théorique précise du $\lambda\Pi$-Calcul Modulo. En particulier, on met en exergue les conditions qui assurent que le système de types vérifie des propriétés élémentaires telles que la préservation du typage par réduction ou l'unicité des types. Ces conditions sont la compatibilité du produit et le bon typage des règles de réécriture. Pour finir, on considère une extension du $\lambda\Pi$-Calcul Modulo avec du polymorphisme et des opérateurs de type que l'on appelle le Calcul des Constructions Modulo.

- Le **chapitre 3** étudie la propriété de bon typage des règles de réécriture. Une règle de réécriture est bien typée si elle préserve le typage. Partant d'un critère simple, à savoir que le membre gauche de la règle doit être algébrique et les membres gauche et droit doivent avoir le même type, on généralise progressivement le résultat pour considérer des membres gauches non algébriques et mal typés. Cette généralisation est particulièrement importante en présence de types dépendants, pour permettre de conserver des règles de réécriture linéaires à gauche et préserver la confluence du système de réécriture. On donne aussi une caractérisation exacte de la notion de bon typage pour les règles de réécriture sous forme d'un problème d'unification et on prouve son indécidabilité.

- Le **chapitre 4** définit une notion de réécriture modulo β pour le $\lambda\Pi$-Calcul Modulo. En partant des observations que (1) la confluence du système de réécriture est une propriété vivement souhaitée car elle a pour conséquence la propriété de la compatibilité du produit ainsi que, avec la terminaison, la décidabilité de la congruence et que (2) la confluence est facilement perdue lorsque les règles de réécriture filtrent sous les abstractions, on propose une nouvelle notion de réécriture qui réconcilie confluence et filtrage sous les abstractions. Cette nouvelle notion est définie à travers un encodage des termes vers un *système de réécriture d'ordre supérieur*. Ceci permet d'importer dans $\lambda\Pi$-Calcul Modulo les résultats de confluence existants pour les systèmes d'ordre supérieur. On détaille aussi comment la réécriture modulo β peut être efficacement implémentée par la compilation des règles de réécriture en arbres de décision.

- Le **chapitre 5** considère les règles de réécriture non linéaires à gauche. Combinées avec la β-réduction, ces règles génèrent généralement un système de réécriture non confluent. Ceci est un problème car la confluence est notre outil principal pour prouver la compatibilité du produit. On prouve que la propriété de compatibilité du produit est toujours vérifiée (même sans la confluence) lorsque les règles de réécriture sont toutes au niveau objet. Ensuite on étudie cette propriété en présence de règles non linéaires à gauche et de règles au niveau type. Pour cela, on introduit une variante du $\lambda\Pi$-Calcul Modulo où la conversion est contrainte par une notion de typage faible.

- Le **chapitre 6** décrit les algorithmes de vérification de type pour les différents éléments du $\lambda\Pi$-Calcul Modulo : termes, contextes locaux et contextes globaux. On montre aussi que ces algorithmes sont corrects et complets en utilisant les résultats des chapitres précédents.

Introduction

The story begins with the development of a piece of software called DEDUKTI. DEDUKTI is a proof checker, that is a tool able to automatically check the validity of mathematical proofs.

What is a proof? A proof is a justification of the truth of a proposition. An early example is the proof of the proposition *Socrate is mortal*:

> Socrate is a man; men are mortal; therefore, Socrate is mortal.

In this proof we can find the statement of two hypotheses (*Socrate is a man* and *Men are mortal*) and the conclusion *Socrate is mortal*. The word *therefore* suggests that the conclusion can be logically deduced from the hypotheses. The nature of this deductive step is, however, left implicit.

Mathematicians have been interested for a long time in the notion of proof and in the rules of logical reasoning. *Proof Theory* is the branch of mathematics that studies proofs as mathematical objects. This domain has been particularly active since the end of the 19th century, starting with the work of Gottlob Frege in his *Begriffsschrift* (1879). Giuseppe Peano, Bertrand Russell, Richard Dedekind, David Hilbert, Kurt Gödel and Gerhard Gentzen, to cite only a few, brought important contributions to the field during the first half of the 20th century. These works gave rise to many formal notions of proof. A particularly convenient one is the presentation of proofs as trees labeled by propositions.

$$\dfrac{\dfrac{\forall x, \mathbf{IsAMan}(x) \implies \mathbf{IsMortal}(x)}{\mathbf{IsAMan}(\mathit{Socrate}) \implies \mathbf{IsMortal}(\mathit{Socrate})}\ \textit{(Inst.)} \qquad \mathbf{IsAMan}(\mathit{Socrate})}{\mathbf{IsMortal}(\mathit{Socrate})}\ \textit{(M-P)}$$

This tree is a formal representation of the proof that Socrate is mortal given above. Natural language being often ambiguous, we write the propositions in a more precise language, close to the mathematical vernacular. At the top, the leaves of the tree $\forall x, \mathbf{IsAMan}(x) \implies \mathbf{IsMortal}(x)$ and $\mathbf{IsAMan}(\mathit{Socrate})$ are the hypotheses (or axioms) of the proof; the first one stands for, *for all x, if x is a man, then x is mortal*, or, in short, *men are mortal*; the second one stands for *Socrate is a man*. At the bottom, the root of the tree $\mathbf{IsMortal}(\mathit{Socrate})$ is the conclusion of the proof; it stands for *Socrate is mortal*.

The tree contains also two internal nodes, *(Inst.)* and *(M-P)*; they correspond to the deductive steps of the logical reasoning. The first one *(Inst.)* is an instance of the *Instantiation* rule. This rule states that, for any *object o*, it is logically valid to deduce $P(o)$ from a proposition $\forall x.P(x)$, where $P(x)$ is a proposition depending on the *variable x* and $P(o)$ is the proposition $P(x)$ where we replaced every occurrence

of x by o. Here the proposition $P(x)$ is **IsAMan**$(x) \implies$ **IsMortal**(x) and the object o is *Socrate*. The second one *(M-P)* is an instance of the *Modus Ponens* rule. This rule says that, for any propositions P and Q, we can deduce Q from the two propositions P and $P \implies Q$. Here, we deduce **IsMortal**(*Socrate*) from **IsAMan**(*Socrate*) and **IsAMan**(*Socrate*) $\implies$ **IsMortal**(*Socrate*).

As we can see, the formal proof that Socrate is mortal is much more detailed and precise than the informal proof we began with. This allows having a much higher degree of confidence in formal proofs than in any other notion of proof as everything is reduced to the validity of a (small) set of axioms and a (small) set of deductive rules. Of course, it is also more tedious to write since every logical step is made explicit.

By choosing the language of propositions, the deductive rules and the axioms, we can define several kinds of logics: classical, constructive, minimal, linear, modal, temporal, etc.

What is a proof checker? If we fix the set of axioms and the set of deductive rules that can be used in a proof, the verification of the validity of a formal proof can be mechanized. Indeed, a proof is valid if it is a tree where the leaves are axioms, the root is the conclusion and each internal node is an instance of a deductive rule. This opens the possibility of automatic proof verification by programs. We call these programs proof checkers.

The first proof checker, *Automath* [NGdV94], has been designed by Nicolaas Govert de Bruijn in the late sixties. Since then, many other proof checkers have been developed. Among the better known are *Agda* [BDN09], *Coq* [CDT], *Isabelle* [NWP02], *PVS* [ORS92], *Nuprl* [Kre], *Twelf* [PS99] and *Beluga* [Pie10].

Proof of programs Because formal proofs are very detailed and can be automatically verified, they offer a strong degree of confidence. An important application domain for formal proofs is the proofs of programs. The goal is to prove that a computer program correspond to its specification. In other words, we prove that a program has no *bugs*. This has already been used in the industry to prove that critical systems were bug-free, for instance in the transport (airplane, railways, subway) or security (cryptography, protocols).

How do proof checkers work? Most existing proof checkers are based on a strong correspondence between formal proofs and functional programs known as the *Curry-Howard correspondence*, or the *formulas-as-types interpretation*.

Roughly speaking, the correspondence states that it is the same thing to check that a proof justifies a given proposition, or to check that a functional program has a given type. The intuition is that deduction rules can be seen as rules for typing functional programs.

For instance the *Modus Ponens* rule, that we have already mentioned,

$$\frac{P \implies Q \qquad P}{Q}$$

corresponds to the rule for typing the application of a function f to an argument a in functional programming, a programming style inspired by the λ-calculus developed by Church.

$$\frac{f : A \longrightarrow B \qquad a : A}{f\ a : B}$$

This rule says that, if f is a function of type $A \longrightarrow B$, meaning that it takes an argument of type A and produces an element of type B, and a is an argument of type A, then the application $(f\ a)$ of f to a is an expression of type B. For instance, if 42 has type `Integer` (the type of integers) and `isEven` is a function of type `Integer` $\longrightarrow$ `Boolean`, that is to say a function from integers to booleans (the type of true and false), then the expression (`isEven` 42) has type `Boolean`.

We can see that there is a correspondence (up to some renaming: P to A, Q to B and $\Longrightarrow$ to $\longrightarrow$) between the propositions occurring in the *Modus Ponens* rule and the types occurring in the typing rule for the application. The correspondence can be extended to other logical and typing rules, building a strong connection between logical systems and type systems. Moreover, following this connection, a program of type A can be seen as a proof of the proposition corresponding to A. Based on this idea, modern proof checkers usually implement a calculus that is at the same time a logical system and a programming language.

DEDUKTI implements a variant of the *formulas-as-types* correspondence known as the *judgment-as-type* correspondence [HHP93]. The basic idea is the same: we reduce the problem of proof checking to the problem of type checking. However, we give up the correspondence between propositions and types. The propositions as well as the deductive steps and the axioms are translated to programs using specific encodings. Choosing an encoding allows choosing a logic. By this means, DEDUKTI is a *universal* proof checker, as it is logic-agnostic. Such a proof checker is called a logical framework.

DEDUKTI has been used as a backend by many implementations of encodings.

- CoqInE [BB12] (Coq In dEdukti) produces DEDUKTI proofs from *Coq* [CDT] proofs.
- Holide [AB14] (HOL In DEdukti) produces DEDUKTI proofs from *HOL* [Har09] proofs, using the *Open Theory* [Hur11] standard.
- Focalide [Cau] (FoCaLize In DEdukti) produces DEDUKTI files from *FoCaLize* [HPWD] developments.
- Krajono [Ass] (*Pencil* in Esperanto) produces DEDUKTI files from *Matita* [ARCT11] proofs.
- iProver Modulo [Bur13] is an extension of the resolution automated theorem prover *iProver* [Kor08] with deduction modulo, producing DEDUKTI files.
- Zenon Modulo [DDG+13] is an extension of the tableaux-based automated theorem prover *Zenon* [BDD07] with typing and deduction modulo, producing DEDUKTI files.

What is so special about DEDUKTI? Proof-checkers based on the Curry-Howard correspondence implement at the same time a logical system and a programming language. This means that we can compute *with* proofs (proofs are programs) but they also allow computing *in* proofs. Indeed, proof checkers usually identify propositions that are the same up to some computation. For instance, the proposition *2+2 = 4* is identified with the proposition *4 = 4* because 2+2 computes to 4. This means

that proving that 2+2 is equal to 4 is just a matter of performing a simple computation and using the reflexivity of the equality:

$$\dfrac{\dfrac{\forall x.x = x}{4 = 4}\ \textit{(Inst.)}}{2+2=4}\ \textit{(Computation)}$$

The distinctive feature of DEDUKTI is to provide a simple means to extend this notion of computation through rewrite rules. For instance, typical rewrite rules for the addition on *Peano integers* are:

$$\begin{array}{ll} n+0 & \hookrightarrow n \\ n+(S\ m) & \hookrightarrow S\ (n+m) \end{array}$$

Peano integers is a simple representation of integers as words of the form $S\ (\ldots (S\ 0))$. The symbol 0 is the zero and the symbol S (successor) adds one to an integer. This means that 2 is represented by $S\ (S\ 0)$ and 4 by $S\ (S\ (S\ (S\ 0)))$. The first rewrite rule says that, for all n, $n+0$ computes to n. The second one says that, for all n and m, $n+(m+1)$ computes to $(n+m)+1$.

Rewrite rules allow DEDUKTI users to have a precise control over the notion of computation.

A short history of DEDUKTI DEDUKTI has been initiated by Dowek as a proof checker/type checker based on the $\lambda\Pi$-Calculus Modulo [CD07], a formalism featuring dependent types (*i.e.*, types depending on values) and rewrite rules (the reason for the *Modulo* qualifier). A first version has been developed by Boespflug [Boe11] and then by Carbonneaux [BCH12]. This first version featured an original architecture. The type-checking process was performed in two steps: first the input problem was parsed and a dedicated type-checker was generated; second the generated code was compiled and run to obtain the result. This architecture allowed implementing binders using *Higher-Order Abstract Syntax* [PE88] and reduction using *Normalization by Evaluation* [BS91]. Moreover, the calculus implemented was *context-free* [Boe11]. Several variants of this first version have been implemented using different programming languages: first a *Haskell* version generating *Haskell* code, then a *Haskell* version generating *Lua* code and finally a *C* version generating *Lua* code.

Because all these variants suffered from performance and scaling issues, we have developed a new version in *OCaml* implementing a more standard (one step) architecture. This thesis describes the theoretical underpinnings of this prototype.

What is addressed in this thesis? Through time, the implementation of DEDUKTI has evolved: features were added; more rewrite systems were supported, etc. On the other hand, the definition of the $\lambda\Pi$-Calculus Modulo did not evolve since the seminal paper of Cousineau and Dowek [CD07]. Moreover, this first paper did not focus on the theoretical side of the $\lambda\Pi$-Calculus Modulo but rather motivated its use as a logical framework. As a result, there was a gap between Cousineau and Dowek's $\lambda\Pi$-Calculus Modulo and the calculus implemented in DEDUKTI. In particular:

- the way rewrites rules are typed in DEDUKTI goes beyond the initial definition of the $\lambda\Pi$-Calculus Modulo as it is iterative; rewrite rules previously added can be used to type new ones;

- the conditions under which type-checking in the $\lambda\Pi$-Calculus Modulo is decidable and the conditions under which DEDUKTI is sound had not been studied in detail;
- there were few theoretical tools allowing verifying that these conditions hold for a given set of rewrite rules.

This thesis gives answers to these specific problems in three ways:

- by proposing a new version of the $\lambda\Pi$-Calculus Modulo that is easy to compare with the calculus implemented by DEDUKTI;
- by performing a detailed theoretical study of this new calculus, focused in particular on making explicit the conditions under which type-checking is decidable and, more generally, the conditions under which the type-system is *well-behaved*;
- and by designing effective criteria on the rewrite system to ensure that these conditions hold.

Outline and contributions of the thesis

- **Chapter 1** reviews some basic notions and results in rewriting theory about abstract reduction systems, term rewriting systems, the λ-calculus and their combination. In particular, we are interested in confluence results.
- **Chapter 2** introduces a new presentation of the $\lambda\Pi$-Calculus Modulo, the calculus implemented in DEDUKTI. This new version aims at improving Cousineau and Dowek's $\lambda\Pi$-Calculus Modulo by two modifications. First we define the notion of rewriting on untyped terms. This makes the calculus easier to compare with its implementation in DEDUKTI. Second, we make explicit and we clarify the typing of the rewrite rules.

  We undertake a precise theoretical study of the $\lambda\Pi$-Calculus Modulo. In particular, we put forward two conditions ensuring that the typing system verifies basic properties such as *subject reduction* and *uniqueness of types*. These conditions are *product compatibility* and *well-typedness of rewrite rules*. Finally, we consider an extension of the $\lambda\Pi$-Calculus Modulo with polymorphism and type operators that we call the Calculus of Constructions Modulo.
- **Chapter 3** investigates the property of well-typedness for rewrite rules. A rewrite rule is well-typed if it preserves typing. Starting from the simple criterion that the rewrite rules should be left-algebraic and both sides of the rule should have the same type, we progressively generalize the result to allow non-algebraic and ill-typed left-hand sides. This latter generalization is particularly important to keep rewrite rules left-linear in presence of dependent typing and to preserve the confluence of the rewriting system. We also give an exact characterisation of well-typedness for rewrite rules as a unification problem and we prove the undecidability of the problem.
- **Chapter 4** introduces a notion of rewriting modulo β for the $\lambda\Pi$-Calculus Modulo. Starting from the observations that (1) the confluence of the rewriting system is a very desirable property as it implies product compatibility and,

together with termination, the decidability of the congruence and (2) confluence is easily lost when we allow matching under binders, we introduce the notion of rewriting modulo β to reconcile confluence and matching under binders. This new notion of rewriting is defined through an encoding in Higher-Order Rewrite Systems. This allows bringing to the $\lambda\Pi$-Calculus Modulo the confluence criteria designed for Higher-Order Rewrite Systems. We also detail how rewriting modulo β can be efficiently implemented by compiling the rewrite rules to decision trees.

- **Chapter 5** considers non left-linear rewrite rules. Non-left linear rewrite rules usually generate non-confluent rewriting systems when combined with β-reduction. This is an issue because confluence is our main tool to prove product compatibility. Adapting previous works, we prove that product compatibility holds when the rewrite rules are at object-level only, even if confluence does not hold. Then we study the problem of proving product compatibility in presence of non left-linear and type-level rewrite rules. For this we introduce a variant of the $\lambda\Pi$-Calculus Modulo where the conversion is constrained to verify a weak notion of typing.

- **Chapter 6** gives algorithms to type-check the different elements of the $\lambda\Pi$-Calculus Modulo: terms, local contexts, rewrite rules and global contexts. These algorithms are shown to be sound and complete using the results of the previous chapters.

Chapter 1

Preliminaries

Résumé Ce chapitre rappelle certaines notions classiques de théorie de la réécriture concernant les systèmes de réduction abstraits, la réécriture du premier ordre, le λ-calcul et leur combinaison qui seront utilisées dans les chapitres suivants. On s'intéresse en particulier aux résultats de confluence.

This short chapter reviews some basic notions and results about abstract reduction systems, term rewriting systems, λ-calculus and their combination that we will use in the next chapters. In particular, we are interested in confluence results.

1.1 Abstract Reduction Systems

The notion of abstract reduction system is, as its name suggests, the most abstract definition of rewriting that exists. In particular, it makes no assumptions on the nature of the objects being reduced (or rewritten). Though very simple, this notion allows us to formally define basic properties of rewriting such as confluence or termination.

Definition 1.1.1 (Abstract Reduction System)**.** *An* abstract reduction system *(ARS) is a pair made of a set $\mathscr{A}$ and a binary relation $\to$ on $\mathscr{A}$* (i.e., $\to \subset \mathscr{A} \times \mathscr{A}$)*.*

Notation 1.1.2.

- *We use the infix notation $x \to y$ to denote $(x, y) \in \to$.*
- *We write $\to^*$ for the reflexive and transitive closure of $\to$.*
- *We write $\equiv$ for the reflexive, symmetric and transitive closure of $\to$.*

Definition 1.1.3. *Two objects $x, y \in \mathscr{A}$ are joinable (written $x \downarrow y$), if there exists z such that $x \to^* z$ and $y \to^* z$.*

Definition 1.1.4. *An element $x \in \mathscr{A}$ is* normal *if there is no y such that $x \to y$.*

Definition 1.1.5. *An abstract reduction system is said to be:*

- locally confluent *(or* weakly confluent*) when, for all x, y, z, if $x \to y$ and $x \to z$, then $y \downarrow z$;*

- confluent *when, for all x, y, z if $x \to^* y$ and $x \to^* z$, then $y \downarrow z$;*
- normalizing *(or* weakly normalizing*) when, for all x, there exists y normal such that $x \to^* y$.*
- terminating *(or* strongly normalizing*) when there is no infinite reduction chain $a_1 \to a_2 \to \ldots a_n \to \ldots$.*

Theorem 1.1.6 (Newman's Lemma [New42])**.** *A terminating ARS is confluent if and only if it is locally confluent.*

1.2 Term Rewriting Systems

A (first-order) term rewriting system is an abstract rewriting system where the objects are first-order terms, and where the reduction relation is given by a set of rewrite rules.

1.2.1 Definition

Definition 1.2.1 (Signature)**.** *A* signature *is a set Σ of constant symbols together with an arity function from Σ to positive integers.*

Definition 1.2.2 (Term)**.** *If Σ is a signature and $\mathcal{V}$ a set of variables (disjoint from Σ and infinite), then the set $T(\Sigma, \mathcal{V})$ of* (first-order) terms *over Σ is defined inductively as follows:*

- *a variable $v \in \mathcal{V}$ is a term;*
- *if $f \in \Sigma$ is a constant symbol of arity n and $t_1, \ldots, t_n$ are n terms, then $f(t_1, \ldots, t_n)$ is a term.*

If t is a term, we write $Var(t)$ the set of variables occurring in t.

Definition 1.2.3 (Substitution)**.** *A* substitution *is a function from the set of variables to the set of terms with a finite domain. The* domain *of a substitution is the set $\{ x \in \mathcal{V} \mid \sigma(x) \neq x \}$.*

If σ is a substitution and t is a term, we write $\sigma(t)$ the term t where we replaced the variables by their image by σ.

Definition 1.2.4 (Rewrite Rule)**.** *A* rewrite rule *is a pair (l, r) of terms such that l is not a variable and $Var(r) \subset Var(l)$.*

We write $(l \hookrightarrow r)$ for the rewrite rule (l, r).

Definition 1.2.5 (Term Rewriting System)**.** *A* Term Rewriting System *(TRS) for a signature Σ and a set of variables $\mathcal{V}$ is a set of rewrite rules R over the signature Σ.*

When it is convenient, we identify a TRS with its underlying ARS $(T(\Sigma, \mathcal{V}), \to_R)$ where $\to_R$ is the relation on $T(\Sigma, \mathcal{V})$ defined inductively as follows:

- *$\sigma(l) \to_R \sigma(r)$ if $(l \hookrightarrow r) \in R$ and σ is a substitution;*
- *$f(t_1, \ldots, t_n) \to_R f(s_1, \ldots, s_n)$ if, for some i, $t_i \to_R s_i$ and, for all $j \neq i$, $t_j = s_j$.*

1.2.2 Critical Pairs

Definition 1.2.6 (Position and Subterm)**.** *Let t be a term. The set of* positions *in t, $Pos(t)$, of sequences of integers and the* subterm $t_{|p}$ *of t at position $p \in Pos(t)$ are defined inductively as follows:*

- *if t is a variable, then $Pos(t) = \{\epsilon\}$ and $t_{|\epsilon} = t$;*
- *if $t = f(t_1, \ldots, t_n)$, then $Pos(t) = \{\epsilon\} \cup \{1.q | q \in Pos(t_1)\} \cup \ldots \cup \{n.q | q \in Pos(t_n)\}$, $t_{|\epsilon} = t$ and $t_{|i.q} = t_{i|q}$.*

Definition 1.2.7 (Most General Substitution)**.** *Let u, v be two terms.*
A substitution σ is a most general substitution *for u and v if:*

- *$\sigma(u) = \sigma(v)$;*
- *for all σ_0 such that $\sigma_0(u) = \sigma_0(v)$, there exists a substitution δ such that, for all x, $\sigma_0(x) = \delta(\sigma(x))$.*

Notation 1.2.8. *Let t and u be two terms and p be a position in t. We write $t[u]_p$ for the term t where we replaced the subterm at position p by u.*

Definition 1.2.9 (Critical Pair)**.** *Let $l_i \hookrightarrow r_i$ for $(i = 1, 2)$ be two rewrite rules. Suppose that they do not share any variable (variables can be renamed if needed).*
If there exists $p \in Pos(l_1)$ such that $l_{1|p}$ is not a variable and σ is a most general unifier of $(l_{1|p}, l_2)$ then we have the following reductions $\sigma(l_1) \to \sigma(r_1)$ and $\sigma(l_1) \to (\sigma(l_1))[\sigma(r_2)]_p$.
We say that the pair $((\sigma(l_1))[\sigma(r_2)]_p, \sigma(r_1))$ is a critical pair *and that the two rewrite rules* overlap*.*
We write $(a \ltimes b)$ when (a, b) is a critical pair.

Theorem 1.2.10 (Critical Pair Theorem [KB83])**.** *A TRS is locally confluent if and only if its critical pairs are joinable.*

Combined with Newman's Lemma (Theorem 1.1.6), we get the following result.

Corollary 1.2.11. *A terminating TRS is confluent if and only if its critical pairs are joinable.*

1.2.3 Confluence

A TRS which is orthogonal, meaning that it is left-linear and does not have critical pairs, is confluent.

Definition 1.2.12 (Linearity)**.**

- *A term is* linear *if no variable occurs twice in it.*
- *A rewrite rule $(l \hookrightarrow r)$ is* left-linear *if l is linear.*
- *A TRS is* left-linear *if all its rewrite rules are left-linear.*

Definition 1.2.13 (Orthogonality)**.** *A rewrite system is* orthogonal *if it is left-linear and includes no critical pairs.*

Theorem 1.2.14 (Confluence by Orthogonality [Ros73, Hue80])**.** *Orthogonal systems are confluent.*

The criterion of orthogonality can be generalized to TRSs whose critical pairs are *parallel closed*.

Definition 1.2.15 ($\rightrightarrows$)**.** *Let R be a TRS. The relation $\rightrightarrows_R$ of* parallel reduction *is defined inductively as follows:*

- $x \rightrightarrows_R x$;
- $f(t_1,\ldots,t_n) \rightrightarrows_R f(s_1,\ldots,s_n)$ *if, for all i, $t_i \rightrightarrows_R s_i$;*
- $t \rightrightarrows_R s$ *if* $t \to_R s$.

Definition 1.2.16 (Parallel Closed TRS)**.** *A TRS R is* parallel closed *if $t_1 \rightrightarrows_R t_2$ for every critical pair $t_1 \ltimes t_2$.*

Theorem 1.2.17 (Parallel Closure Theorem [Hue80])**.** *Left-linear and parallel closed systems are confluent.*

Another way to get confluence is by combining confluent TRSs provided that their signatures are disjoint.

Theorem 1.2.18 (Modularity of Confluence [Toy87])**.** *Let R_1, R_2 be two TRSs on $T(\Sigma_1,\mathcal{V})$ and $T(\Sigma_2,\mathcal{V})$ respectively.*

If Σ_1 and Σ_2 are disjoint signatures and both R_1 and R_2 are confluent, then the TRS $R_1 \cup R_2$ on $T(\Sigma_1 \cup \Sigma_2,\mathcal{V})$ is confluent.

1.3 The λ-Calculus

The λ-calculus is a formalism invented by Alonzo Church in the 1930s. It has been designed to capture the most basic aspects of the ways that functions can be combined to form other functions. As it makes use of bound variables, it is an abstract reduction system but not a term rewriting system.

Definition 1.3.1 (λ-Terms)**.** *We assume given an infinite set of variables. The set of* λ-terms *built from the set of constants $\mathscr{C}$ is defined as follows:*

- *any* variable *is a term;*
- *any* constant *in $\mathscr{C}$ is a term;*
- *if u and v are terms, then the* application *uv is a term;*
- *if A and t are terms and x is a variable, then the abstraction $\lambda x : A.t$ is a term.*

Definition 1.3.2 (Free and Bound Variables)**.** *The set of* free variables *of a term t, written $FV(t)$, is defined as follows:*

- $FV(x) = \{x\}$;
- $FV(uv) = FV(u) \cup FV(v)$;
- $FV(\lambda x : A.u) = FV(A) \cup (FV(u) \setminus \{x\})$.

Variables occurring in t that are not free are called bound.

Convention We identify λ-terms that can be obtained from each other by a renaming of bound variables in a way such that free variables do not become bound.

Definition 1.3.3 (Substitution)**.** *Let x be a variable and t, u be two terms. The* substitution *of x by u in t, written $t[x/u]$ is defined as follows:*

- $x[x/u] = u$;
- $y[x/u] = y$, *if* $y \neq x$;
- $(t_1\, t_2)[x/u] = t_1[x/u]\, t_2[x/u]$;
- $(\lambda y : A.t)[x/u] = \lambda y : A[x/u].t[x/u]$, *if* $x \neq y$ *and* $y \notin FV(u)$. *Remark that we can always rename the bound variables so that these conditions are verified.*

Definition 1.3.4 (β-Reduction)**.** *The* head β-reduction $\to_{\beta^h}$ *on λ-terms is defined as follows:*

- $(\lambda x : A.t)\, u \to_{\beta^h} t[x/u]$ *for all λ-terms A, t and u;*

The β-reduction *relation $\to_\beta$ on λ-terms is defined as follows:*

- $t_1 \to_{\beta^h} t_2$;
- *if* $t_1 \to_\beta t_2$, *then* $t_1\, u \to_\beta t_2\, u$ *and* $u\, t_1 \to_\beta u\, t_2$;
- *if* $t_1 \to_\beta t_2$, *then* $\lambda x : A.t_1 \to_\beta \lambda x : A.t_2$ *and* $\lambda x : t_1.u \to_\beta \lambda x : t_2.u$.

We write $\to_{\beta^i}$ for $\to_\beta \setminus \to_{\beta^h}$.

Theorem 1.3.5 (Confluence [CR36])**.** *The relation $\to_\beta$ is confluent.*

The following theorem is an easy consequence of the so-called *standardization theorem* [CF58].

Theorem 1.3.6. *If $t_1 \to_\beta^* t_2$, then there exists t_3 such that $t_1 \to_{\beta^h}^* t_3 \to_{\beta^i}^* t_2$.*

1.4 Combining Term Rewriting Systems and the λ-Calculus

We now study some extensions of the λ-calculus with rewrite rules from TRSs.

1.4.1 Applicative Term Rewriting Systems

The λ-calculus has no notion of arity; therefore to see first-order terms as λ-terms, we need to currify them.

Definition 1.4.1 (Applicative TRS)**.** *A TRS is* applicative *if it is built from a signature $\{\bullet\} \cup \mathscr{C}$ where $\bullet$ is a symbol of arity 2 and the symbols in $\mathscr{C}$ have arity 0.*

Definition 1.4.2 (Curryfication of a TRS)**.** *Let Σ be a signature. Let Σ^{cur} be the signature $\{\bullet\} \cup \mathscr{C}$ where $\mathscr{C}$ contains the constants in Σ but with arity 0.*

The function **cur** *of* curryfication *from terms over the signature Σ to terms over the signature Σ^{cur} is defined as follows:*

$$\begin{array}{ll} \mathbf{cur}(x) & = x \\ \mathbf{cur}(f(t_1,\ldots,t_n)) & = \bullet(\ldots(\bullet(f, \mathbf{cur}(t_1)),\ldots,)\mathbf{cur}(t_n)) \end{array}$$

Let R be a TRS over the signature Σ. The applicative TRS R^{cur} over the signature Σ^{cur} is $\{\, (\mathbf{cur}(l), \mathbf{cur}(r)) \mid (l \hookrightarrow r) \in R \,\}$.

Theorem 1.4.3 (Preservation of Confluence by Curryfication [Ter03])**.** *Let R be a TRS. If R is confluent, then R^{cur} is confluent.*

1.4.2 Combining Term Rewriting Systems and the λ-Calculus

Definition 1.4.4 (From TRS Terms to λ-terms)**.** *Let $\{\bullet\} \cup \mathscr{C}$ be a signature where $\bullet$ is a symbol of arity 2 and the symbols in $\mathscr{C}$ have arity 0.*

The function $|.|_\lambda$ from first-order terms over $\{\bullet\} \cup \mathscr{C}$ to λ-terms over $\mathscr{C}$ is defined as follows:

$$\begin{array}{lll} |x|_\lambda & = x & \text{if } x \text{ is a variable} \\ |c|_\lambda & = c & \text{if } c \in \mathscr{C} \\ |\bullet(t_1, t_2)|_\lambda & = |t_1|_\lambda |t_2|_\lambda & \end{array}$$

Definition 1.4.5 (λR-Calculus)**.** *Let R be an applicative TRS for the signature $\{\bullet\} \cup \mathscr{C}$. The relation $\to_{\beta R}$ on λ-terms is defined as follows:*

- *if $t \to_\beta s$, then $t \to_{\beta R} s$;*
- *if $(l \hookrightarrow r) \in R$ and σ is a substitution, then $\sigma(|l|_\lambda) \to_{\beta R} \sigma(|r|_\lambda)$;*
- *if $t_1 \to_{\beta R} t_2$, then $t_1 u \to_{\beta R} t_2 u$ and $u t_1 \to_{\beta R} u t_2$;*
- *if $t_1 \to_{\beta R} t_2$, then $\lambda x : A.t_1 \to_{\beta R} \lambda x : A.t_2 u$ and $\lambda x : t_1.u \to_{\beta R} \lambda x : t_2.u$.*

1.4.3 Confluence

When the TRS is left-linear, the confluence is preserved when combined with β-reduction, provided that it is non variable-applying.

Definition 1.4.6 (Variable-Applying TRS)**.** *An applicative TRS R is* variable-applying *if, for some $(l, r) \in R$, there is a subterm of l of the form $\bullet(x, t)$ where x is a variable.*

Theorem 1.4.7 (Confluence for Left-Linear Systems [Mül92])**.** *Let R be a left-linear and non-variable-applying TRS.*

If R is confluent, then $\to_{\beta R}$ is confluent.

However, when the TRS is not left-linear, adding the β-reduction breaks the confluence in most cases.

Lemma 1.4.8 (Turing's Ω Combinator)**.** *Let A be an arbitrary type and let $Z = \lambda z : A.\lambda x : A.x\ (z\ z\ x)$. The term $\Omega = Z\ Z$ is a fix-point combinator,* i.e.*, for any term t, $\Omega\ t \to_\beta^* t\ (\Omega\ t)$.*

Proof. $Z\ Z\ t = (\lambda z : A.\lambda x : A.x\ (z\ z\ x))\ Z\ t \to_\beta^2 t\ (Z\ Z\ t)$. □

In the following theorems, we write $t\ u$ instead of $\bullet(t, u)$.

Theorem 1.4.9. *Let $R = \{(\texttt{minus}\ n\ n \hookrightarrow 0), (\texttt{minus}\ (\texttt{S}\ n)\ n \hookrightarrow \texttt{S}\ 0)\}$ where $\texttt{minus}, \texttt{S}$ and 0 are constants. The relation $\to_{\beta R}$ is not confluent.*

Proof. We have the following reductions:

$\texttt{minus}\ (\Omega\texttt{S})\ (\Omega\texttt{S}) \to_{\beta R} 0$.
$\texttt{minus}\ (\Omega\texttt{S})\ (\Omega\texttt{S}) \to_\beta^* \texttt{minus}\ (\texttt{S}\ (\Omega\texttt{S}))\ (\Omega\texttt{S}) \to_{\beta R} \texttt{S}\ 0$.

However 0 and $\texttt{S}\ 0$ are not joinable. □

Theorem 1.4.10 ([Klo80])**.** *Let $R = \{(\texttt{eq}\ n\ n \hookrightarrow \texttt{true})\}$. The relation $\to_{\beta R}$ is not confluent.*

Proof. Let $c = \Omega\ (\lambda x : A.\lambda y : A.\mathtt{eq}\ y\ (x\ y))$ and $a = \Omega\ c$.

We have $a = \Omega\ c \to^*_\beta c\ a$.

Moreover, $c\ a = \Omega\ (\lambda x : A.\lambda y : A.\mathtt{eq}\ y\ (x\ y))\ a \to^*_\beta (\lambda x : A.\lambda y : A.\mathtt{eq}\ y\ (x\ y))\ c\ a \to^*_\beta \mathtt{eq}\ a\ (c\ a) \to^*_\beta \mathtt{eq}\ (c\ a)\ (c\ a) \to_{\beta R} \mathtt{true}$.

Since $c\ a \to^*_\beta c\ (c\ a)$, we also have $c\ a \to^*_{\beta R} c\ \mathtt{true}$.

Therefore, $c\ a \to^*_{\beta R} \mathtt{true}$ and $c\ a \to^*_{\beta R} c\ \mathtt{true}$.

If the relation $\to_{\beta R}$ is confluent, then we have $c\ \mathtt{true} \to_{\beta R} \mathtt{true}$. We now prove that it is impossible (hence, that $\to_{\beta R}$ is not confluent).

Suppose that $c\ \mathtt{true} \to^*_{\beta R} \mathtt{true}$.

Take the shortest reduction sequence $c\ \mathtt{true} \to^*_\beta t \to^*_R \mathtt{true}$ such that the sequence $c\ \mathtt{true} \to^*_\beta t$ is standard (*i.e.*, the β-reduction are made from left to right). Such a sequence exists, by the *standardization theorem* [Ter03] and because the rule of R cannot create β-redexes.

This sequence has the following shape: $c\ \mathtt{true} = \Omega\ (\lambda x : A.\lambda y : A.\mathtt{eq}\ y\ (x\ y))\ \mathtt{true} \to^*_\beta (\lambda x : A.\lambda y : A.\mathtt{eq}\ y\ (x\ y))\ c\ \mathtt{true} \to^*_\beta \mathtt{eq}\ \mathtt{true}\ (c\ \mathtt{true}) \to^*_{\beta R} \mathtt{true}$. Therefore, this sequence must contain a standard sub-sequence $c\ \mathtt{true} \to^*_\beta t \to^*_R \mathtt{true}$ to perform the last reductions. This is impossible by assumption. □

Chapter 2

The $\lambda\Pi$-Calculus Modulo

Résumé Ce chapitre donne une nouvelle présentation du $\lambda\Pi$-Calcul Modulo correspondant au calcul implémenté par DEDUKTI. Cette nouvelle version améliore celle de Cousineau et Dowek de deux manières. D'abord on définit une notion de réécriture sur les termes sans aucune notion de typage. Ceci permet de rendre sa comparaison avec son implémentation plus directe. Ensuite, on explicite et on clarifie le typage des règles de réécriture.

On procède à une étude théorique précise du $\lambda\Pi$-Calcul Modulo. En particulier, on met en exergue les conditions qui assurent que le système de types vérifie des propriétés élémentaires telles que la préservation du typage par réduction ou l'unicité des types. Ces conditions sont la compatibilité du produit et le bon typage des règles de réécriture. Pour finir, on considère une extension du $\lambda\Pi$-Calcul Modulo avec du polymorphisme et des opérateurs de type que l'on appelle le Calcul des Constructions Modulo.

2.1 Introduction

The $\lambda\Pi$-Calculus Modulo has been introduced by Cousineau and Dowek [CD07] as an extension of the $\lambda\Pi$-Calculus (the dependently typed λ-calculus) meant to express the proofs of Deduction Modulo [DHK03]. This extension features a generalized conversion rule where the congruence is extended to take into account user-defined rewrite rules. Types are not identified modulo β-conversion but modulo βR-conversion where R is a set of rewrite rules. They show that the resulting calculus, although very simple, is a very expressive logical framework [HHP93]. It can embed, in a shallow way, that is in a way that preserves their computational content, many logics and calculus such as: functional Pure Type Systems [CD07], First-Order Logic [Dor11], Higher-Order Logic [AB14], the Calculus of Inductive Constructions [BB12], resolution and superposition proofs [Bur13], or the ς-calculus [CD15].

The original presentation of the $\lambda\Pi$-Calculus Modulo did not include any theoretical study of the calculus and gave a restricted version of it, the article being concerned on how to encode functional pure type systems in it.

In this chapter we give a new presentation of the $\lambda\Pi$-Calculus Modulo and we study its properties in details. This presentation differs from the original one by several aspects. Firstly, it is build upon a completely untyped notion of rewriting. Secondly, we make explicit the typing of rewrite rules and we make it iterative: rewrite

x, y, z	$\in$	$\mathscr{V}_O$	(Object Variable)
$\underline{c}, \underline{f}$	$\in$	$\mathscr{C}_O$	(Object Constant)
C, F	$\in$	$\mathscr{C}_T$	(Type Constant)
$\underline{t}, \underline{u}, \underline{v}$	::=	$x \mid \underline{c} \mid \underline{u}\ \underline{v} \mid \lambda x : \underline{U}.\underline{t}$	(Object)
$\underline{T}, \underline{U}, \underline{V}$	::=	$C \mid \underline{U}\ \underline{v} \mid \lambda x : \underline{U}.\underline{T} \mid \Pi x : \underline{U}.\underline{T}$	(Type)
K	::=	$\mathbf{Type} \mid \Pi x : \underline{U}.K$	(Kind)
t, u, v	::=	$\underline{u} \mid \underline{U} \mid K \mid \mathbf{Kind}$	(Term)

Figure 2.1: Terms of the $\lambda\Pi$-Calculus Modulo

rules previously added are taken in account when typing new ones. All these modifications make our presentation closer to its implementation in DEDUKTI.

2.2 Terms, Contexts and Rewrite Rules

We start by defining the basic elements of our calculus and giving their syntax.

2.2.1 Terms

The terms of the $\lambda\Pi$-Calculus Modulo are the same as for the $\lambda\Pi$-Calculus.

Definition 2.2.1 (Terms)**.** *An* object *is either a* variable *in the set $\mathscr{V}_O$, or an* object constant *in the set $\mathscr{C}_O$, or an* application $\underline{u}\ \underline{v}$ *where $\underline{u}$ and $\underline{v}$ are objects, or an* abstraction $\lambda x : \underline{U}.\underline{t}$ *where $\underline{t}$ is an object and $\underline{U}$ is a type.*

A type *is either a* type constant *in the set $\mathscr{C}_T$, or an* application $\underline{U}\ \underline{v}$ *where $\underline{U}$ is a type and $\underline{v}$ is an object, or an* abstraction $\lambda x : \underline{U}.\underline{V}$ *where $\underline{U}$ and $\underline{V}$ are types, or a* product $\Pi x : \underline{U}.\underline{V}$ *where $\underline{U}$ and $\underline{V}$ are types.*

A kind *is either a* product $\Pi x : \underline{U}.K$ *where $\underline{U}$ is a type and K is a kind or the symbol* **Type**.

A term *is either an object, a type, a kind or the symbol* **Kind**.

We write Λ for the set of terms.

The sets $\mathscr{V}_O$, $\mathscr{C}_O$ and $\mathscr{C}_T$ are assumed to be infinite and pairwise disjoint. The grammars for objects, types, kinds and terms are given Figure 2.1.

We have chosen a syntactic presentation of terms that enforces the distinction between objects, types and kinds. Another approach would be to define these categories by typing: kinds are terms whose type is **Kind**; types have type **Type** and objects are terms whose type is a type. The benefit of the syntactic approach over the typed approach is that it will allow us to ensure syntactically that rewriting preserves this stratification. This will later simplify the theory of the $\lambda\Pi$-Calculus Modulo, in particular when confluence is not known.

Notation 2.2.2. *In addition with the naming convention of Figure 2.1, we use:*

- *c, f to denote object or type constants;*
- *A, B, T, U, V to denote types, kinds or* **Kind***;*

$$\Delta \quad ::= \quad \emptyset \mid \Delta(\underline{x} : \underline{T}) \qquad \text{(Local Context)}$$

Figure 2.2: Local contexts of the $\lambda\Pi$-Calculus Modulo

$$\begin{array}{lcll} R & ::= & (\underline{u} \hookrightarrow \underline{v}) \mid (\underline{U} \hookrightarrow \underline{V}) & \text{(Rewrite Rule)} \\ \Xi & ::= & R \mid \Xi R & \text{(Batch of Rewrite Rules)} \\ \Gamma & ::= & \emptyset \mid \Gamma(\underline{c} : \underline{T}) \mid \Gamma(C : K) \mid \Gamma\Xi & \text{(Global Context)} \end{array}$$

Figure 2.3: Global contexts of the $\lambda\Pi$-Calculus Modulo

- *s for* **Type** *or* **Kind**.

Moreover we write $t\vec{u}$ to denote the application of t to an arbitrary number of arguments $u_1, \ldots, u_n$. Terms are identified up to renaming of bound variables (α-equivalence). We write $u[x/v]$ for the usual (capture avoiding) substitution of x by v in u. We write $A \to B$ for $\Pi x : A.B$ when B does not depend on x. If t is a term, we write $FV(t)$ (respectively $BV(t)$) for the set of free (respectively bound) variables of t. By convention, we assume that the sets of bound and free variables in a terms are always disjoint. This can always be obtained by an appropriate renaming of bound variables.

2.2.2 Local Contexts

As in the $\lambda\Pi$-Calculus, local contexts consist of typing declarations for variables.

Definition 2.2.3 (Local Context)**.** *A local context is a list of pairs made of an object variable together with a type. The grammar for local contexts is given Figure 2.2.*

Notation 2.2.4. *We write $dom(\Delta)$ for the set $\{x \in \mathcal{V} \mid (x : A) \in \Delta\}$. If $(x, A) \in \Delta$, we sometimes write $\Delta(x)$ for A. We write $\Delta_1 \subset \Delta_2$ if Δ_1 is a prefix of Δ_2.*

2.2.3 Rewrite Rules and Global Contexts

We now define rewrite rules and global contexts, two distinctive features of the $\lambda\Pi$-Calculus Modulo with respect to the $\lambda\Pi$-Calculus.

Definition 2.2.5 (Rewrite Rules)**.** *An object-level rewrite rule is a pair of objects. A type-level rewrite rule is a pair of types. A rewrite rule is either an object-level rewrite rule or a type-level rewrite rule.*

This definition is different from the one in [CD07]. There, a rewrite rule is a quadruple made of two terms, a typing context and a type. The reason is that we do not use exactly the same notion of rewriting (see Definition 2.3.3).

A global context contains typing declaration for constants. They can also contain rewrite rules.

Definition 2.2.6 (Global Contexts)**.** *A global context is a list of pairs formed by an object constant with a type, pairs formed by a type constant with a kind and lists of rewrite rules. The grammar for global contexts is given Figure 2.3.*

Remark that the rewrite rules are not added one by one in the context but by groups. This will have its importance for the type system (see Remark 2.4.11).

Notation 2.2.7. *As for local contexts, we write* $\Gamma_1 \subset \Gamma_2$ *if* Γ_1 *is a prefix of* Γ_2*. We write* $dom(\Gamma)$ *for the set* $\{c \in \Gamma | (c : A) \in \Gamma\}$*. If* $(c, A) \in \Gamma$*, we sometimes write* $\Gamma(c)$ *for* A*.*

The distinction between local and global contexts, as well as the presence of rewrite rules in the global context, are distinctive features of our presentation of the $\lambda\Pi$-Calculus Modulo with respect to the original one. First, Cousineau and Dowek do not distinguish between variables and constants (there are only variables). Second, we want to make explicit the role of the rewrite rules to be able to *dynamically* add them in a type-safe manner.

2.3 Rewriting

In the $\lambda\Pi$-Calculus Modulo we distinguish two kinds of rewriting.

2.3.1 β-Reduction

The first kind of rewriting is β-reduction, which is defined as usual.

Definition 2.3.1 (β-reduction)**.** *The* β*-reduction relation* $\rightarrow_\beta$ *is the smallest relation on terms containing* $(\lambda x : A.u)\, v \rightarrow_\beta u[x/v]$ *for any* A, u *and* v *and closed by subterm reduction.*

Notation 2.3.2. *We write* $\rightarrow_\beta^*$ *for the reflexive and transitive closure of* $\rightarrow_\beta$ *and* $\equiv_\beta$ *for the congruence generated by* $\rightarrow_\beta$*.*

2.3.2 Γ-Reduction

The second kind of rewriting is Γ-reduction, the relation generated by the rewrite rules of a global context Γ.

Definition 2.3.3 (Γ-Reduction)**.** *Let* Γ *be a global context. The* Γ*-reduction relation* $\rightarrow_\Gamma$ *is the smallest relation on terms containing* $u \rightarrow_\Gamma v$ *for each rule* $(u \hookrightarrow v) \in \Gamma$ *and closed by substitution and subterm reduction.*

Notation 2.3.4. *We write* $\rightarrow_\Gamma^*$ *for the reflexive and transitive closure of* $\rightarrow_\Gamma$*,* $\equiv_\Gamma$ *for the congruence generated by* $\rightarrow_\Gamma$*,* $\rightarrow_{\beta\Gamma}$ *for* $\rightarrow_\beta \cup \rightarrow_\Gamma$*,* $\rightarrow_{\beta\Gamma}^*$ *for the reflexive and transitive closure of* $\rightarrow_{\beta\Gamma}$ *and* $\equiv_{\beta\Gamma}$ *for the equivalence relation generated by* $\rightarrow_{\beta\Gamma}$*.*

As already mentioned, our notion of rewriting is different from the one in [CD07]. In the original presentation a rewrite rule is a quadruple (Δ, l, r, T) where Δ is a context, l and r are, respectively, the left-hand side and the right-hand side and T is their common type within the context Δ. Rewriting is defined as follows: if σ is a well-typed substitution (Definition 2.4.12) from Δ to Δ_2, then $\sigma(l)$ rewrites to $\sigma(r)$ in the context Δ_2. Anticipating a bit, this means that rewriting and typing are two mutually defined notions. Indeed, typing depends on rewriting through the conversion rule which says that we can change the type of terms if the two types are convertible.

On the contrary, our notion of rewriting does not depend on typing. Therefore, our definition is simpler. Moreover, untyped rewriting is what is usually implemented in type checkers such as DEDUKTI. It would be completely inefficient

to check the well-typedness of the substitution at each reduction step. Therefore proving the correspondence between the $\lambda\Pi$-Calculus Modulo and DEDUKTI will be easier. The original presentation of $\lambda\Pi$-Calculus Modulo is, in this respect, closely related to Martin Löf Logical Framework [NPS90].

Of course, untyped reduction brings its own difficulties; in particular the proof of subject reduction (Theorem 2.6.22) will require additional hypotheses.

We can now prove our first lemma about rewriting, which states that rewriting respects syntactical categories.

Lemma 2.3.5 (Stratification of the Conversion)**.** *Let Γ be a global context.*

- *If* **Kind** $\equiv_{\beta\Gamma} t$*, then* $t =$ **Kind**.
- *If* **Type** $\equiv_{\beta\Gamma} t$*, then* $t =$ **Type**.
- *If* $\Pi x : \underline{T_1}.K_1 \equiv_{\beta\Gamma} t$*, then* $t = \Pi x : \underline{T_2}.K_2$*,* $\underline{T_1} \equiv_{\beta\Gamma} \underline{T_2}$ *and* $K_1 \equiv_{\beta\Gamma} K_2$*.*
- *If* $u \equiv_{\beta\Gamma} v$ *and* u *is an object (respectively a type), then* v *is an object (respectively a type).*

Proof.

- By definition of rewriting, no term reduces to **Kind** or **Type**.
- Note that since $\Pi x : \underline{T_1}.K_1$ is a kind, if $t \to_{\beta\Gamma} \Pi x : \underline{T_1}.K_1$, then $t = \Pi x : \underline{T_2}.K_1$ or $t = \Pi x : \underline{T_1}.K_2$ with $\underline{T_1} \to_{\beta\Gamma} \underline{T_2}$ or $K_1 \to_{\beta\Gamma} K_2$ respectively.
- β-reduction and Γ-reduction preserve objects, types and kinds.

□

Definition 2.3.6 (Convertible Local Contexts)**.** *Let Γ be a global context. Two local contexts Δ_1 and Δ_2 are* convertible *in Γ if they declare the same variables in the same order and, for all $x \in dom(\Delta_1) = dom(\Delta_2)$, $\Delta_1(x) \equiv_{\beta\Gamma} \Delta_2(x)$.*

We write $\Delta_1 \equiv_{\beta\Gamma} \Delta_2$ if Δ_1 and Δ_2 are convertible in Γ.

2.4 Type System

We now give the typing rules of the $\lambda\Pi$-Calculus Modulo. We start by the typing rules for terms; then we proceed with the typing rules for local contexts. Finally, we discuss well-typedness for global contexts.

2.4.1 Terms

Definition 2.4.1 (Well-Typed Term)**.** *We say that a term t has type A in global context Γ and local context Δ if the judgment $\Gamma; \Delta \vdash t : A$ is derivable by the inference rules of Figure 2.4. We say that a term is* well-typed *if such an A exists.*

Remark 2.4.2. *The only difference with the $\lambda\Pi$-Calculus is the replacement, in the* **(Conversion)** *rule, of $\equiv_\beta$ by the extended congruence $\equiv_{\beta\Gamma}$. The congruence now depends on the global context.*

(**Sort**) $$\frac{}{\Gamma;\Delta \vdash \mathbf{Type} : \mathbf{Kind}}$$

(**Variable**) $$\frac{(x : A) \in \Delta}{\Gamma;\Delta \vdash x : A}$$

(**Constant**) $$\frac{(c : A) \in \Gamma}{\Gamma;\Delta \vdash c : A}$$

(**Application**) $$\frac{\Gamma;\Delta \vdash t : \Pi x : A.B \qquad \Gamma;\Delta \vdash u : A}{\Gamma;\Delta \vdash tu : B[x/u]}$$

(**Abstraction**) $$\frac{\Gamma;\Delta(x : A) \vdash t : B \qquad \Gamma;\Delta \vdash \Pi x : A.B : s}{\Gamma;\Delta \vdash \lambda x : A.t : \Pi x : A.B}$$

(**Product**) $$\frac{\Gamma;\Delta \vdash A : \mathbf{Type} \qquad \Gamma;\Delta(x : A) \vdash B : s}{\Gamma;\Delta \vdash \Pi x : A.B : s}$$

(**Conversion**) $$\frac{\Gamma;\Delta \vdash t : A \qquad \Gamma;\Delta \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma;\Delta \vdash t : B}$$

Figure 2.4: Typing rules for terms in the $\lambda\Pi$-Calculus Modulo.

(Empty Local Context) $$\frac{}{\Gamma \vdash^{ctx} \emptyset}$$

(Variable Declaration) $$\frac{\Gamma \vdash^{ctx} \Delta \qquad \Gamma;\Delta \vdash \underline{U} : \mathbf{Type} \qquad x \notin dom(\Delta)}{\Gamma \vdash^{ctx} \Delta(x : \underline{U})}$$

Figure 2.5: Well-formedness rules for local contexts

2.4.2 Local Contexts

For local contexts, the typing rules basically ensure that type declarations are well-typed. The rules are the same as for the $\lambda\Pi$-Calculus.

Definition 2.4.3 (Well-Formed Local Context)**.** *A local context* Δ *is* well-formed *with respect to a global context* Γ *if the judgment* $\Gamma \vdash^{ctx} \Delta$ *is derivable by the inference rules of Figure 2.5.*

2.4.3 Global Contexts

Besides the new conversion relation, the main difference between the $\lambda\Pi$-calculus and the $\lambda\Pi$-Calculus Modulo is the presence of rewrite rules in global contexts. We need to take this into account when typing global contexts.

A key feature of any type system is the preservation of typing by reduction: the subject reduction property.

Definition 2.4.4 (Subject Reduction)**.** *Let* $\rightarrow_r$ *be a relation on terms. We say that a global context* Γ *satisfies the* subject reduction *property for* $\rightarrow_r$ *if the following proposition is verified.*

For any local context Δ *well-formed for* Γ*, terms* t_1 *and* t_2 *such that* $t_1 \rightarrow_r t_2$ *and term* T*, if* $\Gamma;\Delta \vdash t_1 : T$ *then* $\Gamma;\Delta \vdash t_2 : T$*.*

We write $\mathbf{SR}^r(\Gamma)$ *if* Γ *satisfies the subject reduction property for* $\rightarrow_r$*.*

In the $\lambda\Pi$-Calculus Modulo, we cannot allow adding arbitrary rewrite rules in the context if we want to preserve subject reduction for $\rightarrow_{\beta\Gamma}$.

In particular, as we will see, subject reduction for the β-reduction requires the following property to hold:

Definition 2.4.5 (Product Compatibility)**.** *We say that a global context* Γ *satisfies the* product compatibility *property if the following proposition is verified for any* A_1, A_2, B_1, B_2 *and* Δ*:*

If $\Gamma \vdash^{ctx} \Delta$ *and* $\Gamma;\Delta \vdash \Pi x : A_1.B_1 : s$ *and* $\Gamma;\Delta \vdash \Pi x : A_2.B_2 : s$ *and* $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$*, then we have* $A_1 \equiv_{\beta\Gamma} A_2$ *and* $B_1 \equiv_{\beta\Gamma} B_2$*.*

We write $\mathbf{PC}(\Gamma)$ *when product compatibility holds for* Γ*.*

On the other hand, subject reduction for Γ-reduction requires rewrite rules to be well-typed in the following sense:

Definition 2.4.6 (Well Typed Rewrite Rules)**.** *A rewrite rule* $(u \hookrightarrow v)$ *is* well-typed *for a global context* Γ *if, for any substitution* σ*, any well-formed local context* Δ *and any term* T*, if* $\Gamma;\Delta \vdash \sigma(u) : T$*, then* $\Gamma;\Delta \vdash \sigma(v) : T$*.*

We write $\Gamma \vdash u \hookrightarrow v$ *if* $(u \hookrightarrow v)$ *is well-typed in* Γ*.*

The simplest way to show that a rewrite rule is well-typed is to show that it is strongly well-formed.

Definition 2.4.7 (Algebraic Term)**.** *A term is* algebraic *if it is built from constants, variables and applications, variables do not have arguments and it is not a variable.*

Definition 2.4.8 (Strongly Well-Formed Rewrite Rule)**.** *Let* Γ *be a global context such that* $\rightarrow_{\beta\Gamma}$ *is confluent. A rewrite rule* $(u \hookrightarrow v)$ *is* strongly-well-formed *in* Γ *if* u *is algebraic and, for some local context* Δ *and term* T*, we have:*

- $dom(\Delta) = FV(u)$,
- $\Gamma \vdash^{ctx} \Delta$,
- $\Gamma; \Delta \vdash u : T$ *and*
- $\Gamma; \Delta \vdash v : T$

We write $\Gamma \vdash^{str} u \hookrightarrow v$ *if* $(u \hookrightarrow v)$ *is strongly well-formed in* Γ.

We can now define the notion of well-typed global context. Contrary to terms and local contexts, we give an axiomatic definition and not an inductive definition based on inference rules. The reason is that there is not *one* obvious set of inference rules for this notion. In fact, in this thesis, we define different notions of well-formed global contexts based on different sets of inference rules. Each time, we prove that well-formed global contexts are well-typed.

Definition 2.4.9 (Well-Typed Global Contexts)**.** *A global context* Γ *is* well-typed *if:*

- **(Well-Typed Declarations)** *for all* $(c : T) \in \Gamma$, *we have* $\Gamma; \emptyset \vdash T : s$, *for some sort* s*;*
- **(Product Compatibility)** Γ *satisfies the product compatibility property;*
- **(Well-Typed Rewrite Rules)** *for all* $(u \hookrightarrow v) \in \Gamma$, *we have* $\Gamma \vdash u \hookrightarrow v$.

It is not obvious how to check that product compatibility and well-typedness of rewrite rules hold. In fact they are undecidable properties (Section 2.6.6 and Section 3.7.5) and an important part of this thesis is dedicated to finding sufficient criteria for them to hold. For instance criteria for typing of rewrite rules are given in Chapter 3 and criteria for product compatibility are given in Chapter 4 and Chapter 5.

We now give a first notion of well-formed global context using inference rules: strongly well-formed global contexts.

Definition 2.4.10 (Strongly Well-Formed Global Context)**.** *A global context is* strongly well-formed *if the judgment* Γ **swf** *is derivable by the inference rules of Figure 2.6.*

In Section 2.6.3, we will prove that strongly well-formed global contexts are well-typed.

Remark 2.4.11. *The rule* **(Rewrite Rules)** *allows adding several rewrite rules at once. On the one hand, only the confluence of the whole system is required (and not the confluence after adding each rewrite rule). On the other hand, the rewrite rules must be shown strongly well-typed independently from the other rules added at the same time (*i.e. *in* Γ *and not in* $\Gamma\Xi$*).*

2.4.4 Substitutions

Definition 2.4.12 (Well-typed substitution)**.** *A substitution* σ *is* well-typed *from* Δ_1 *to* Δ_2 *in* Γ, *written* $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta_2$ *if, for all* $x \in dom(\Delta_1)$, *we have* $\Gamma; \Delta_2 \vdash \sigma(x) : \sigma(\Delta_1(x))$.

2.5 Examples

We give simple examples of strongly well-formed global contexts.

(Empty Global Context)

$$\frac{}{\emptyset \ \textbf{swf}}$$

(Object Declaration)

$$\frac{\Gamma \ \textbf{swf} \qquad \Gamma;\emptyset \vdash \underline{U} : \textbf{Type} \qquad \underline{c} \notin dom(\Gamma)}{\Gamma(\underline{c} : \underline{U}) \ \textbf{swf}}$$

(Type Declaration)

$$\frac{\Gamma \ \textbf{swf} \qquad \Gamma;\emptyset \vdash K : \textbf{Kind} \qquad C \notin dom(\Gamma)}{\Gamma(C : K) \ \textbf{swf}}$$

(Rewrite Rules)

$$\frac{\Gamma \ \textbf{swf} \qquad \rightarrow_{\beta\Gamma\Xi} \text{ is confluent} \qquad \begin{array}{c}(\forall i)\Gamma \vdash^{str} u_i \hookrightarrow v_i \\ \Xi = (u_1 \hookrightarrow v_1)\dots(u_n \hookrightarrow v_n)\end{array}}{\Gamma\Xi \ \textbf{swf}}$$

Figure 2.6: Strong well-formedness rules for global contexts

2.5.1 Arithmetic Operations on Peano Integers

Addition and multiplication on Peano integers are easily expressed in the $\lambda\Pi$-Calculus Modulo. We use parentheses in terms in the usual way.

We begin by the definition of Peano integers using three declarations of constants.

$\texttt{nat} : \textbf{Type}.$
$0 : \texttt{nat}.$
$\texttt{S} : \texttt{nat} \longrightarrow \texttt{nat}.$

For readability, we will write k instead $\overbrace{\texttt{S}\ (\texttt{S}\ \dots\ (\texttt{S}}^{k \text{ times}}\ 0))$.
We now declare a constant for the addition and we add rewrite rules related to it. We use italics to distinguish variables.

$\texttt{plus} : \texttt{nat} \longrightarrow \texttt{nat} \longrightarrow \texttt{nat}.$
$\texttt{plus}\ 0\ n \hookrightarrow n.$
$\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

We do the same thing for the multiplication.

$\texttt{mult} : \texttt{nat} \longrightarrow \texttt{nat} \longrightarrow \texttt{nat}.$
$\texttt{mult}\ 0\ n \hookrightarrow 0.$
$\texttt{mult}\ (S\ n_1)\ n_2 \hookrightarrow \texttt{plus}\ n_2\ (\texttt{mult}\ n_1\ n_2).$

Then we add a weak form of equality (only convertible terms are equal).

$\texttt{eq} : \texttt{nat} \longrightarrow \texttt{nat} \longrightarrow \textbf{Type}.$
$\texttt{refl} : \Pi n : \texttt{nat}.\texttt{eq}\ n\ n.$

The term $\texttt{refl}\ 4$ has type $\texttt{eq}\ (\texttt{plus}\ 2\ 2)\ 4$. We have just proved that $2+2=4$!

The rewrite system we have defined so far is orthogonal (it is left-linear and there

are no critical pairs between the rewrite rules); this is not mandatory. For instance, it is possible to complete the definition of `plus` with the following rewrite rules that are symmetric to the previous ones:

$\texttt{plus}\ n\ 0 \hookrightarrow n.$
$\texttt{plus}\ n_1\ (\texttt{S}\ n_2) \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

Other rewrite rules we might want to consider are rules for associativity and commutativity of the addition:

$\texttt{plus}\ n_1\ (\texttt{plus}\ n_2\ n_3) \hookrightarrow \texttt{plus}\ (\texttt{plus}\ n_1\ n_2)\ n_3.$
$\texttt{plus}\ n_1\ n_2 \hookrightarrow \texttt{plus}\ n_2\ n_1.$

By 1.4.7, the rewrite system, although not orthogonal, is still confluent; therefore the context is still strongly well-formed. However, the last rewrite rule is obviously not terminating. We usually do not consider non-terminating rewrite rules as they make type checking undecidable.

2.5.2 The Map Function on Lists

Let us define the type of lists of integers.

$\texttt{list} : \textbf{Type}.$
$\texttt{nil} : \texttt{list}.$
$\texttt{cons} : \texttt{nat} \longrightarrow \texttt{list} \longrightarrow \texttt{list}.$

The function `map` allows applying a function to every element of a list.

$\texttt{map} : (\texttt{nat} \longrightarrow \texttt{nat}) \longrightarrow \texttt{list} \longrightarrow \texttt{list}.$
$\texttt{map}\ f\ \texttt{nil} \hookrightarrow \texttt{nil}.$
$\texttt{map}\ f\ (\texttt{cons}\ hd\ tl) \hookrightarrow \texttt{cons}\ (f\ hd)\ (\texttt{map}\ f\ tl).$

For instance, we can use this function to increment the elements of a list by a constant.

$$\texttt{map}\ (\texttt{plus}\ 3)\ (\texttt{cons}\ 1\ (\texttt{cons}\ 2\ (\texttt{cons}\ 3\ \texttt{nil}))) \rightarrow^*_{\beta\Gamma} \texttt{cons}\ 4\ (\texttt{cons}\ 5\ (\texttt{cons}\ 6\ \texttt{nil}))$$

2.5.3 Addition on Brouwer's Ordinals

We now define addition on Brouwer's ordinals:

$\texttt{ord} : \textbf{Type}.$
$\texttt{o\_0} : \texttt{ord}.$
$\texttt{o\_S} : \texttt{ord} \longrightarrow \texttt{ord}.$
$\texttt{lim} : (\texttt{nat} \longrightarrow \texttt{ord}) \longrightarrow \texttt{ord}.$

$\texttt{o\_plus} : \texttt{ord} \longrightarrow \texttt{ord} \longrightarrow \texttt{ord}.$
$\texttt{o\_plus}\ \texttt{o\_0}\ x \hookrightarrow x.$
$\texttt{o\_plus}\ (\texttt{o\_S}\ x)\ y \hookrightarrow \texttt{o\_S}\ (\texttt{o\_plus}\ x\ y).$
$\texttt{o\_plus}\ (\texttt{lim}\ f)\ y \hookrightarrow \texttt{lim}\ (\lambda x{:}\texttt{nat}.\texttt{o\_plus}\ (f\ x)\ y).$

2.6 Properties

In this section we give some meta-theoretical results about the $\lambda\Pi$-Calculus Modulo. We start by basic lemmas, and then we prove more interesting properties about

well-typed and strongly well-formed contexts, subject reduction and uniqueness of types. Then, we show the undecidability of subject reduction and uniqueness of types.

2.6.1 Basic Properties

This subsection gathers technical lemmas about the $\lambda\Pi$-Calculus Modulo.

Lemma 2.6.1 (Inversion)**.** *If* $\Gamma;\Delta \vdash t : T$ *then*

- *either* $t =$ **Type** *and* $T =$ **Kind***;*
- *or* $t = x$ *and there exists* A *such that* $(x : A) \in \Delta$ *and* $T \equiv_{\beta\Gamma} A$*;*
- *or* $t = c$ *and there exists* A *such that* $(c : A) \in \Gamma$ *and* $T \equiv_{\beta\Gamma} A$*;*
- *or* $t = f\,u$ *and there exist* A *and* B *such that* $\Gamma;\Delta \vdash f : \Pi x : A.B$ *and* $\Gamma;\Delta \vdash u : A$ *and* $T \equiv_{\beta\Gamma} B[x/u]$*;*
- *or* $t = \lambda x : A.t$ *and there exist* B *and a sort* s *such that* $\Gamma;\Delta \vdash \Pi x : A.B : s$ *and* $\Gamma;\Delta(x : A) \vdash t : B$ *and* $T \equiv_{\beta\Gamma} \Pi x : A.B$*;*
- *or* $t = \Pi x : A.B$ *and there exists a sort* s *such that* $\Gamma;\Delta \vdash A :$ **Type** *and* $\Gamma;\Delta(x : A) \vdash B : s$ *and* $T = s$*.*

Proof. By induction on the typing derivation and Lemma 2.3.5. □

Lemma 2.6.2. *No term containing* **Kind** *is typable.*

Proof. By induction on the typing derivation, we have that, if t typable, then **Kind** $\notin t$. □

Lemma 2.6.3. *Every sub-term of a well-typed term is well-typed.*

Proof. By induction on the typing derivation. □

Lemma 2.6.4 (Local Weakening)**.** *Let* Δ_1 *and* Δ_2 *be two local contexts. Assume that* Δ_1 *is a subset of* Δ_2*.*
If $\Gamma;\Delta_1 \vdash t : T$*, then* $\Gamma;\Delta_2 \vdash t : T$*.*

Proof. By induction on the typing derivation. □

Lemma 2.6.5 (Global Weakening)**.** *If* $\Gamma;\Delta \vdash t : T$ *and* $\Gamma_2 \supset \Gamma$*, then* $\Gamma_2;\Delta \vdash t : T$*.*

Proof. By induction on the typing derivation. □

Lemma 2.6.6 (Inversion for $\vdash^{ctx}$)**.** *If* $\Gamma \vdash^{ctx} \Delta(x : A)$*, then* $\Gamma \vdash^{ctx} \Delta$ *and* $\Gamma;\Delta \vdash A :$ **Type**.

Proof. By induction on the typing derivation. □

Lemma 2.6.7 (Well-Typed Local Declaration)**.** *If* $\Gamma \vdash^{ctx} \Delta$ *and* $(x : A) \in \Delta$*, then* $\Gamma;\Delta \vdash A :$ **Type**.

Proof. By induction on the typing derivation and local weakening (Lemma 2.6.4). □

Lemma 2.6.8. *Let Γ a global context. Suppose that Δ_1 and Δ_2 are two local contexts such that $\Delta_1 \equiv_{\beta\Gamma} \Delta_2$ and $\Gamma \vdash^{ctx} \Delta_1$.*
If $\Gamma;\Delta_1 \vdash t : T$, then $\Gamma;\Delta_2 \vdash t : T$.

Proof. We prove that, for all local context Σ, if $\Gamma;\Delta_1\Sigma \vdash t : T$, then $\Gamma;\Delta_2\Sigma \vdash t : T$. We proceed by induction on the pair (Δ_1, derivation of $\Gamma;\Delta_1\Sigma \vdash t : T$) ordered lexicographically.

- Cases **(Sort)** and **(Constant)**. Trivial.
- Case **(Variable)**:
  - if $t = x \in dom(\Sigma)$ then $\Gamma;\Delta_2\Sigma \vdash t : T$;
  - if $t = x \in dom(\Delta_1)$ then $(x : T) \in \Delta_1$ and, therefore $T \equiv_{\beta\Gamma} T_2$, for some T_2 such that $(x : T_2) \in \Delta_2$. Since Δ_1 is well-formed, there is a well-formed (strict) prefix Ξ_1 of Δ_1 such that $\Gamma;\Xi_1 \vdash T :$ **Type**. By induction hypothesis on Ξ_1, we have $\Gamma;\Xi_2 \vdash T :$ **Type**, where Ξ_2 is the prefix of Δ_2 such that $\Xi_1 \equiv_{\beta\Gamma} \Xi_2$. By local weakening (Lemma 2.6.4), $\Gamma;\Delta_2 \vdash T :$ **Type**. It follows, by the conversion rule, that $\Gamma;\Delta_2 \vdash x : T$.
- Cases **(Application)**, **(Abstraction)**, **(Product)**, **(Conversion)**. By induction hypothesis.

□

Lemma 2.6.9. *Let Γ be a global context whose declarations are closed.*
If $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta_2$ and $\Gamma;\Delta_1 \vdash t : T$, then $\Gamma;\Delta_2 \vdash \sigma(t) : \sigma(T)$.

Proof. By induction on the typing derivation.

- **(Sort)** Trivial.
- **(Variable)** By hypothesis.
- **(Constant)** Trivial, since global declarations are closed.
- **(Application)**, **(Abstraction)**, **(Product)**, **(Conversion)** By induction hypothesis.

□

Lemma 2.6.10 (Stratification)**.** *Let Γ be a global context whose declarations are well-typed.*
If $\Gamma \vdash^{ctx} \Delta$ and $\Gamma;\Delta \vdash t : T$, then

- *either t is an object, T is a type and $\Gamma;\Delta \vdash T :$ **Type***
- *or t is a type, T is a kind and $\Gamma;\Delta \vdash T :$ **Kind***
- *or t is a kind and $T =$ **Kind**.*

Proof. By induction on the typing derivation.

- **(Sort)** Trivial.
- **(Variable)** By well-typedness of local declarations (Lemma 2.6.7).

- **(Constant)** By hypothesis.
- **(Application)** If $t = uv$, then $T = B[x \backslash v]$, $\Gamma;\Delta \vdash u : \Pi x : A.B$ and $\Gamma;\Delta \vdash v : A$.

  If t is an object, then u is an object and, by induction hypothesis, $\Pi x : A.B$ is a type and $\Gamma;\Delta \vdash \Pi x : A.B : \mathbf{Type}$. It follows that B and $B[x/u]$ are also types. By inversion (Lemma 2.6.1), $\Gamma;\Delta(x : A) \vdash B : \mathbf{Type}$ and $\Gamma;\Delta \vdash A : \mathbf{Type}$. Finally, by the property of well-typed substitutions (Lemma 2.6.9), $\Gamma;\Delta \vdash B[x/v] : \mathbf{Type}$.

  If t is a type, then u is a type and a similar reasoning applies.
- **(Abstraction)** If $t = \lambda x : A.u$, then $T = \Pi x : A.B$, $\Gamma;\Delta(x : A) \vdash u : B$ and $\Gamma;\Delta \vdash \Pi x : A.B : s$.

  If t is an object, then u is an object and, by induction hypothesis, B is a type. It follows that $\Pi x : A.B$ is a type and, by induction hypothesis, $\Gamma;\Delta \vdash s : \mathbf{Kind}$. Thus, $s = \mathbf{Type}$.

  If t is a type, then u is a type and, by induction hypothesis, B is a kind. It follows that $\Pi x : A.B$ is a kind and, by induction hypothesis, $s = \mathbf{Kind}$.
- **(Product)** If $t = \Pi x : A.B$, then $T = s$, $\Gamma;\Delta \vdash A : \mathbf{Type}$ and $\Gamma;\Delta(x : A) \vdash B : s$.

  If B is a type (respectively kind), then $\Pi x : A.B$ is a type (respectively kind) and, by induction hypothesis, $s = \mathbf{Type}$ (respectively $s = \mathbf{Kind}$).
- **(Conversion)** If $\Gamma;\Delta \vdash t : A$, then $\Gamma;\Delta \vdash T : s$ and $A \equiv_{\beta\Gamma} T$.

  If t is an object, then by induction hypothesis, A is a type and $\Gamma;\Delta \vdash A : \mathbf{Type}$. Since $A \equiv_{\beta\Gamma} T$, by Lemma 2.3.5, T is also a type. Therefore, by induction hypothesis, s is a kind. Hence, $s = \mathbf{Type}$.

  If t is a type, then by induction hypothesis, A is a kind and $\Gamma;\Delta \vdash A : \mathbf{Kind}$. Since $A \equiv_{\beta\Gamma} T$, by Lemma 2.3.5, T is also a kind. Therefore, by induction hypothesis, $s = \mathbf{Kind}$.

  If t is a kind, then by induction hypothesis, $A = \mathbf{Kind}$. Since $A \equiv_{\beta\Gamma} T$, by Lemma 2.3.5, $T = \mathbf{Kind}$.

□

2.6.2 Product Compatibility

The usual way to prove product compatibility is by showing the confluence of the rewrite system.

Theorem 2.6.11 (Product Compatibility from Confluence)**.** *Let Γ be a global context. If $\rightarrow_{\beta\Gamma}$ is confluent, then product compatibility holds for Γ.*

Proof. Assume that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ then, by confluence, there exist A_0 and B_0 such that $A_1 \rightarrow^*_{\beta\Gamma} A_0$, $A_2 \rightarrow^*_{\beta\Gamma} A_0$, $B_1 \rightarrow^*_{\beta\Gamma} B_0$ and $B_2 \rightarrow^*_{\beta\Gamma} B_0$. It follows that $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$. □

Section 1.4 provides several criteria for proving confluence of $\rightarrow_{\beta\Gamma}$.

Moreover, product compatibility cannot be lost by adding object constants in the global context.

Lemma 2.6.12. *Let Γ be a well-typed global context. If $\Gamma;\emptyset \vdash \underline{U} : \mathbf{Type}$, then product compatibility holds for $\Gamma(\underline{c} : \underline{U})$.*

Proof. Suppose that $\Pi x : A_1.B_1 \equiv_{\beta(\Gamma(\underline{c}:\underline{U}))} \Pi x : A_2.B_2$, $\Gamma(\underline{c} : \underline{U}); \Delta \vdash \Pi x : A_1.B_1 : s_1$ and $\Gamma(\underline{c} : \underline{U}); \Delta \vdash \Pi x : A_2.B_2 : s_2$.

Then we have, for a fresh variable z, $(\Pi x : A_1.B_1)[c/z] \equiv_{\beta\Gamma} (\Pi x : A_2.B_2)[c/z]$, $\Gamma; (z : U)\Delta[c/z] \vdash (\Pi x : A_1.B_1)[c/z] : s_1$ and $\Gamma; (z : U)\Delta[c/z] \vdash (\Pi x : A_2.B_2)[c/z] : s_2$.

By product compatibility for Γ, we have $A_1[c/z] \equiv_{\beta\Gamma} A_2[c/z]$ and $B_1[c/z] \equiv_{\beta\Gamma} B_2[c/z]$.

It follows that $A_1 \equiv_{\beta(\Gamma(\underline{c}:\underline{U}))} A_2$ and $B_1 \equiv_{\beta(\Gamma(\underline{c}:\underline{U}))} B_2$. □

Because type variables do not exist in the $\lambda\Pi$-Calculus Modulo, the proof of the lemma above does not work for type declarations.

However, we conjecture that the result holds nonetheless for type declarations.

Conjecture 2.6.13. *Let Γ be a well-typed global context. If $\Gamma; \emptyset \vdash \underline{C} : \mathbf{Kind}$, then product compatibility holds for $\Gamma(\underline{C} : K)$.*

2.6.3 Strongly Well-Formed Global Contexts

In this section, we prove that strongly well-formed global contexts are well-typed, opening the road to an iterative way to check well-typedness of global contexts.

Lemma 2.6.14 (Well-typed Global Declarations)**.** *If Γ is a strongly well-formed global context, then, for all $(c : A) \in \Gamma$, we have $\Gamma; \emptyset \vdash A : s$, for some sort s.*

Proof. By induction on the derivation of Γ **swf** and global weakening (Lemma 2.6.5). □

Lemma 2.6.15 (Product Compatibility)**.** *Product compatibility holds for strongly well-formed global contexts.*

Proof. By induction on the derivation of Γ **swf**, we have that $\rightarrow_{\beta\Gamma}$ is confluent. Therefore, Theorem 2.6.11 applies. □

A rewrite rule that remains well-typed in all possible extensions of the global context Γ is called permanently well-typed in Γ.

Definition 2.6.16 (Permanently Well-Typed Rewrite Rules)**.** *A rewrite rule is* permanently well-typed *in Γ if it is well-typed for any well-typed extension $\Gamma_0 \supset \Gamma$.*

The notion of permanently well-typed rewrite rule makes possible to type-check rewrite rules once and for all and not each time we make new declarations or add other rewrite rules in the context.

We will see in Chapter 3 that not all well-typed rewrite rules are permanently well-typed.

Remark 2.6.17. *If a rewrite rule is permanently well-typed in Γ, then it is also permanently well-typed in any extension of Γ.*

Strongly well-formed rewrite rules are permanently well-typed.

Lemma 2.6.18. *Let Γ be a well-typed global context and $(u \hookrightarrow v)$ be a rewrite rule. If $(u \hookrightarrow v)$ is strongly well-formed in Γ, then it is permanently well-typed in Γ.*

Proof. See Theorem 3.2.1. □

We are now able to prove that strongly well-formed global contexts are well-typed.

Theorem 2.6.19. *If Γ is a strongly well-formed global context, then it is well-typed.*

Proof. We already know that global declarations are well-typed (Lemma 2.6.14) and that product compatibility holds (Lemma 2.6.15).

We prove, by induction on the derivation of Γ **swf**, that the rewrite rules in Γ are permanently well-typed.

- **(Empty Global Context)** Trivial.
- **(Object Declaration)**, **(Type Declaration)** By Remark 2.6.17.
- **(Rewrite Rules)** By Remark 2.6.17, Lemma 2.6.18 and induction hypothesis.

□

2.6.4 Subject Reduction

The subject reduction property (also called type preservation property) is a key property of a type system. It basically says that reduction preserves typing and it implies that the set of well-typed terms is closed by reduction. It is the very purpose of a type system to capture an information (the type) that is invariant by computation (reduction). Therefore, a type system that does not satisfy such property is not interesting. In particular, one cannot hope to use it to prove dynamic properties such as termination.

We claimed in the previous sections that subject reduction for $\rightarrow_\beta$ follows from product compatibility. We now prove it.

Lemma 2.6.20 (Subject Reduction for $\rightarrow_\beta$)**.** *Let Γ be a global context satisfying well-typedness of rewrite rules and product compatibility and let Δ be a local context well-formed for Γ.*

If $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \rightarrow_\beta t_2$, then $\Gamma;\Delta \vdash t_2 : T$.

Proof. We proceed by induction on $\Gamma;\Delta \vdash t_1 : T$.

- **(Sort)**, **(Constant)**, **(Variable)**. Impossible.
- **(Application)**. $t_1 = u_1\, v_1$ and there exist A and B such that $\Gamma;\Delta \vdash u_1 : \Pi x : A_1.B_1$, $\Gamma;\Delta \vdash v_1 : A_1$ and $T = B_1[x/v_1]$.
  - If $u_1 = (\lambda x : A_2.u_0)$ and $t_2 = u_0[x/v_1]$ then, by inversion (Lemma 2.6.1), $\Gamma;\Delta \vdash A_2 :$ **Type**, $\Gamma;\Delta(x : A_2) \vdash u_0 : B_2$ and $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$. By product compatibility, $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$. Therefore, we have $\Gamma;\Delta \vdash v_1 : A_2$ and, by the property of well-typed substitutions (Lemma 2.6.9), we have $\Gamma;\Delta \vdash u_0[x/v_1] : B_2[x/v_1]$. Finally, since, by stratification (Lemma 2.6.10), T is well-typed and $T \equiv_{\beta\Gamma} B_2[x/v_1]$, we have $\Gamma;\Delta \vdash u_0[x/v_1] : T$.
  - If $t_2 = u_2\, v_1$ with $u_1 \rightarrow_\beta u_2$, then the induction hypothesis applies.
  - If $t_2 = u_1\, v_2$ with $v_1 \rightarrow_\beta v_2$ then, using the induction hypothesis, we get $\Gamma;\Delta \vdash t_2 : B_1[x/v_2]$. By Lemma 2.6.10, $\Gamma;\Delta \vdash T : s$. Therefore, using the conversion rule, we have $\Gamma;\Delta \vdash t_2 : T$.

- **(Abstraction)**. $t_1 = \lambda x : A_1.u_1$ and there is B such that $\Gamma;\Delta \vdash A_1 : \mathbf{Type}$, $\Gamma;\Delta(x : A_1) \vdash u_1 : B$, and $T = \Pi x : A.B$.
  - If $t_2 = \lambda x : A_1.u_2$ with $u_1 \to_\beta u_2$, then, by induction hypothesis, $\Gamma;\Delta(x : A_1) \vdash u_2 : B$ and therefore $\Gamma;\Delta \vdash \lambda x : A_1.u_2 : \Pi x : A_1.B$.
  - If $t_2 = \lambda x : A_2.u_1$ with $A_1 \to_\beta A_2$, then, by induction hypothesis, we have $\Gamma;\Delta \vdash A_2 : \mathbf{Type}$. By Lemma 2.6.8, $\Gamma;\Delta(x : A_2) \vdash u_1 : B$ and therefore $\Gamma;\Delta \vdash \lambda x : A_2.u_1 : \Pi x : A_2.B$. By Lemma 2.6.10, we have $\Gamma;\Delta \vdash \Pi x : A_2.B : s$. Therefore, by the conversion rule, $\Gamma;\Delta \vdash \lambda x : A_2.u_1 : \Pi x : A_1.B$.
- **(Product)**. $t_1 = \Pi x : A_1.B_1$ and we have $\Gamma;\Delta \vdash A_1 : \mathbf{Type}$ and $\Gamma;\Delta(x : A_1) \vdash B_1 : s$, for some sort s.
  - If $t_2 = \Pi x : A_1.B_2$ with $B_1 \to_\beta B_2$, then, by induction hypothesis, we have $\Gamma;\Delta(x : A_1) \vdash B_2 : s$ and therefore $\Gamma;\Delta \vdash t_2 : s$.
  - If $t_2 = \Pi x : A_2.B_1$ with $A_1 \to_\beta A_2$, then, by induction hypothesis, we have $\Gamma;\Delta \vdash A_2 : \mathbf{Type}$ and, by Lemma 2.6.8, $\Gamma;\Delta(x : A_2) \vdash B_1 : s$. It follows that $\Gamma;\Delta \vdash \Pi x : A_2.B_1 : s$.
- **(Conversion)**. By induction hypothesis.

□

Subject reduction for $\to_\Gamma$ directly follows from well-typedness of rewrite rules.

Lemma 2.6.21 (Subject Reduction for $\to_\Gamma$)**.** *Let Γ be a global context satisfying well-typedness of declarations and well-typedness of rewrite rules and let Δ be a local context well-formed for Γ.*

If $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \to_\Gamma t_2$, then $\Gamma;\Delta \vdash t_2 : T$.

Proof. Same proof as for Lemma 2.6.20. Remark that product compatibility is not needed, since we do not need to consider β-redexes. However, well-typedness of rewrite rules is needed to deal with Γ-redexes. □

From the lemma above, one can deduce that product compatibility is not necessary to prove subject reduction for $\to_\Gamma$. This is not quite true since one cannot expect to prove that a (dependently typed) rewrite rule is well-typed without product compatibility (see the proof of Lemma 2.6.18).

Theorem 2.6.22 (Subject Reduction)**.** *Let Γ be a well-typed global context and Δ a local context well-formed for Γ.*

If $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \to_{\beta\Gamma} t_2$, then $\Gamma;\Delta \vdash t_2 : T$.

Proof. It follows from Lemma 2.6.20 and Lemma 2.6.21. □

Corollary 2.6.23. *Subject reduction holds for strongly well-formed global contexts.*

Proof. By Theorem 2.6.22, since strongly well-formed global contexts are well-typed (Theorem 2.6.19). □

2.6.5 Uniqueness of Types

Uniqueness of types is another key property of the $\lambda\Pi$-Calculus Modulo. It states that all the types of a given term are convertible. An important application of this is that type checking can be reduced to type inference and a convertibility check.

Definition 2.6.24 (Uniqueness of Types)**.** *The property of* uniqueness of types *for a global context* Γ*, written* $\mathbf{UT}(\Gamma)$*, is the following proposition.*

For any terms t, T_1, T_2 *and local context* Δ*, if* $\Gamma \vdash^{ctx} \Delta$ *and* $\Gamma;\Delta \vdash t : T_1$ *and* $\Gamma;\Delta \vdash t : T_2$*, then* $T_1 \equiv_{\beta\Gamma} T_2$*.*

Theorem 2.6.25. *If* Γ *is a well-typed global context, then it satisfies the uniqueness of types property.*

Proof. By induction on the first typing derivation.

- **(Sort)**, **(Variable)**, **(Constant)** By inversion on the second typing derivation.
- **(Application)** Suppose that $t = uv$, $\Gamma;\Delta \vdash u : \Pi x : A_1.B_1$, $\Gamma;\Delta \vdash v : A_1$ and $T_1 = B_1[x/v]$. By inversion on the second typing derivation, $\Gamma;\Delta \vdash u : \Pi x : A_2.B_2$, $\Gamma;\Delta \vdash v : A_2$ and $T_2 \equiv_{\beta\Gamma} B_2[x/v]$. By induction hypothesis, we have $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$. By product compatibility, $B_1 \equiv_{\beta\Gamma} B_2$. Therefore, $T_1 = B_1[x/v] \equiv_{\beta\Gamma} B_2[x/v] \equiv_{\beta\Gamma} T_2$.
- **(Abstraction)** By inversion on the second typing derivation and induction hypothesis.
- **(Product)** By inversion on the second typing derivation, induction hypothesis and Lemma 2.3.5.
- **(Conversion)** By induction hypothesis.

□

Remark 2.6.26. *We have also shown that the* **(Conversion)** *rule commutes with the other inference rules.*

Corollary 2.6.27. *Uniqueness of types holds for strongly well-formed global contexts.*

Proof. By Theorem 2.6.25, since strongly well-formed global contexts are well-typed (Theorem 2.6.19). □

2.6.6 Undecidability Results

We have seen that subject reduction (for $\rightarrow_\beta$) and uniqueness of types are key properties of the $\lambda\Pi$-Calculus Modulo and that they follow from product compatibility. We study in this section the exact relation between these properties and show that one cannot decide in general if they hold for a particular global context.

Product Compatibility from Subject Reduction for $\rightarrow_\beta$

Lemma 2.6.20 shows that product compatibility implies subject reduction for $\rightarrow_\beta$. In this section we show that these properties are in fact equivalent.

First we prove that product compatibility always holds for non-dependent products.

Lemma 2.6.28 (Non-Dependent Product Compatibility)**.** *Let* Γ *be a global context whose declarations are well-typed and satisfying the subject reduction property for* β*.*

If $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$, $\Gamma \vdash^{ctx} \Delta$, $\Gamma;\Delta \vdash \Pi x : A_1.B_1 : s_1$, $\Gamma;\Delta \vdash \Pi x : A_2.B_2 : s_2$ *and* B_1 *does not depend on* x*, then, for all* a *such that* $\Gamma;\Delta \vdash a : A_2$*, we have* $B_1 \equiv_{\beta\Gamma} B_2[x/a]$*.*

Proof. By Lemma 2.3.5 and stratification (Lemma 2.6.10), $s_1 = s_2$. If $s_1 = s_2 =$ **Kind**, then Lemma 2.3.5 applies. Otherwise, $s_1 = s_2 =$ **Type**. From $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ and $\Gamma;\Delta \vdash a : A_2$ we can derive $\Gamma;\Delta(y : B_1) \vdash (\lambda x : A_1.y)a : B_2[x/a]$ for some fresh variable y. By subject reduction, $\Gamma;\Delta(y : B_1) \vdash y : B_2[x/a]$. Finally, by inversion (Lemma 2.6.1), we have $B_1 \equiv_{\beta\Gamma} B_2[x/a]$. □

Corollary 2.6.29. *Let* Γ *be a global context whose declarations are well-typed and satisfying the subject reduction property for* β*.*

If $\Gamma \vdash^{ctx} \Delta$ *and* $\Gamma;\Delta \vdash (\lambda x : A.x)a : T$ *then* $A \equiv_{\beta\Gamma} T$*.*

Proof. By inversion (Lemma 2.6.1), we have $\Gamma;\Delta \vdash \lambda x : A.x : \Pi x : A_2.B_2$, $\Gamma;\Delta \vdash a : A_2$ and $T \equiv_{\beta\Gamma} B_2[x/a]$. Also, by inversion, we have $\Gamma;\Delta(x : A) \vdash x : B_1$ (with $A \equiv_{\beta\Gamma} B_1$), for some B_1 and $\Pi x : A_2.B_2 \equiv_{\beta\Gamma} \Pi x : A.B_1 \equiv_{\beta\Gamma} \Pi x : A.A$. Finally, by Lemma 2.6.28, we have $A \equiv_{\beta\Gamma} B_2[x/a] \equiv_{\beta\Gamma} T$. □

Lemma 2.6.30. *Let* Γ *a global context whose declarations are well-typed.*

$$\mathbf{PC}(\Gamma) \iff \mathbf{SR}^{\beta}(\Gamma)$$

Proof. We already know that product compatibility implies subject reduction for β (Lemma 2.6.20).

We now prove the converse. Assume that $\Gamma \vdash^{ctx} \Delta$, $\Gamma;\Delta \vdash \Pi x : A_1.B_1 : s_1$, $\Gamma;\Delta \vdash \Pi x : A_2.B_2 : s_2$ and $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$. By stratification (Lemma 2.3.5 and Lemma 2.6.10), $s_1 = s_2$. If $s_1 = s_2 =$ **Kind** then Lemma 2.3.5 applies. Otherwise, $s_1 = s_2 =$ **Type**.

- From $\Gamma;\Delta(a : A_2)(f : \Pi x : A_1.B_1) \vdash (\lambda x : A_1.f((\lambda y : A_1.y)x))a : B_2[x/a]$ we deduce, by subject reduction, $\Gamma;\Delta(a : A_2)(f : \Pi x : A_1.B_1) \vdash f((\lambda y : A_1.y)a) : B_2[x/a]$. The set of typed terms being closed by taking a subterm (Lemma 2.6.3), we have, for some T, $\Gamma;\Delta(a : A_2)(f : \Pi x : A_1.B_1) \vdash (\lambda y : A_1.y)a : T$. Then, by Corollary 2.6.29, we have $A_1 \equiv_{\beta\Gamma} T$. β-Reducing further, we have $\Gamma;\Delta(a : A_2)(f : \Pi x : A_1.B_1) \vdash a : T$. Thus, by inversion, $T \equiv_{\beta\Gamma} A_2$. It follows that $A_1 \equiv_{\beta\Gamma} A_2$.
- From $\Gamma;\Delta(x : A_2)(f : \Pi x : A_1.B_1) \vdash (\lambda y : A_1.((\lambda z : B_1[x/y].z)(fy)))x : B_2$, we deduce, by subject reduction, $\Gamma;\Delta(x : A_2)(f : \Pi x : A_1.B_1) \vdash (\lambda z : B_1.z)(fx) : B_2$. By Corollary 2.6.29, we have $B_1 \equiv_{\beta\Gamma} B_2$.

□

As a corollary, we get that subject reduction implies uniqueness of types.

Corollary 2.6.31. *Let* Γ *a global context whose declarations are well-typed.*

$$\mathbf{SR}^{\beta}(\Gamma) \implies \mathbf{UT}(\Gamma)$$

Proof. Remark that, in the proof of Theorem 2.6.25, we only need product compatibility. Therefore, since subject reduction for β implies product compatibility (Lemma 2.6.30), it also implies uniqueness of types. □

Right Product Compatibility from Uniqueness of Type

Product compatibility implies uniqueness of types (Theorem 2.6.25). The converse is not true. In fact, uniqueness of types is equivalent to a restricted notion of product compatibility that we call right product compatibility.

Definition 2.6.32 (Right Product Compatibility)**.** *A global context* Γ *satisfies the* right product compatibility *property if, for all* A, B_1, B_2, s_1, s_2 *and* Δ*, if* $\Gamma \vdash^{ctx} \Delta, \Gamma; \Delta \vdash \Pi x : A.B_1 : s_1, \Gamma; \Delta \vdash \Pi x : A.B_2 : s_2$ *and* $\Pi x : A.B_1 \equiv_{\beta\Gamma} \Pi x : A.B_2$*, then* $B_1 \equiv_{\beta\Gamma} B_2$*.*

We write **R-PC**(Γ) *if* Γ *satisfies right product compatibility.*

Remark 2.6.33. *Product compatibility implies right product compatibility.*

Lemma 2.6.34. *Let* Γ *be a global context whose declarations are well-typed.*

$$\textbf{R-PC}(\Gamma) \iff \textbf{UT}(\Gamma)$$

Proof.

- To prove that right product compatibility implies uniqueness of types, it suffices to adapt Theorem 2.6.25. Remark that, in the proof, we do not need full product compatibility. Indeed, from $A_1 \equiv_{\beta\Gamma} A_2$ (induction hypothesis) and $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$, we can deduce that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_1.B_2$. Then, by right product compatibility, we have $B_1 \equiv_{\beta\Gamma} B_2$.
- We now prove that uniqueness of types implies right product compatibility. Assume that $\Gamma; \Delta \vdash \Pi x : A.B_1 : s_1$, $\Gamma; \Delta \vdash \Pi x : A.B_2 : s_2$ and $\Pi x : A.B_1 \equiv_{\beta\Gamma} \Pi x : A.B_2$. By Lemma 2.3.5 and Lemma 2.6.10, $s_1 = s_2$.
  - If $s_1 = s_2 =$ **Kind**, then Lemma 2.3.5 applies.
  - If $s_1 = s_2 =$ **Type**, then we have $\Gamma; \Delta(f : \Pi x : A.B_1)(x : A) \vdash f\ x : B_1, \Gamma; \Delta(f : \Pi x : A.B_1)(x : A) \vdash f : \Pi x : A.B_2$ and therefore $\Gamma; \Delta(f : \Pi x : A.B_1)(x : A) \vdash f\ x : B_2$. By uniqueness of types, we deduce $B_1 \equiv_{\beta\Gamma} B_2$.

□

Undecidability of (Right) Product Compatibility

In this section we prove that product compatibility and right product compatibility are undecidable properties. Therefore, subject reduction and uniqueness of types are also undecidable. This is not really surprising; product compatibility has a lot to do with confluence and confluence is already undecidable for (first-order) term rewriting systems. Indeed, we adapt the proof of undecidability for confluence found in [Ter03] to product compatibility.

We will reduce the following word problem to both product compatibility and right product compatibility.

Theorem 2.6.35 (Matijasevitch [Mat67])**.** *The equality relation on words over the alphabet* $\{a, b\}$ *generated from the set of equations* E *of Figure 2.7 is undecidable.*

Lemma 2.6.36 (Undecidability of (Right) Product Compatibility)**.** *Let* Γ *be a global context. Product compatibility in* Γ *is undecidable. Right product compatibility in* Γ *is undecidable.*

$$
\begin{array}{lcl}
x(yz) & = & (xy)z \\
abaabb & = & bbaaba \\
aababba & = & bbaaaba \\
abaaabb & = & abbabaa \\
bbbaabbaaba & = & bbbaabbaaaa \\
aaaabbaaba & = & bbaaaa
\end{array}
$$

Figure 2.7: Equational theory with an undecidable word problem.

Proof. We reduce the word problem for E (Figure 2.7) to (right) product-compatibility. To each pair of words $p = (w_1, w_2)$ on the alphabet $\{a, b\}$, we associate a global context Γ_p. Then we prove that $w_1 =_E w_2$ if and only if (right) product compatibility holds for Γ_p.

Γ_p is built from a prefix common for every pair.

$\texttt{Word} : \textbf{Type}.$
$\dot{\epsilon} : \texttt{Word}.$
$\dot{a} : \texttt{Word} \longrightarrow \texttt{Word}.$
$\dot{b} : \texttt{Word} \longrightarrow \texttt{Word}.$

Word corresponds to the type of words. $\dot{\epsilon}$ corresponds to the empty word. $\dot{a}$ and $\dot{b}$ are word constructors, appending the letter a or b to a word.

This allows us to define an encoding $|w|$ of a word w.

$$|\epsilon| \mapsto \dot{\epsilon}, \quad |aw_0| \mapsto \dot{a}|w_0|, \quad |bw_0| \mapsto \dot{b}|w_0|,$$

Since we want to work modulo the equalities in E, we add them as rewrite rules in both directions:

$$
\begin{array}{lcl}
|abaabb| & \hookrightarrow & |bbaaba|. \\
|aababba| & \hookrightarrow & |bbaaaba|. \\
|abaaabb| & \hookrightarrow & |abbabaa|. \\
|bbbaabbaaba| & \hookrightarrow & |bbbaabbaaaa|. \\
|aaaabbaaba| & \hookrightarrow & |bbaaaa|.
\end{array}
$$

$$
\begin{array}{lcl}
|bbaaba| & \hookrightarrow & |abaabb|. \\
|bbaaaba| & \hookrightarrow & |aababba|. \\
|abbabaa| & \hookrightarrow & |abaaabb|. \\
|bbbaabbaaaa| & \hookrightarrow & |bbbaabbaaba|. \\
|bbaaaa| & \hookrightarrow & |aaaabbaaba|.
\end{array}
$$

Finally, to get Γ_p for a given pair $p = (w_1, w_2)$, we add two type declarations and two rewrite rules:

$B : \texttt{Word} \longrightarrow \textbf{Type}.$
$T : \textbf{Type}.$
$T \hookrightarrow \texttt{Word} \longrightarrow B\ |w_1|.$
$T \hookrightarrow \texttt{Word} \longrightarrow B\ |w_2|.$

First we prove that, if $w_1 =_E w_2$, then (right) product compatibility holds for Γ (write Γ for Γ_p). By Theorem 2.6.11, it suffices to prove that the reduction is confluent,

and since $\to_\Gamma$ is orthogonal to $\to_\beta$, by Theorem 1.4.7, it suffices to prove that $\to_\Gamma$ is confluent. Suppose that $t_0 \to_\Gamma^* t_1$ and $t_0 \to_\Gamma^* t_2$. Then $t_0 = C[T,\dots,T]$ where C is a multi context that does not contain T, $t_1 = C_1[U_1,\dots,U_n]$ with $C \to_\Gamma^* C_1$ (without using a T-rule) and $T \to_{\beta\Gamma}^* U_i$ and $t_2 = C_2[V_1,\dots,V_n]$ with $C \to_\Gamma^* C_2$ (without using a T-rule) and $T \to_{\beta\Gamma}^* V_i$. Since all rewrite rules except the rules on T are reversible, we have $C_1 \to_{\beta\Gamma}^* C$, $C_2 \to_{\beta\Gamma}^* C$, $|w_1| \to_{\beta\Gamma}^* |w_2|$ and, for all i, $U_i \to_{\beta\Gamma}^* \mathtt{Word} \longrightarrow B\,|w_2|$ and $V_i \to_{\beta\Gamma}^* \mathtt{Word} \longrightarrow B\,|w_2|$. It follows that both t_1 and t_2 reduce to $C[\mathtt{Word} \longrightarrow B\,|w_2|,\dots,\mathtt{Word} \longrightarrow B\,|w_2|]$.

Now, we prove that, if (right) product compatibility holds for Γ, then $w_1 =_E w_2$. Since $\mathtt{Word} \longrightarrow B\,|w_1| \leftarrow T \to \mathtt{Word} \longrightarrow B\,|w_2|$, we have, by right product compatibility, $B\,|w_1| \equiv_{\beta\Gamma} B\,|w_2|$. In fact we have also $B\,|w_1| \equiv_{\beta\Gamma} B\,|w_2|$ without T-rule (replace any T and reduct of T by, say, $\mathtt{Word}$). And, since the rewrite system without the T rules is confluent and there are no possible β-redexes, we have $B\,|w_1| \downarrow_\Gamma B\,|w_2|$. It follows that $|w_1| \downarrow |w_2|$ and $w_1 =_E w_2$. □

Corollary 2.6.37 (Undecidability of Subject Reduction)**.** *Subject reduction of β-reduction in a global context is undecidable.*

Proof. It follows from Lemma 2.6.30 and Lemma 2.6.36. □

Corollary 2.6.38 (Undecidability of Uniqueness of Type)**.** *Uniqueness of types in a global context is undecidable.*

Proof. It follows from Lemma 2.6.34 and Lemma 2.6.36. □

2.7 Applications

As already said, the main purpose of the $\lambda\Pi$-Calculus Modulo is to serve as a logical framework, that is, a framework in which we express other logics or calculus by encoding them. The $\lambda\Pi$-Calculus has also been used with success for the same purpose [HHP93, CHJ+12]. The advantage of the $\lambda\Pi$-Calculus Modulo is that, thanks to its extended notion of rewriting, it allows designing *shallow* encodings. The term *shallow* has been used in the literature as an informal term meaning that an encoding reuses the target language features to encode the corresponding features in the source language. In particular, if there exists a notion of computation in the encoded language, we want to preserve it by the translation. More concretely, a reduction step in the source language should correspond to one or more β (Γ)-reduction steps in the $\lambda\Pi$-Calculus (Modulo). Another example: if there is a notion of judgment in the source language, as in typed λ-calculi or in natural deduction, then we want to link this notion to the typing judgments of the $\lambda\Pi$-Calculus (Modulo). An encoding that is not shallow is *deep*.

In this section, we illustrate the use of the $\lambda\Pi$-Calculus Modulo by giving shallow encodings for various logics and calculus. We proceed in the same way for each example; first we define a global context declaring the constants and rewrite rules used by the encoding; then we explain how we encode the elements of the source language (depending on the encoded system these are terms, propositions and/or proofs); finally we prove theorems formalizing the correspondence between the initial system and the encoding.

2.7.1 Constructive Predicate Logic

In this section, we encode the constructive predicate logic [Gen35] (also called first-order intuitionistic logic) following Dorra [Dor11].

Definition

Definition 2.7.1 (Terms)**.** *Let Σ be a signature, that is a set of function symbols and predicate symbols with their arity, and let $\mathcal{V}$ be a set of variables.*

Let set of first-order terms is defined as follows:

- *if x is a variable, then x is a term;*
- *if c is a function symbol of arity n and $t_1, \dots t_n$ are n terms, then $c(t_1, \dots, t_n)$ is a term.*

Definition 2.7.2 (Propositions)**.** *The set of first order propositions is defined as follows:*

- *if P is a predicate symbol of arity n and $t_1, \dots t_n$ are n terms, then $P(t_1, \dots, t_n)$ is a proposition;*
- *$\top$ and $\bot$ are propositions;*
- *if P and Q are propositions, then $P \wedge Q$, $P \vee Q$ and $P \implies Q$ are propositions;*
- *if P is a proposition, then $\forall x.P$ and $\exists x.P$ are propositions.*

Definition 2.7.3 (Constructive Predicate Logic)**.** *Let Σ be a signature and Ξ be a set of proposition.*

The proposition P is provable in the context Ξ in constructive predicate logic if the judgment $\Xi \vdash_\Sigma P$ is derivable from the inference rules of Figure 2.8.

Global Context

We introduce two types for the propositions and the terms of the logic:

`prop` : **Type**.
`term` : **Type**.

We also introduce constants for the logical connectors:

`true` : `prop`.
`false` : `prop`.
`imp` : `prop` ⟶ `prop` ⟶ `prop`.
`and` : `prop` ⟶ `prop` ⟶ `prop`.
`or` : `prop` ⟶ `prop` ⟶ `prop`.
`forall` : (`term` ⟶ `prop`) ⟶ `prop`.
`exists` : (`term` ⟶ `prop`) ⟶ `prop`.

We add a constant to represent the proofs of a given proposition as a type:

`prf` : `prop` ⟶ **Type**.

Now we add rewrite rules to describe what the proofs of a proposition are. We mimic the elimination rules of natural deduction.

$$\frac{}{\Xi \vdash_\Sigma \top}\ \text{(True)} \qquad \frac{P \in \Xi}{\Xi \vdash_\Sigma P}\ \text{(Axiom)} \qquad \frac{\Xi, P \vdash_\Sigma Q}{\Xi \vdash_\Sigma P \Longrightarrow Q}\ (\Longrightarrow\text{-intro})$$

$$\frac{\Xi \vdash_\Sigma P \Longrightarrow Q \qquad \Xi \vdash_\Sigma P}{\Xi \vdash_\Sigma Q}\ (\Longrightarrow\text{-elim}) \qquad \frac{\Xi \vdash_\Sigma P \qquad \Xi \vdash_\Sigma Q}{\Xi \vdash_\Sigma P \wedge Q}\ (\wedge\text{-intro})$$

$$\frac{\Xi \vdash_\Sigma P \wedge Q}{\Xi \vdash_\Sigma P}\ \text{(Left-}\wedge\text{-elim)} \qquad \frac{\Xi \vdash_\Sigma P \wedge Q}{\Xi \vdash_\Sigma Q}\ \text{(Right-}\wedge\text{-elim)}$$

$$\frac{\Xi \vdash_\Sigma P}{\Xi \vdash_\Sigma P \vee Q}\ \text{(Left-}\vee\text{-intro)} \qquad \frac{\Xi \vdash_\Sigma Q}{\Xi \vdash_\Sigma P \vee Q}\ \text{(Right-}\vee\text{-intro)}$$

$$\frac{\Xi \vdash_\Sigma P \vee Q \qquad \Xi \vdash_\Sigma P \Longrightarrow R \qquad \Xi \vdash_\Sigma Q \Longrightarrow R}{\Xi \vdash_\Sigma R}\ (\vee\text{-elim})$$

$$\frac{\Xi \vdash_\Sigma P \qquad x \notin \Xi}{\Xi \vdash_\Sigma \forall x.P}\ (\forall\text{-intro}) \qquad \frac{\Xi \vdash_\Sigma \forall x.P}{\Xi \vdash_\Sigma P[x/t]}\ (\forall\text{-elim})$$

$$\frac{\Xi \vdash_\Sigma P[x/t]}{\Xi \vdash_\Sigma \exists x.P}\ (\exists\text{-intro})$$

$$\frac{\Xi \vdash_\Sigma \exists x.P \qquad \Xi \vdash_\Sigma P \Longrightarrow Q \qquad x \notin \Xi, Q}{\Xi \vdash_\Sigma Q}\ (\exists\text{-elim})$$

Figure 2.8: Inference rules of Constructive Predicate Logic

A proof of the implication $\texttt{imp}\ P\ Q$ is a function from the proofs of P to the proofs of Q:

$$\texttt{prf}\ (\texttt{imp}\ P\ Q) \quad \hookrightarrow \quad \texttt{prf}\ P \longrightarrow \texttt{prf}\ Q.$$

A proof of a universally quantified formula $\texttt{forall}\ P$ is a function mapping an x of type $\texttt{term}$ to a proof of $P(x)$:

$$\texttt{prf}\ (\texttt{forall}\ P) \quad \hookrightarrow \quad \Pi x : \texttt{term}.\texttt{prf}\ (P\ x).$$

A proof of the universally valid proposition is the identity function:

$$\texttt{prf}\ \texttt{true} \quad \hookrightarrow \quad \Pi P : \texttt{prop}.\texttt{prf}\ P \longrightarrow \texttt{prf}\ P.$$

A proof of the absurd proposition is a function giving a proof for any proposition:

$$\texttt{prf}\ \texttt{false} \quad \hookrightarrow \quad \Pi P : \texttt{prop}.(\texttt{prf}\ P).$$

A proof of the conjunction $\texttt{and}\ P_1\ P_2$ is a function from the proofs that P_1 implies P_2 implies Q to the proofs of Q:

$$\texttt{prf}\ (\texttt{and}\ P_1\ P_2) \quad \hookrightarrow \quad \Pi Q : \texttt{prop}.\texttt{prf}\ (\texttt{imp}\ P_1\ (\texttt{imp}\ P_2\ Q)) \longrightarrow \texttt{prf}\ Q.$$

A proof of the disjunction $\texttt{or}\ P_1\ P_2$ is a function building a proof of Q from a proof that P_1 implies Q and a proof that P_2 implies Q:

$$\texttt{prf}\ (\texttt{or}\ P_1\ P_2) \quad \hookrightarrow \quad \Pi Q : \texttt{prop}.\texttt{prf}\ (\texttt{imp}\ P_1\ Q) \longrightarrow \texttt{prf}\ (\texttt{imp}\ P_2\ Q) \longrightarrow \texttt{prf}\ Q.$$

A proof of the existentially quantified formula $\texttt{exists}\ P$ is a function building a proof of Q from a proof that $P\ x$ implies Q for some x:

$$\texttt{prf}\ (\texttt{exists}\ P) \quad \hookrightarrow \quad \Pi Q : \texttt{prop}.(\Pi x : \texttt{term}.\texttt{prf}\ (\texttt{imp}\ (P\ x)\ Q)) \longrightarrow \texttt{prf}\ Q.$$

Embedding

Let Γ^{FO} be the global context defined in the previous section.

Definition 2.7.4 (Embedding of Signatures)**.** *If Σ is a signature, then the global context $\Gamma(\Sigma)$ is obtained from Γ^{FO} by adding the following declarations:*

- *for each function symbol f of arity n, we declare a variable $\dot{f}$ of type $\texttt{term}^n \longrightarrow \texttt{term}$;*
- *for each predicate symbol P of arity n, we declare a variable $\dot{P}$ of type $\texttt{term}^n \longrightarrow \texttt{prop}$.*

We write Γ instead of $\Gamma(\Sigma)$ when no confusion is possible.

Lemma 2.7.5. *$\Gamma(\Sigma)$ is a strongly well-formed global context.*

Proof. Trivial. □

The propositions of the constructive predicate logic are translated into $\lambda\Pi$-terms using the following function.

Definition 2.7.6 (Embedding of Propositions)**.**

$$
\begin{array}{lcll}
\|x\| & = & x & \textit{where } x \textit{ is a variable} \\
\|f(t_1,\ldots,t_n)\| & = & \dot{f}\,\|t_1\|\ldots\|t_n\| & \\
\|P(t_1,\ldots,t_n)\| & = & \dot{P}\,\|t_1\|\ldots\|t_n\| & \\
\|\top\| & = & \texttt{true} & \\
\|\bot\| & = & \texttt{false} & \\
\|P \Rightarrow Q\| & = & \texttt{imp}\,\|P\|\,\|Q\| & \\
\|P \wedge Q\| & = & \texttt{and}\,\|P\|\,\|Q\| & \\
\|P \vee Q\| & = & \texttt{or}\,\|P\|\,\|Q\| & \\
\|\forall x.P\| & = & \texttt{forall}\,(\lambda x:\texttt{term}.\|P\|) & \\
\|\exists x.P\| & = & \texttt{exists}\,(\lambda x:\texttt{term}.\|P\|) &
\end{array}
$$

Notation 2.7.7. *When P is a proposition, we write $|P|$ for* $\texttt{prf}\ \|P\|$.

Definition 2.7.8 (Embedding of Hypothesis)**.** *Let $\Xi = H_1,\ldots,H_n$ be a set of propositions. We write $|\Xi|$ for the local context $(h_1 : |H_1|)\ldots(h_n : |H_n|)$.*

Remark 2.7.9. *Let Ξ be a set of proposition and $x_1,\ldots,x_n$ be the variables occurring free in Ξ. The local context $(x_1 : \texttt{term})\ldots(x_n : \texttt{term})$ $|\Xi|$ is well-formed in $\Gamma(\Sigma)$.*

Soundness and Conservativity

We have the following correspondences between the proofs in constructive predicate logic and the typing judgments for the context Γ in the $\lambda\Pi$-Calculus Modulo.

Theorem 2.7.10 (Soundness)**.** *If $\Xi \vdash_\Sigma P$ and $x_1,\ldots,x_n$ are the variables occurring free in Ξ and P, then there exists π such that $\Gamma(\Sigma);\Delta \vdash \pi : |P|$ and $\Delta = (x_1 : \texttt{term})\ldots(x_n :$ $\texttt{term})$ $|\Xi|$.*

Proof. We proceed by induction on the proof of the sequent $\Xi \vdash_\Sigma |P|$. We write Γ for $\Gamma(\Sigma)$ and V_P for $(x_1 : \texttt{term})\ldots(x_n : \texttt{term})$, where $x_1,\ldots,x_n$ are the variables free in P but not occurring in Ξ.

- **(True)** $P = \top$ We have $|P| = \Pi x : \texttt{prop}.\Pi y : |x|.|x|$ and $\Gamma;\Delta \vdash \lambda x : \texttt{prop}.\lambda y : |x|.y : \Pi x : \texttt{prop}.\Pi y : |x|.|x|$.
- **(Axiom)** $P \in \Xi$. For some h, we have $(h : |P|) \in |\Xi|$. Therefore, $\Gamma;\Delta \vdash h : |P|$.
- **($\Longrightarrow$-intro)** $P = A \Longrightarrow B$. We have $|A \Longrightarrow B| \equiv_{\beta\Gamma} |A| \longrightarrow |B|$.

  By induction hypothesis, $\Gamma;\Delta(h : |A|) \vdash \pi_B : |B|$.

  By **(Abstraction)**, we have $\Gamma;\Delta \vdash \lambda h : |A|.\pi_B : |A| \longrightarrow |B|$.

  By **(Conversion)**, it follows that $\Gamma;\Delta \vdash \lambda h : |A|.\pi_B : |P|$.
- **($\Longrightarrow$-elim)** $P = B$. By induction hypothesis, we have $\Gamma;\Delta_1 \vdash \pi_1 : |A \Longrightarrow B|$ and $\Gamma;\Delta_2 \vdash \pi_2 : |A|$.

  Since all the variable declarations in Δ_1 and Δ_2 are in Δ, by Lemma 2.6.4, we have $\Gamma;\Delta \vdash \pi_1 : |A \Longrightarrow B|$ and $\Gamma;\Delta \vdash \pi_2 : |A|$.

  Since $|A \Longrightarrow B| \equiv_{\beta\Gamma} |A| \longrightarrow |B|$, by **(Conversion)**, we have $\Gamma;\Delta \vdash \pi_1 : |A| \longrightarrow |B|$.

  By **(Application)**, $\Gamma;\Delta \vdash \pi_1\ \pi_2 : |B|$.

- **($\wedge$-intro)** $P = A \wedge B$. We have $|A \wedge B| \equiv_{\beta\Gamma} \Pi x : \mathtt{prop}.(|A| \longrightarrow |B| \longrightarrow |x|) \longrightarrow |x|$. By induction hypothesis, we have $\Gamma; \Delta_A \vdash \pi_A : |A|$ and $\Gamma; \Delta_B \vdash \pi_B : |B|$.

  Let $\Delta_2 = \Delta(x : \mathtt{prop})(y : |A| \longrightarrow |B| \longrightarrow |x|)$. Since all the variable declarations in Δ_A and Δ_B are in Δ_2, by Lemma 2.6.4, we have $\Gamma; \Delta_2 \vdash \pi_A : |A|$ and $\Gamma; \Delta_2 \vdash \pi_B : |B|$.

  By **(Application)**, we have $\Gamma; \Delta_2 \vdash y\ \pi_A\ \pi_B : |x|$.

  By **(Abstraction)**, we have $\Gamma; \Delta \vdash \lambda x : \mathtt{prop}.\lambda y : (|A| \longrightarrow |B| \longrightarrow |x|).y\ \pi_A\ \pi_B : \Pi x : \mathtt{prop}.(|A| \longrightarrow |B| \longrightarrow |x|) \longrightarrow |x|$.

  Finally, by **(Conversion)**, we have $\Gamma; \Delta \vdash \lambda x : \mathtt{prop}.\lambda y : (|A| \longrightarrow |B| \longrightarrow |x|).y\ \pi_A\ \pi_B : |A \wedge B|$.

- **(Left-$\wedge$-elim)** $P = A$. By induction hypothesis, we have $\Gamma; \Delta \vdash \pi_0 : |A \wedge B|$.

  Since $\Pi x : \mathtt{prop}.(|A| \longrightarrow |B| \longrightarrow |x|) \longrightarrow |x| \equiv_{\beta\Gamma} |A \wedge B|$, by **(Conversion)**, we have $\Gamma; \Delta \vdash \pi_0 : \Pi x : \mathtt{prop}.(|A| \longrightarrow |B| \longrightarrow |x|) \longrightarrow |x|$.

  By **(Application)**, we have $\Gamma; \Delta \vdash \pi_0\ \|A\| : (|A| \longrightarrow |B| \longrightarrow |A|) \longrightarrow |A|$

  Moreover, by **(Abstraction)**, we have $\Gamma; \Delta \vdash \lambda x : |A|.\lambda y : |B|.x : |A| \longrightarrow |B| \longrightarrow |A|$.

  Therefore, by **(Application)**, we have $\Gamma; \Delta \vdash \pi_0\ \|A\|\ (\lambda x : |A|.\lambda y : |B|.x) : |A|$.

- **(Right-$\wedge$-elim)** $P = B$. As **(Left-$\wedge$-elim)**, with $\pi = \pi_0\ \|B\|\ (\lambda x : |A|.\lambda y : |B|.y)$.

- **(Left-$\vee$-intro)** $P = A \vee B$. We have $|A \vee B| \equiv_{\beta\Gamma} \Pi x : \mathtt{prop}.(|A| \longrightarrow |x|) \longrightarrow (|B| \longrightarrow |x|) \longrightarrow |x|$, By induction hypothesis, we have $\Gamma; \Delta_A \vdash \pi_A : |A|$.

  Since all the variable declarations in Δ_A are in Δ, by local weakening (Lemma 2.6.4), we have $\Gamma; \Delta(x : \mathtt{prop})(a : |A| \longrightarrow |x|)(b : |B| \longrightarrow |x|) \vdash \pi_A : |A|$.

  By **(Application)**, we have $\Gamma; \Delta(x : \mathtt{prop})(a : |A| \longrightarrow |x|)(b : |B| \longrightarrow |x|) \vdash a\ \pi_A : |x|$.

  By **(Abstraction)**, we have $\Gamma; \Delta \vdash \lambda x : \mathtt{prop}.\lambda a : |A| \longrightarrow |x|.\lambda b : |B| \longrightarrow |x|.a\ \pi_A : \Pi x : \mathtt{prop}.(|A| \longrightarrow |x|) \longrightarrow (|B| \longrightarrow |x|) \longrightarrow |x|$.

  By **(Conversion)**, we have $\Gamma; \Delta \vdash \lambda x : \mathtt{prop}.\lambda a : |A| \longrightarrow |x|.\lambda b : |B| \longrightarrow |x|.a\ \pi_A : |A \vee B|$.

- **(Right-$\vee$-intro)** $P = A \vee B$. As **(Left-$\vee$-intro)**, with $\pi = \lambda x : \mathtt{prop}.\lambda a : |A| \longrightarrow |x|.\lambda b : |B| \longrightarrow |x|.b\ \pi_B$.

- **($\vee$-elim)** By induction hypothesis, we have $\Gamma; \Delta_1 \vdash \pi_1 : |A \vee B|$, $\Gamma; \Delta_2 \vdash \pi_2 : |A \implies P|$ and $\Gamma; \Delta_3 \vdash \pi_3 : |B \implies P|$.

  By Lemma 2.6.4, we have $\Gamma; \Delta \vdash \pi_1 : |A \vee B|$, $\Gamma; \Delta \vdash \pi_2 : |A \implies P|$ and $\Gamma; \Delta \vdash \pi_3 : |B \implies P|$.

  Since $|A \vee B| \equiv_{\beta\Gamma} \Pi x : \mathtt{prop}.(|A| \longrightarrow |x|) \longrightarrow (|B| \longrightarrow |x|) \longrightarrow |x|$, $|A \implies P| \equiv_{\beta\Gamma} |A| \longrightarrow |B|$ and $|B \implies P| \equiv_{\beta\Gamma} |B| \longrightarrow |P|$, by **(Conversion)**, we have $\Gamma; \Delta \vdash \pi_1 : \Pi x : \mathtt{prop}.(|A| \longrightarrow |x|) \longrightarrow (|B| \longrightarrow |x|) \longrightarrow |x|$, $\Gamma; \Delta \vdash \pi_2 : |A| \longrightarrow |P|$ and $\Gamma; \Delta \vdash \pi_3 : |B| \longrightarrow |P|$.

  By **(Application)**, we have $\Gamma; \Delta \vdash \pi_1\ \|P\|\ \pi_2\ \pi_3 : |P|$.

- **($\forall$-intro)** $P = \forall x.A$. We have $|\forall x.A| \equiv_{\beta\Gamma} \Pi x : \mathtt{term}.|A|$.

  By induction hypothesis, we have $\Gamma; \Delta_A \vdash \pi_A : |A|$

By Lemma 2.6.4, we have $\Gamma;\Delta(x:\texttt{term}) \vdash \pi_A : |A|$.

By **(Abstraction)**, we have $\Gamma;\Delta \vdash \lambda x:\texttt{term}.\pi_A : \Pi x:\texttt{term}.|A|$.

By **(Conversion)**, we have $\Gamma;\Delta \vdash \lambda x.\texttt{term}.\pi_A : |\forall x.A|$.

- **($\forall$-elim)** $P = A[x/t]$. We have $|A[x/t]| = |A|[x/\|t\|]$.

  By induction hypothesis, we have $\Gamma;\Delta_0 \vdash \pi_0 : |\forall x.A|$.

  By Lemma 2.6.8, we have $\Gamma;\Delta \vdash \pi_0 : |\forall x.A|$.

  Since $|\forall x.A| \equiv_{\beta\Gamma} \Pi x:\texttt{term}.|A|$, by **(Conversion)**, we have $\Gamma;\Delta \vdash \pi_0 : \Pi x:\texttt{term}.|A|$.

  By **(Application)**, $\Gamma;\Delta \vdash \pi\ \|t\| : |A[x/t]|$.

- **($\exists$-intro)** $P = \exists x.A$. We have $|\exists x.A| \equiv_{\beta\Gamma} \Pi y:\texttt{prop}.(\Pi z:\texttt{term}.|A[x/z]| \longrightarrow |y|) \longrightarrow |y|$.

  By induction hypothesis, we have $\Gamma;\Delta \vdash \pi_0 : |A[x/t]|$.

  By local weakening (Lemma 2.6.4), we have $\Gamma;\Delta(y:\texttt{prop})(w:\Pi z:\texttt{term}.|A[x/z]| \longrightarrow |y|) \vdash \pi_0 : |A[x/t]|$.

  By **(Application)**, we have $\Gamma;\Delta(y:\texttt{prop})(w:\Pi z:\texttt{term}.|A[x/z]| \longrightarrow |y|) \vdash w\ \|t\|\ \pi_0 : |y|$.

  By **(Abstraction)**, $\Gamma;\Delta \vdash \lambda y:\texttt{prop}.\lambda w:(\Pi z:\texttt{term}.|A[x/z]| \longrightarrow |y|).w\ \|t\|\ \pi_0 : \Pi y:\texttt{prop}.(\Pi z:\texttt{term}.|A[x/z]| \longrightarrow |y|) \longrightarrow |y|$.

  By **(Conversion)**, $\Gamma;\Delta \vdash \lambda y:\texttt{prop}.\lambda : w:(\Pi z:\texttt{term}.|A[x/z]| \longrightarrow |y|).w\ \|t\|\ \pi_0 : |\exists x.A|$.

- **($\exists$-elim)** By induction hypothesis, we have $\Gamma;\Delta_1 \vdash \pi_1 : |\exists x.A|$ and $\Gamma;\Delta_2 \vdash \pi_2 : |A \implies P|$.

  By Lemma 2.6.4, we have $\Gamma;\Delta \vdash \pi_1 : |\exists x.P|$ and $\Gamma;\Delta \vdash \pi_2 : |A \implies P|$.

  By **(Conversion)**, since $|\exists x.P| \equiv_{\beta\Gamma} \Pi q:\texttt{prop}.\Pi y:\texttt{term}.\texttt{prf}\ (\texttt{imp}\ ((\lambda x:\texttt{term}.\|P\|)\ y)\ q) \longrightarrow \texttt{prf}\ q$, $\Gamma;\Delta \vdash \pi_1 : \Pi q:\texttt{prop}.\Pi y:\texttt{term}.\texttt{prf}\ (\texttt{imp}\ ((\lambda x:\texttt{term}.\|P\|)\ y)\ q) \longrightarrow \texttt{prf}\ q$.

  By **(Application)**, $\Gamma;\Delta \vdash \pi_1\ \|Q\|\ x\ \pi_2 : |P|$.

□

Lemma 2.7.11 (Termination)**.** *The relation $\rightarrow_{\beta\Gamma}$ is weakly normalizing on well-typed terms.*

Lemma 2.7.12. *Let P and Q be two propositions. If $|P| \equiv_{\beta\Gamma(\Sigma)} |Q|$, then $P = Q$.*

Proof. By induction on P and Q. □

Lemma 2.7.13. *If $\Gamma;\Delta(x:A) \vdash t : T$ and x does not occur in Δ, then $\Gamma(x:A);\Delta \vdash t : T$, where x is now an object constant.*

Proof. By induction on the typing derivation. □

Theorem 2.7.14 (Conservativity)**.** *If there exist a term π and a local context Δ such that $\Gamma(\Sigma) \vdash^{ctx} \Delta$ and $\Gamma(\Sigma);\Delta \vdash \pi : |P|$ with $\Delta = (x_1:\texttt{term})\ldots(x_n:\texttt{term})|\Xi|$, then $\Xi \vdash_\Sigma P$ is provable.*

Proof. We write Γ for $\Gamma(\Sigma)$. By Lemma 2.7.11 and subject reduction Theorem 2.6.22, we can assume that π is normal.

We proceed by induction on π. Remark that π is an object.

- π cannot be a constant or an application headed by a constant, since object-level constant have type $\mathtt{prop}$ or $T \longrightarrow \mathtt{prop}$, for some T.
- Suppose that $\pi = \lambda x : U.u$. By inversion, $\Gamma; \Delta(x : U) \vdash u : V$ and $|P| \equiv_{\beta\Gamma} \Pi x : U.V$.
  - If $P = \top$, then P is provable.
  - If $P = \bot$, then $|P| \equiv_{\beta\Gamma} \Pi x : \mathtt{prop}.|x|$. By product compatibility, we have $U \equiv_{\beta\Gamma} \mathtt{prop}$ and $V \equiv_{\beta\Gamma} |x|$.
    Therefore, we have $\Gamma; \Delta(x : \mathtt{prop}) \vdash u : |x|$.
    By Lemma 2.7.13, we have $\Gamma(\Sigma \cup \{x\}); \Delta \vdash u : |x|$.
    By induction hypothesis, $\Xi \vdash_{\Sigma\cup\{x\}} x$.
    Substituting x by $\bot$, we have $\Xi \vdash_\Sigma \bot$.
  - If $P = A \implies B$, then $|P| \equiv_{\beta\Gamma} |A| \longrightarrow |B|$. By product compatibility, we have $U \equiv_{\beta\Gamma} |A|$ and $V \equiv_{\beta\Gamma} |B|$.
    Therefore, we have $\Gamma; \Delta(x : |A|) \vdash u : |B|$.
    By induction hypothesis, $\Xi, A \vdash_\Sigma B$ is provable.
    Therefore, $\Xi \vdash_\Sigma A \implies B$ is provable.
  - If $P = A \wedge B$, then $|P| \equiv_{\beta\Gamma} \Pi x : \mathtt{prop}.|(A \implies B \implies x) \implies x|$. By product compatibility, we have $U \equiv_{\beta\Gamma} \mathtt{prop}$ and $V \equiv_{\beta\Gamma} |(A \implies B \implies x) \implies x|$.
    Therefore, we have $\Gamma; \Delta(x : \mathtt{prop}) \vdash u : |(A \implies B \implies x) \implies x|$.
    By Lemma 2.7.13, we have $\Gamma(\Sigma \cup \{x\}); \Delta \vdash u : |(A \implies B \implies x) \implies x|$.
    By induction hypothesis, $\Xi \vdash_{\Sigma\cup\{x\}} (A \implies B \implies x) \implies x$.
    Substituting x by $A \wedge B$, we get $\Xi \vdash_\Sigma (A \implies B \implies (A \wedge B)) \implies (A \wedge B)$.
    Since $\Xi \vdash_\Sigma A \implies B \implies (A \wedge B)$ is provable, $\Xi \vdash_\Sigma A \wedge B$ is also provable.
  - If $P = A \vee B$, then $|P| \equiv_{\beta\Gamma} \Pi x : \mathtt{prop}.|(A \implies x) \implies (B \implies x) \implies x|$.
    By product compatibility, we have $U \equiv_{\beta\Gamma} \mathtt{prop}$ and $V \equiv_{\beta\Gamma} |(A \implies x) \implies (B \implies x) \implies x|$.
    Therefore, we have $\Gamma; \Delta(x : \mathtt{prop}) \vdash u : |(A \implies x) \implies (B \implies x) \implies x|$.
    By Lemma 2.7.13, we have $\Gamma(\Sigma \cup \{x\}); \Delta \vdash u : |(A \implies x) \implies (B \implies x) \implies x|$.
    By induction hypothesis, $\Xi \vdash_{\Sigma\cup\{x\}} (A \implies x) \implies (B \implies x) \implies x$.
    Substituting x by $A \vee B$, we get $\Xi \vdash_\Sigma (A \implies (A \vee B)) \implies (B \implies (A \vee B)) \implies (A \vee B)$.
    Since $(A \implies (A \vee B))$ and $(B \implies (A \vee B))$ are provable, $\Xi \vdash_\Sigma A \vee B$ is also provable.
  - If $P = \forall x.A$, then $|P| \equiv_{\beta\Gamma} \Pi x : \mathtt{term}.|A|$. By product compatibility, we have $U \equiv_{\beta\Gamma} \mathtt{term}$ and $V \equiv_{\beta\Gamma} |A|$.
    Therefore, we have $\Gamma; \Delta(x : \mathtt{term}) \vdash u : |A|$.
    By Lemma 2.6.4, we have $\Gamma; (x : \mathtt{term})\Delta \vdash u : |A|$.
    By induction hypothesis, $\Xi \vdash_\Sigma A$ is provable and, since $x \notin \Xi$, $\Xi \vdash_\Sigma \forall x.A$ is also provable.

– If $P = \exists z.A$, then $|P| \equiv_{\beta\Gamma} \Pi x : \texttt{prop}.|(\forall z.A \implies x) \implies x|$.
By product compatibility, we have $U \equiv_{\beta\Gamma} \texttt{prop}$ and $V \equiv_{\beta\Gamma} |(\forall z.A \implies x) \implies x|$.
Therefore, we have $\Gamma; \Delta(x : \texttt{prop}) \vdash u : |(\forall z.A \implies x) \implies x|$.
By Lemma 2.7.13, we have $\Gamma(\Sigma \cup \{x\}); \Delta \vdash u : |(\forall z.A \implies x) \implies x|$.
By induction hypothesis, $\Xi \vdash_{\Sigma \cup \{x\}} (\forall z.A \implies x) \implies x$.
Substituting x by $\exists z.A$, we get $\Xi \vdash_\Sigma (\forall z.A \implies (\exists z.A)) \implies (\exists z.A)$.
Since $\Xi \vdash_\Sigma \forall z.A \implies (\exists z.A)$ is provable, $\Xi \vdash_\Sigma \exists z.A$ is also provable.

- Suppose that $\pi = h\ u_1 \dots u_p$. We have $\Gamma; \Delta \vdash h : |H|$ for $H \in \Xi$ and $\Xi \vdash_\Sigma H$.
We prove, by induction on q, that, if $q \leq p$, then there exists A_q such that $\Gamma; \Delta \vdash h\ u_1 \dots u_q : |A_q|$ and $\Xi \vdash_\Sigma A_q$.
If $q = 0$, we take $A_0 = H$.
If $q = r + 1$, by induction hypothesis, $\Gamma; \Delta \vdash h\ u_1 \dots u_r : |A_r|$ and $\Xi \vdash_\Sigma A_r$.
We proceed by case analysis on A_r.
  – If $A_r = \top, \bot, \wedge, \vee$ or $\exists$, then u_q has type $\texttt{prop}$ and there exists a proposition U such that $\|U\| = u_q$. If $A_r = \top$, we take $A_q = U \implies U$. If $A_r = \bot$, we take $A_q = U$. If $A_r = P_1 \wedge P_2$, we take $A_q = (P_1 \implies P_2 \implies U) \implies U$. If $A_r = P_1 \vee P_2$, we take $A_q = (P_1 \implies U) \implies (P_2 \implies U) \implies U$. If $A_r = \exists x.P_0$, we take $A_q = (\exists x.P_0(x) \implies U) \implies U$.
  – If $A_r = P_1 \implies P_2$, then u_q has type $|P_1|$. We take $A_q = P_2$.
  – If $A_r = \forall x.P_0$, then u_q has type $\texttt{term}$ and there exists a term t such that $\|t\| = u_q$. We take $A_q = P_0[x/t]$.

  By Theorem 2.6.25, $|P| \equiv_{\beta\Gamma} |A_p|$. Therefore, by Lemma 2.7.12, $P = A_p$ is provable.

□

2.7.2 The Calculus Of Constructions

In this section, we encode the Calculus of Constructions [CH88], an extension of the $\lambda\Pi$-Calculus introducing polymorphism and type operators. This is (a particular case of) the original motivational example for the $\lambda\Pi$-Calculus Modulo [CD07].

Definition

Definition 2.7.15 (Terms and Contexts)**.** *A term of the calculus of constructions is either a variable x, an application $u\,v$ when u and v are terms, an abstraction $\lambda x : A.u$ when A and u are terms, an product $\Pi x : A.B$ when A and B are terms or a sort among* **Type** *and* **Kind**.
A context is a list of variables together with a term.

Definition 2.7.16 (Calculus of Constructions)**.** *A term t has type T in the context Δ in the calculus of construction if the judgment $\Delta \vdash^{CoC} t : T$ is derivable from the inference rules of Figure 2.9.*

$$\frac{}{\vdash^{CoC} \emptyset} \qquad \frac{\vdash^{CoC} \Delta \qquad \Gamma;\Delta \vdash U : s \qquad x \notin dom(\Delta)}{\Gamma \vdash^{ctx} \Delta(x:U)}$$

$$\frac{\vdash^{CoC} \Delta}{\Delta \vdash^{CoC} \mathbf{Type} : \mathbf{Kind}} \qquad \frac{\vdash^{CoC} \Delta \qquad (x:A) \in \Delta}{\Delta \vdash^{CoC} x : A}$$

$$\frac{\Delta \vdash^{CoC} t : \Pi x : A.B \qquad \Delta \vdash^{CoC} u : A}{\Delta \vdash^{CoC} tu : B[x/u]}$$

$$\frac{\Delta(x:A) \vdash^{CoC} t : B \qquad \Delta \vdash^{CoC} \Pi x : A.B : s}{\Delta \vdash^{CoC} \lambda x : A.t : \Pi x : A.B}$$

$$\frac{\Delta \vdash^{CoC} A : s_1 \qquad \Delta(x:A) \vdash^{CoC} B : s_2}{\Delta \vdash^{CoC} \Pi x : A.B : s_2}$$

$$\frac{\Delta \vdash^{CoC} t : A \qquad \Delta \vdash^{CoC} B : s \qquad A \equiv_\beta B}{\Delta \vdash^{CoC} t : B}$$

Figure 2.9: Typing rules for the Calculus of Constructions.

Global Context

For each sort $s \in \{\mathbf{Type}, \mathbf{Kind}\}$, we declare a type constant $\mathtt{U}_s$ (the *universe* associated with s) and a *decoding* function ϵ_s to see the elements of $\mathtt{U}_s$ as types:

$\mathtt{U}_{\mathbf{Type}}$: **Type**. (written $\mathtt{U}_\mathtt{T}$ for short)
$\mathtt{U}_{\mathbf{Kind}}$: **Type**. (written $\mathtt{U}_\mathtt{K}$ for short)
$\epsilon_{\mathbf{Type}}$: $\mathtt{U}_\mathtt{T} \longrightarrow$ **Type**. (written $\epsilon_\mathtt{T}$ for short)
$\epsilon_{\mathbf{Kind}}$: $\mathtt{U}_\mathtt{K} \longrightarrow$ **Type**. (written $\epsilon_\mathtt{K}$ for short)

We add the constant `type` of type $\mathtt{U}_\mathtt{K}$ as the representative of **Type** in the universe $\mathtt{U}_\mathtt{K}$:

`type` : $\mathtt{U}_\mathtt{K}$.

We add representatives for the different product types in the universe $\mathtt{U}_\mathtt{T}$ and $\mathtt{U}_\mathtt{K}$:

$\dot{\Pi}_{(T,T)}$: $\Pi x : \mathtt{U}_\mathtt{T}.(((\epsilon_\mathtt{T}\ x) \longrightarrow \mathtt{U}_\mathtt{T}) \longrightarrow \mathtt{U}_\mathtt{T})$.
$\dot{\Pi}_{(T,K)}$: $\Pi x : \mathtt{U}_\mathtt{T}.(((\epsilon_\mathtt{T}\ x) \longrightarrow \mathtt{U}_\mathtt{K}) \longrightarrow \mathtt{U}_\mathtt{K})$.
$\dot{\Pi}_{(K,T)}$: $\Pi x : \mathtt{U}_\mathtt{K}.(((\epsilon_\mathtt{K}\ x) \longrightarrow \mathtt{U}_\mathtt{T}) \longrightarrow \mathtt{U}_\mathtt{T})$.
$\dot{\Pi}_{(K,K)}$: $\Pi x : \mathtt{U}_\mathtt{K}.(((\epsilon_\mathtt{K}\ x) \longrightarrow \mathtt{U}_\mathtt{K}) \longrightarrow \mathtt{U}_\mathtt{K})$.

Finally, we add rewrite rules to relate sorts and product seen as objects (whose type is a universe) and seen as types (universes):

$\epsilon_\mathtt{K}(\mathtt{type}) \hookrightarrow \mathtt{U}_\mathtt{T}$.
$\epsilon_\mathtt{T}(\dot{\Pi}_{(T,T)}\ x\ y) \hookrightarrow \Pi z : (\epsilon_\mathtt{T}\ x).\epsilon_\mathtt{T}\ (y\ z)$.
$\epsilon_\mathtt{K}(\dot{\Pi}_{(T,K)}\ x\ y) \hookrightarrow \Pi z : (\epsilon_\mathtt{T}\ x).\epsilon_\mathtt{K}\ (y\ z)$.
$\epsilon_\mathtt{T}(\dot{\Pi}_{(K,T)}\ x\ y) \hookrightarrow \Pi z : (\epsilon_\mathtt{K}\ x).\epsilon_\mathtt{T}\ (y\ z)$.
$\epsilon_\mathtt{K}(\dot{\Pi}_{(K,K)}\ x\ y) \hookrightarrow \Pi z : (\epsilon_\mathtt{K}\ x).\epsilon_\mathtt{K}\ (y\ z)$.

Let Γ be this global context.

Lemma 2.7.17. Γ *is a strongly well-formed.*

Proof. Trivial. □

Embedding

We now give two encodings for the terms of the Calculus of Construction. The first one is an encoding as objects.

Definition 2.7.18 (Translation as Object)**.**

$$
\begin{array}{lcll}
|x| & = & x & \\
|\mathbf{Type}| & = & \texttt{type} & \\
|\Pi x : A.B| & = & \dot{\Pi}_{(T,T)}\ |A|\ (\lambda x : (\epsilon_{\mathtt{T}}\ |A|).|B|) & \textit{if } A \textit{ and } B \textit{ are types} \\
|\Pi x : A.B| & = & \dot{\Pi}_{(T,K)}\ |A|\ (\lambda x : (\epsilon_{\mathtt{T}}\ |A|).|B|) & \textit{if } A \textit{ is a type and } B \textit{ is a kind} \\
|\Pi x : A.B| & = & \dot{\Pi}_{(K,T)}\ |A|\ (\lambda x : (\epsilon_{\mathtt{K}}\ |A|).|B|) & \textit{if } A \textit{ is a kind and } B \textit{ is a type} \\
|\Pi x : A.B| & = & \dot{\Pi}_{(K,K)}\ |A|\ (\lambda x : (\epsilon_{\mathtt{K}}\ |A|).|B|) & \textit{if } A \textit{ and } B \textit{ are kinds} \\
|\lambda x : A.t| & = & \lambda x : (\epsilon_{\mathtt{T}}|A|).|t| & \textit{if } A \textit{ is a type} \\
|\lambda x : A.t| & = & \lambda x : (\epsilon_{\mathtt{K}}|A|).|t| & \textit{if } A \textit{ is a kind} \\
|t\,u| & = & |t|\ |u| &
\end{array}
$$

The second one is an encoding as types.

Definition 2.7.19 (Translation as Type)**.**

$$
\begin{array}{lcll}
\|A\| & = & \epsilon_s|A| & \textit{if } A \textit{ has type } s. \\
\|\mathbf{Kind}\| & = & \mathtt{U}_{\mathtt{K}} &
\end{array}
$$

This encoding extends to typing contexts in the obvious way:

Definition 2.7.20 (Translation of Contexts)**.**

$$
\begin{array}{lcl}
\|\emptyset\| & = & \emptyset \\
\|\Sigma(x : A)\| & = & \|\Sigma\|(x : \|A\|)
\end{array}
$$

Soundness and Conservativity

Let Γ be the global context defined in the previous section. We have the following correspondences between terms of the Calculus of Constructions and their encoding.

Theorem 2.7.21 (β-reduction [CD07])**.** *If* $t_1 \to_\beta t_2$ *then* $|t_1| \to_\beta |t_2|$.

Theorem 2.7.22 (Termination [Dow15])**.** *The relation* $\to_{\beta\Gamma}$ *is strongly normalizing on well-typed terms.*

Theorem 2.7.23 (Soundness [CD07])**.** *If* $\Sigma \vdash^{CoC} t : B$*, then* $\Gamma \vdash^{ctx} \|\Sigma\|$ *and* $\Gamma; \|\Sigma\| \vdash |t| : \|B\|$.

Theorem 2.7.24 (Convervativity [CD07])**.** *If* $\Gamma; \|\Sigma\| \vdash N : \|B\|$ *and* $\Gamma \vdash^{ctx} \|\Sigma\|$*, then there exists* M *such that then* $\Sigma \vdash^{CoC} M : B$.

2.7.3 Heyting Arithmetic

In this section we embed Heyting Arithmetic, following Dowek and Werner [DW05].

Definition

Definition 2.7.25 (Signature of Heyting Arithmetic)**.** *The signature Σ_{HA} of Heyting arithmetic contains the following symbols:*

- *the predicate symbol $=$ of arity 2;*
- *the function symbol O of arity 0;*
- *the function symbol S of arity 1;*
- *and the function symbols $+$ and $*$ of arity 2.*

Definition 2.7.26 (Heyting Arithmetic)**.** *A proposition P built from the signature Σ_{HA} is provable under the hypotheses Ξ in Heyting arithmetic if P is provable in constructive predicate logic under the assumptions Ξ and the following propositions (axioms):*

- *for all proposition P with free variable x:*
  $\forall n.\forall m.n = m \implies P[x/n] \implies P[x/m]$;
- $\forall n.\neg(S(n) = 0)$;
- $\forall n.\forall m.S(n) = S(m) \implies n = m$;
- $\forall n.0 + n = n$;
- $\forall n.\forall m.S(n) + m = S(n + m)$;
- $\forall n.0 * n = 0$;
- $\forall n.\forall m.S(n) * m = n * m + m$;
- $P[x/0] \implies (\forall n.P[x/n] \implies P[x/S(n)]) \implies (\forall n.P[x/n])$.

We write $\Xi \vdash_{HA} P$ if P is provable under the hypotheses Ξ in Heyting arithmetic.

Global Context

Since Heyting Arithmetic is a theory of constructive predicate logic, we extend the context Γ defined in Section 2.7.1.

We add Peano integers.

`nat` : **Type**.
0 : `nat`.
`S` : `nat` $\longrightarrow$ `nat`.

We identify the terms and the naturals.

`term` $\hookrightarrow$ `nat`.

We add a notion of equality.

`eq` : `nat` $\longrightarrow$ `nat` $\longrightarrow$ `prop`.
`eq` 0 0 $\hookrightarrow$ `true`.
`eq` (`S` n_1) (`S` n_2) $\hookrightarrow$ `eq` n_1 n_2.
`eq` (`S` n) 0 $\hookrightarrow$ `false`.
`eq` 0 (`S` n) $\hookrightarrow$ `false`.

We define addition.

$\texttt{plus} : \texttt{nat} \longrightarrow \texttt{nat} \longrightarrow \texttt{nat}.$
$\texttt{plus}\ 0\ n \hookrightarrow n.$
$\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

We define multiplication.

$\texttt{mult} : \texttt{nat} \longrightarrow \texttt{nat} \longrightarrow \texttt{nat}.$
$\texttt{mult}\ 0\ n \hookrightarrow 0.$
$\texttt{mult}\ (S\ n_1)\ n_2 \hookrightarrow \texttt{plus}\ n_2\ (\texttt{mult}\ n_1\ n_2).$

Finally, we add an induction principle.

$\texttt{rec} : \Pi\, p : (\texttt{nat} \longrightarrow \texttt{prop}).\texttt{prf}\ (p\ 0) \longrightarrow (\Pi n : \texttt{nat}.\texttt{prf}\ (p\ n) \longrightarrow \texttt{prf}\ (p\ (\texttt{S}\ n))) \longrightarrow \Pi n : \texttt{nat}.\texttt{prf}\ (p\ n).$

Let Γ^{HA} be this global context. We abbreviate it by Γ when no confusion is possible.

Embedding

The encoding of terms and proposition of Heyting Arithmetic is the same as for any theory of constructive predicate logic (see Section 2.7.1).

Lemma 2.7.27. *Γ^{HA} is a strongly well-formed global context.*

Proof. Trivial. □

Soundness and Conservativity

Theorem 2.7.28 (Soundness)**.** *If $\Xi \vdash_{HA} P$ and $x_1, \ldots, x_n$ are the variables occurring free in Ξ and P, then there exists π such that, $\Gamma; \Delta \vdash \pi : |P|$, where $\Delta = (x_1 : \texttt{term}) \ldots (x_n : \texttt{term})\ |\Xi|$.*

Proof. Since we already have Theorem 2.7.10, it suffices to consider the cases where P is an axiom.

- If $P = \forall n.((S(n) = 0) \implies \bot)$, then $|P| = \Pi n : \texttt{nat}.|S(n) = 0| \longrightarrow |\bot|$ and, since $|S(n) = 0| \to_\Gamma |\bot|$, we take $\pi = \lambda n.\lambda x : \texttt{prf}\ (\texttt{eq}\ (\texttt{S}\ n)\ 0).x$.
- If $P = \forall n.\forall m.S(n) = S(m) \implies n = m$, then $|P| = \Pi n : \texttt{nat}.\Pi m : \texttt{nat}.|S(n) = S(m)| \longrightarrow |n = m|$. Since $|S(n) = S(m)| \to_\Gamma |n = m|$, we take $\pi = \lambda n : \texttt{nat}.\lambda m : \texttt{nat}.\lambda x : |S(n) = S(m)|.x$.
- If $P = \forall n.0 + n = n$ or $P = \forall n.0 * n = 0$, then $|P| = \Pi n : \texttt{nat}.|\top|$. We take $\pi = \lambda n : \texttt{nat}.\lambda x : \texttt{prop}.\lambda y : |x|.y$.
- If $P = \forall n.\forall m.S(n) + m = S(n + m)$ or $P = \forall n.\forall m.S(n) * m = n * m + m$, then $|P| = \Pi n : \texttt{nat}.\Pi m : \texttt{nat}.|\top|$. We take $\pi = \lambda n : \texttt{nat}.\lambda m : \texttt{nat}.\lambda x : \texttt{prop}.\lambda y : |x|.y$.
- If $P = Q[x/0] \implies (\forall n.Q[x/n] \implies Q[x/S(n)]) \implies (\forall n.Q[x/n])$, then we take $\pi = \texttt{rec}\ (\lambda x : \texttt{nat}.\|Q\|)$.

□

Theorem 2.7.29 (Conservativity)**.** *If there exist a term π and a local context Δ such that $\Gamma \vdash^{ctx} \Delta$ and $\Gamma; \Delta \vdash \pi : |P|$ with $\Delta = (x_1 : \texttt{term}) \ldots (x_n : \texttt{term})|\Xi|$, then $\Xi \vdash_{HA} P$ is provable.*

$\underline{x}, \underline{y}, \underline{z}$	$\in$	$\mathscr{V}_O$	(Object Variable)
X, Y, Z	$\in$	$\mathscr{V}_T$	(Type Variable)
$\underline{c}, \underline{f}$	$\in$	$\mathscr{C}_O$	(Object Constant)
C, F	$\in$	$\mathscr{C}_T$	(Type Constant)
$\underline{t}, \underline{u}, \underline{v}$	::=	$\underline{x} \mid \underline{c} \mid \underline{u}\ \underline{v} \mid \underline{u}\ \underline{V} \mid \lambda \underline{x} : \underline{U}.\underline{t}$ $\mid \lambda X : K.\underline{t}$	(Object)
$\underline{T}, \underline{U}, \underline{V}$	::=	$X \mid C \mid \underline{U}\ \underline{v} \mid \underline{U}\ \underline{V} \mid \lambda \underline{x} : \underline{U}.\underline{T}$ $\mid \lambda X : K.\underline{T} \mid \Pi \underline{x} : \underline{U}.\underline{T} \mid \Pi X : K.\underline{T}$	(Type)
K	::=	$\mathbf{Type} \mid \Pi \underline{x} : \underline{U}.K \mid \Pi X : K.K$	(Kind)
t, u, v	::=	$\underline{u} \mid \underline{U} \mid K \mid \mathbf{Kind}$	(Term)

Figure 2.10: Syntax for the terms of the Calculus of Constructions Modulo

Proof. We proceed as in Theorem 2.7.14. The case where π is a constant or an application headed by a constant is now possible if the constant is `rec`. In this case we proceed as for the variable case. □

2.8 The Calculus Of Constructions Modulo

The $\lambda\Pi$-calculus is one vertex of the λ-cube [B+91]. Extending the conversion of the $\lambda\Pi$-calculus with rewrite rules gives us the $\lambda\Pi$-Calculus Modulo. Extending the $\lambda\Pi$-calculus with polymorphism and type operators gives us the Calculus of Constructions. Mixing this two extensions we get a new system: the Calculus of Constructions Modulo.

2.8.1 Terms and Contexts

To be able to have polymorphic terms (objects depending on types) and type operators (types depending on types), we need to extend our definition of terms allowing types applied to types, types applied to objects, as well as abstractions over types.

Definition 2.8.1 (Term)**.** *An* object *is either an* object variable *in the set $\mathscr{V}_O$, or an* object constant *in the set $\mathscr{C}_O$, or an* application $\underline{u}\ v$ *where $\underline{u}$ is an objects and v is an object or a type, or an* abstraction over a type $\lambda \underline{x} : \underline{U}.\underline{t}$ *where $\underline{t}$ is an object and $\underline{U}$ is a type or an* abstraction over a kind $\lambda X : K.\underline{t}$ *where $\underline{t}$ is an object and K is a kind.*

A type *is either a* type variable *in the set $\mathscr{V}_T$, or a* type constant *in the set $\mathscr{C}_T$, or an* application $\underline{U}\ v$ *where $\underline{U}$ is a type and v is a type or an object, or an* abstraction over a type $\lambda \underline{x} : \underline{U}.\underline{V}$ *where $\underline{U}$ and $\underline{V}$ are types or an* abstraction over a kind $\lambda X : K.\underline{U}$ *where $\underline{U}$ is a type and K is a kind or a* product over a type $\Pi \underline{x} : \underline{U}.\underline{V}$ *where U and $\underline{V}$ are types or a* product over a kind $\Pi X : K.\underline{U}$ *where K is a kind and $\underline{U}$ is a type.*

A kind *is either a* product over a type $\Pi \underline{x} : \underline{U}.K$ *where $\underline{U}$ is a type and K is a kind A* kind *is either a* product over a kind $\Pi X : K_1.K_2$ *where K_1 and K_2 are kind or the symbol* **Type**.

A term *is either an object, a type, a kind or the symbol* **Kind**.

$$\Delta \quad ::= \quad \emptyset \mid \Delta(\underline{x} : \underline{T}) \mid \Delta(X : K) \qquad \text{(Local Context)}$$

Figure 2.11: Syntax for local contexts of the Calculus of Constructions Modulo

$$\textbf{(CoC-Product)} \quad \frac{\Gamma;\Delta \vdash A : s_1 \qquad \Gamma;\Delta(x : A) \vdash B : s_2}{\Gamma;\Delta \vdash \Pi x : A.B : s_2}$$

Figure 2.12: Product rule for the Calculus of Constructions Modulo

The sets $\mathcal{V}_O$, $\mathcal{V}_T$, $\mathscr{C}_O$ and $\mathscr{C}_T$ are assumed to be infinite and pairwise disjoint. The grammars for objects, types, kinds and terms are given Figure 2.10.

We also extend the notion of local context to cope with type variables.

Definition 2.8.2 (Local Context)**.** *A local context is a list of pairs of object variables together with a type or type variables together with a kind. The grammar for local contexts is given Figure 2.11.*

The notions of global context (Definition 2.2.6) and rewriting (Definition 2.3.3) do not change.

2.8.2 Type System

The only thing we need to modify to allow polymorphism and type operators is to generalize the **(Product)** rule for typing terms.

Definition 2.8.3 (Well-Typed Term)**.** *We say that a term t* has type *A in a global context Γ and a local context Δ if the judgment $\Gamma;\Delta \vdash t : A$ is derivable by the inference rules of Figure 2.4 replacing the rule* **(Product)** *by the rule* **(CoC-Product)** *of Figure 2.12.*

The typing of local contexts is updated to check type variable declarations.

Definition 2.8.4 (Well-Formed Local Context)**.** *A local context Δ is* well-formed *with respect to a global context Γ if the judgment $\Gamma \vdash^{ctx} \Delta$ is derivable by the inference rules of Figure 2.13.*

The definitions of well-typed global contexts (Definition 2.4.9) and strongly well-formed global context (Definition 2.4.10) do not change.

$$\textbf{(Empty)} \quad \frac{}{\Gamma \vdash^{ctx} \emptyset}$$

$$\textbf{(Type Declaration)} \quad \frac{\Gamma \vdash^{ctx} \Delta \qquad \Gamma;\Delta \vdash K : \textbf{Kind} \qquad X \notin dom(\Delta)}{\Gamma \vdash^{ctx} \Delta(X : K)}$$

$$\textbf{(Object Declaration)} \quad \frac{\Gamma \vdash^{ctx} \Delta \qquad \Gamma;\Delta \vdash \underline{U} : \textbf{Type} \qquad \underline{x} \notin dom(\Delta)}{\Gamma \vdash^{ctx} \Delta(\underline{x} : \underline{U})}$$

Figure 2.13: Typing rules for local contexts

2.8.3 Example

Polymorphism and type operators allow us defining polymorphic lists, that is lists parametrized by the type of their elements:

$\texttt{PList} : \textbf{Type} \longrightarrow \textbf{Type}$.
$\texttt{PNil} : \Pi X : \textbf{Type}.\texttt{PList}\ X$.
$\texttt{PCons} : \Pi X : \textbf{Type}.X \longrightarrow \texttt{PList}\ X \longrightarrow \texttt{PList}\ X$.

Remark that none of the three declarations above are allowed in the $\lambda\Pi$-Calculus Modulo, since they all quantify over **Type**. We can now define a function `Plength` computing the size of a polymorphic list:

$\texttt{Plength} : \Pi X : \textbf{Type}.\texttt{PList}\ X \longrightarrow \texttt{nat}$.
$\texttt{Plength}\ X\ (\texttt{PNil}\ X) \hookrightarrow 0$.
$\texttt{Plength}\ X\ (\texttt{PCons}\ X\ x\ l) \hookrightarrow \texttt{S}\ (\texttt{Plength}\ X\ l)$.

2.8.4 Properties

Most of the properties we have proved so far for the $\lambda\Pi$-Calculus Modulo continue to hold for the Calculus of Constructions Modulo (sometimes with minor modifications).

The inversion lemma (Lemma 2.6.1) has to be updated for the case where t is a product.

Lemma 2.8.5 (Inversion for the Calculus of Constructions Modulo)**.** *If* $\Gamma;\Delta \vdash t : T$ *then*

- *either* $t = \textbf{Type}$ *and* $T = \textbf{Kind}$
- *or* $t = x$ *and there exists* A *such that* $(x : A) \in \Delta$ *and* $T \equiv_{\beta\Gamma} A$.
- *or* $t = c$ *and* $T \equiv_{\beta\Gamma} \Gamma(c)$.
- *or* $t = f\,u$ *and there exist* A *and* B *such that* $\Gamma;\Delta \vdash f : \Pi x : A.B$, $\Gamma;\Delta \vdash u : A$ *and* $T \equiv_{\beta\Gamma} B[x/u]$.
- *or* $t = \lambda x : A.u$ *and there exist* B *and* s *such that* $\Gamma;\Delta \vdash \Pi x : A.B : s$, $\Gamma;\Delta(x : A) \vdash u : B$, *and* $T \equiv_{\beta\Gamma} \Pi x : A.B$.
- *or* $t = \Pi x : A.B$ *and there exist* s_1 *and* s_2 *such that* $\Gamma;\Delta \vdash A : s_1$, $\Gamma;\Delta(x : A) \vdash B : s_2$, *and* $T = s_2$.

Proof. By induction on the typing derivation. □

The following properties stated for the $\lambda\Pi$-Calculus Modulo still hold without a change and with the same proof for the Calculus of Constructions Modulo:

- **Kind** is not typable (Lemma 2.6.2);
- subterms of well-typed terms are well-typed (Lemma 2.6.3);
- local weakening (Lemma 2.6.4);
- the property of convertible local contexts (Lemma 2.6.8);

- the property of well-typed substitutions (Lemma 2.6.9);
- stratification (Lemma 2.6.10) (a few more cases have to be considered in the proof);
- product compatibility by confluence (Theorem 2.6.11);
- global weakening (Lemma 2.6.5);
- well-typed global declarations (Lemma 2.6.14);
- product compatibility preserved by object declarations (Lemma 2.6.12);
- product compatibility for strongly well-formed global contexts (Lemma 2.6.15);
- well-typedness of rules for strongly well-formed rewrite rules (Lemma 2.6.18);
- subject reduction (Theorem 2.6.22);
- uniqueness of typed (Theorem 2.6.25)
- and undecidablity of product compatibility (Lemma 2.6.36).

Inversion of local contexts has a slightly different statement:

Lemma 2.8.6 (Inversion for $\vdash^{ctx}$)**.** *If* $\Gamma \vdash^{ctx} \Delta(x : A)$*, then* $\Gamma \vdash^{ctx} \Delta$ *and* $\Gamma;\Delta \vdash A : s$ *for some* $s \in \{\mathbf{Type}, \mathbf{Kind}\}$.

Proof. By induction on the typing derivation. □

Lemma 2.8.7 (Well-typed Local Declaration)**.** *If* $\Gamma \vdash^{ctx} \Delta$ *and* $(x : A) \in \Delta$*, then* $\Gamma;\Delta \vdash A : s$.

Proof. By induction on the typing derivation. □

Finally Conjecture 2.6.13 can be proved for the Calculus of Constructions Modulo.

Lemma 2.8.8. *If product compatibility holds for* Γ*, then it also holds for* $\Gamma(\underline{C} : K)$.

Proof. Since we now have type variables we can adapt the proof of Lemma 2.6.12. □

2.8.5 Toward Pure Type Systems Modulo

A natural extension from there would be to consider an arbitrary pure type system and extend it with rewrite rules. We conjecture that most of the results of this chapter still hold for an arbitrary functional pure type system. However, in the $\lambda\Pi$-Calculus Modulo and the Calculus of Constructions Modulo, we use the syntactic classification of terms into objects, types and kinds to ensure that rewriting respects the stratification of terms. Such a classification does not exist a priori in pure type systems. Thus, extra assumptions may be needed.

2.9 Related Work

The first work on the combination of typed λ-calculus and Term Rewriting Systems is due to Breazu-Tannen [Tan88]. He showed that confluence is preserved when we combine the simply typed lambda-calculus with a confluent TRS. This work was soon extended to system F and to strong normalization by Breazu-Tannen and Gallier [TG89] and, independently, by Okada [Oka89]. Barbanera [Bar90] extended the result of strong normalization to the Calculus of Constructions (with β-conversion only) and Dougherty [Dou92] proved the preservation of confluence and strong normalization for any *stable* set of λ-terms.

An important step was reached by Jouannaud and Okada [JO91] with the introduction of the *General Schemata*, a criterion for strong normalization able to deal with higher-order rewrite rules, that is rewrite rules where the variables can have functional types. The General Schemata was successively adapted to deal with F^{ω} [BF93b], intersection types [BF93a], the systems of the λ-cube [BFG94] and Pure Type Systems [BG95].

A different and more powerful criterion called *Higher-Order Recursive Path Ordering (HORPO)* was then introduced by Jouannaud and A. Rubio [JR99] for simply typed lambda calculus with higher-order rewrite rules.

In 1999, Jouannaud, Okada and Blanqui [BJO99] extended the General Schema, keeping simply typed symbols, in order to deal with *strictly positive types*.

For formalizing the metatheory of the Calculus of Inductive Constructions in Coq, Barras [Bar99] defined a notion of pure type systems with operators.

In 2000, Walukiewicz [Wal03] extended HORPO to the Calculus of Constructions.

But these works have in common that rewriting is always confined to the object level. Blanqui [Bla05a, Bla04] worked on the termination in the *Calculus of Algebraic Constructions*, an extension of the Calculus of Constructions with object-level and type-level rewrite rules.

2.10 Conclusion

We have presented a new version of the $\lambda\Pi$-Calculus Modulo. It differs from the original presentation [CD07] by two aspects: it uses an untyped notion of reduction and it explicits the typing of the rewrite rules and makes it iterative. These two modifications clarify the relation between rewriting and typing and make the framework closer to its implementation in DEDUKTI.

We have provided a complete meta-theoretical study of the $\lambda\Pi$-Calculus Modulo which was lacking in its original presentation. In particular, we have emphasized the role played by two properties, product compatibility and well-typedness of rewrite rules, in the proofs of many basic results such as subject reduction and uniqueness of types. These properties will be studied further in the next chapters. We have also given several examples of encodings in the $\lambda\Pi$-Calculus Modulo. Finally, we have studied an extension of the $\lambda\Pi$-Calculus Modulo with polymorphism and type operators.

Chapter 3

Typing Rewrite Rules

Résumé Ce chapitre étudie la propriété de bon typage des règles de réécriture. Une règle de réécriture est bien typée si elle préserve le typage. Partant d'un critère simple, à savoir que le membre gauche de la règle doit être algébrique et les membres gauche et droit doivent avoir le même type, on généralise progressivement le résultat pour considérer des membres gauches non algébriques et mal typés. Cette généralisation est particulièrement importante en présence de types dépendants, pour permettre de conserver des règles de réécriture linéaires à gauche et préserver la confluence du système de réécriture. On donne aussi une caractérisation exacte de la notion de bon typage pour les règles de réécriture sous forme d'un problème d'unification et on prouve son indécidabilité.

3.1 Introduction

We have seen in the previous chapter that most properties of the $\lambda\Pi$-Calculus Modulo depend on two of them: product compatibility and well-typedness of rewrite rules. In this chapter we are interested in finding sufficient conditions to ensure the second property. The criterion that we have used so far is the following (for the proof see Section 3.2):

Definition 3.1.1 (Strongly Well-Formed Rewrite Rule)**.** *Let Γ be a global context such that $\rightarrow_{\beta\Gamma}$ is confluent. A rewrite rule $(u \hookrightarrow v)$ is strongly-well-formed in Γ if, for some local context Δ and term T,*

- *u is algebraic,*
- *$dom(\Delta) = FV(u)$,*
- *$\Gamma \vdash^{ctx} \Delta$,*
- *$\Gamma; \Delta \vdash u : T$ and*
- *$\Gamma; \Delta \vdash v : T$*

Theorem (Strongly Well-Formed Rewrite Rules are Permanently Well-Typed)**.** *Let Γ be a well-typed global context. If $(u \hookrightarrow v)$ is strongly well-formed in Γ, then it is permanently well-typed.*

This criterion is not entirely satisfactory for two reasons. First, the restriction to algebraic left-hand side is too strong. We would like to be able to type-check rewrite rules such as this one:

$\texttt{getCst} : (\texttt{nat} \longrightarrow \texttt{nat}) \longrightarrow \texttt{nat}.$
$\texttt{getCst}\ (\lambda x : \texttt{nat}.n) \hookrightarrow n.$

This rewrite rule extracts the value of a constant function. While it is well-typed (as we will see), it is not strongly well-formed because of the abstraction on the left-hand side.

Second, the restriction to well-typed left-hand sides is also too strong. This may be more surprising. This restriction often makes the criterion incompatible with another important property of the system: the confluence of the rewriting system (recall that confluence is our main tool to prove product compatibility). The reason is that, in presence of dependent types, strongly well-formed rewrite rules tend to be non left-linear. And β-reduction do not behave nicely with non left-linear rewrite rules (see Section 1.4.3).

Consider the following example. We want to define basic functions to manipulate vectors. We first define the type `Vector` of lists parametrized by their length.

$\texttt{term} : \textbf{Type}.$
$\texttt{Vector} : \texttt{nat} \longrightarrow \textbf{Type}.$
$\texttt{vnil} : \texttt{Vector}\ 0.$
$\texttt{vcons} : \Pi n : \texttt{nat}.\texttt{term} \longrightarrow \texttt{Vector}\ n \longrightarrow \texttt{Vector}\ (\texttt{S}\ n).$

`vnil` is the empty vector. `vcons` builds a vector of size $(\texttt{S}\ n)$ from an element of type `term` and a vector of size n. Let us define the function `head` to extract the first element of a non-empty vector.

$\texttt{head} : \Pi n : \texttt{nat}.\texttt{Vector}\ (\texttt{S}\ n) \longrightarrow \texttt{term}.$
$\texttt{head}\ n\ (\texttt{vcons}\ n\ e\ l) \hookrightarrow e.$

This last rewrite rule is strongly well-formed. However, it is not left-linear since the variable n occurs twice in the left-hand side. In fact, we can show that the resulting rewrite system is not confluent (Section 1.4.3).

We can wonder what happens if we replace the last rewrite rule by its *linearized* version where we use two different variables instead of two occurrences of the same variable:

$\texttt{head}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow e.$

The first consequence is that we immediately get back the confluence of the rewriting system since Theorem 1.4.7 applies. But is it still the rewrite rule we wanted to add? The short answer is yes. Indeed, both rewrite rules (the non left-linear rule and its linearized version) match the same redexes as soon as the redexes are well-typed. In other words, both rewrite rules have the same computational behaviour on well-typed terms. If the redex $r = \sigma(\texttt{head}\ n_1\ (\texttt{vcons}\ n_2\ e\ l))$ is well-typed (and normal) then it necessarily verifies $\sigma(n_1) = \sigma(n_2)$. Therefore, r is also a redex of the non-linear rule since $r = \sigma_2(\texttt{head}\ n\ (\texttt{vcons}\ n\ e\ l))$ for $\sigma_2 = \sigma \cup \{n \mapsto \sigma(n_1)\}$. By replacing the non-left-linear rule by its linearized version, we are able to recover confluence of the rewriting system without modifying the behaviour of the rule on *well-typed* terms. Moreover, we will see that this linearization is type-safe.

In this chapter, we build on these ideas to iteratively generalize the criterion for well-typedness of rewrite rules given above in order to allow non-algebraic and ill-

typed left-hand sides. In particular we justify the linearization of rewrite rules presented above. We conclude by a characterization of well-typedness for rewrite rules as a problem of inclusion between two sets of solutions of unifications problems and a proof of its undecidability.

3.2 Strongly Well-Formed Rewrite Rules

We begin by proving that strongly well-formed rewrite rules are permanently well-typed.

Theorem 3.2.1 (Strongly Well-Formed Rewrite Rules are Permanently Well-typed)**.** *Let Γ be a global context such that $\to_{\beta\Gamma}$ is confluent. If $(u \hookrightarrow v)$ is strongly well-formed in Γ, then it is permanently well-typed.*

Most of the proofs in this chapter will follow the same scheme. The idea is to prove that, if a redex $\sigma(u)$ is well-typed and u is algebraic then, first, the substitution σ is well-typed and, second, the redex has type $\sigma(T_0)$ where T_0 is the type of u. More precisely, from $\Gamma_0;\Delta_0 \vdash u : T_0$ and $\Gamma;\Delta \vdash \sigma(u) : T$, we prove $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$ and $T \equiv_{\beta\Gamma} \sigma(T_0)$. Subsequently, by the property of well-typed substitutions (Lemma 2.6.9) and by conversion, we can conclude from $\Gamma_0;\Delta_0 \vdash v : T_0$ that $\Gamma;\Delta \vdash \sigma(v) : T$.

For each theorem in the chapter, we prove a *main lemma* corresponding to the first part of the proof.

Lemma 3.2.2 (Main Lemma for Theorem 3.2.1)**.** *Let Γ_0 be a global context such that $\to_{\beta\Gamma_0}$ is confluent and let Γ be a well-typed extension of Γ_0. Assume that:*

- *t is algebraic,*
- *$\Gamma \vdash^{ctx} \Delta$,*
- *$\Gamma;\Delta \vdash \sigma(t) : T$*
- *and $\Gamma_0;\Delta_0 \vdash t : T_0$.*

Then, we have:

- *$T \equiv_{\beta\Gamma} \sigma(T_0)$,*
- *$\Gamma;\Delta \vdash \sigma(R) : s$ for some reduct R of T_0*
- *and, for all $x \in FV(t)$, $\Gamma;\Delta \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma} \sigma(\Delta_0(x))$.*

Proof. We proceed by induction on t.

- If $t = f$ is a constant, then $FV(t) = \emptyset$ and, by inversion, on the one hand, $T_0 \equiv_{\beta\Gamma_0} \Gamma_0(f)$ and, on the other hand, $T \equiv_{\beta\Gamma} \Gamma(f)$. Since $\sigma(\Gamma_0(f)) = \Gamma(f)$, we have $\sigma(T_0) \equiv_{\beta\Gamma} T$. By confluence of $\to_{\beta\Gamma_0}$, T_0 and $\Gamma_0(f)$ have a common reduct R. Since $\sigma(\Gamma_0(f)) \to_{\beta\Gamma} \sigma(R)$ and $\sigma(\Gamma_0(f)) = \Gamma(f)$ is well-typed, by subject reduction (Theorem 2.6.22), R is well-typed.
- If $t = u\,v$ with u and v algebraic, then, by inversion, on the one hand, $\Gamma_0;\Delta_0 \vdash u : \Pi x : A_0.B_0$, $\Gamma_0;\Delta_0 \vdash v : A_0$ and $T_0 \equiv_{\beta\Gamma_0} B_0[x/v]$. On the other hand, $\Gamma;\Delta \vdash \sigma(u) : \Pi x : A.B$, $\Gamma;\Delta \vdash \sigma(v) : A$ and $T \equiv_{\beta\Gamma} B[x/\sigma(v)]$.

By induction hypothesis on u, $\sigma(\Pi x : A_0.B_0) \equiv_{\beta\Gamma} \Pi x : A.B$, $\Gamma;\Delta \vdash \sigma(R) : s$ for some reduct R of $\Pi x : A_0.B_0$ and, for all $x \in FV(u)$, $\Gamma;\Delta \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma} \sigma(\Delta_0(x))$.

Let $R = \Pi x : A_2.B_2$, we have $\Gamma;\Delta \vdash \sigma(\Pi x : A_2.B_2) : s$, $\Gamma;\Delta \vdash \Pi x : A.B : s$ and $\sigma(\Pi x : A_2.B_2) \equiv_{\beta\Gamma} \Pi x : A.B$; therefore, by product-compatibility, $\sigma(A_2) \equiv_{\beta\Gamma} A$ and $\sigma(B_2) \equiv_{\beta\Gamma} B$.

It follows that $\sigma(T_0) \equiv_{\beta\Gamma} \sigma(B_0[x/v]) \equiv_{\beta\Gamma} \sigma(B_2[x/v]) \equiv_{\beta\Gamma} B[x/\sigma(v)] \equiv_{\beta\Gamma} T$.

Moreover, we have $B_2[x/v]$ is a reduct of T_0 and $\Gamma;\Delta \vdash \sigma(B_2[x/v]) : s$.

By induction hypothesis on v, for all $x \in FV(v)$, $\Gamma;\Delta \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma} \sigma(\Delta_0(x))$.

- If $t = ux$ with u algebraic and x a variable, then, by inversion, on the one hand, $\Gamma_0;\Delta_0 \vdash u : \Pi y : A_0.B_0$, $\Gamma_0;\Delta_0 \vdash x : A_0$, $A_0 \equiv_{\beta\Gamma_0} \Delta_0(x)$, and $T_0 \equiv_{\beta\Gamma_0} B_0[y/x]$. On the other hand, $\Gamma;\Delta \vdash \sigma(u) : \Pi y : A.B$, $\Gamma;\Delta \vdash \sigma(x) : A$, and $T \equiv_{\beta\Gamma} B[y/\sigma(x)]$.

  By induction hypothesis on u, $\sigma(\Pi y : A_0.B_0) \equiv_{\beta\Gamma} \Pi y : A.B$, $\Gamma;\Delta \vdash \sigma(R) : s$ for some reduct R of T_0 and, for all $z \in FV(u)$, $\Gamma;\Delta \vdash \sigma(z) : T_z$ with $T_z \equiv_{\beta\Gamma} \sigma(\Delta_0(z))$.

  Let $R = \Pi y : A_2.B_2$, we have $\Gamma;\Delta \vdash \sigma(\Pi y : A_2.B_2) : s$, $\Gamma;\Delta \vdash \Pi y : A.B : s$ and $\sigma(\Pi y : A_2.B_2) \equiv_{\beta\Gamma} \Pi y : A.B$; therefore, by product-compatibility, $\sigma(A_2) \equiv_{\beta\Gamma} A$ and $\sigma(B_2) \equiv_{\beta\Gamma} B$.

  It follows that $T \equiv_{\beta\Gamma} B[y/\sigma(x)] \equiv_{\beta\Gamma} \sigma(B_2)[y/\sigma(x)] = \sigma(B_2[y/x]) \equiv_{\beta\Gamma} \sigma(B_0[y/x]) \equiv_{\beta\Gamma} \sigma(T_0)$.

  To type $\sigma(x)$, we can take $T_x = A$, since $\Gamma;\Delta \vdash \sigma(x) : A$ with $A \equiv_{\beta\Gamma} \sigma(A_0) \equiv_{\beta\Gamma} \sigma(\Delta_0(x))$.

  B_2 is a reduct of B_0 and, by inversion, $\sigma(B_2)$ is well-typed.

□

It follows that the substitution σ is well-typed.

Lemma 3.2.3. *Let Γ be a global context and Δ_0 be a local context well-formed in Γ. If, for all $x \in dom(\Delta_0)$, $\Gamma;\Delta \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma} \sigma(\Delta_0(x))$, then $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.*

Proof. By induction on $\Gamma \vdash^{ctx} \Delta_0$.

- **(Empty Local Context)** Trivial.
- **(Variable Declaration)** Suppose that, $\Delta_0 = \Delta_2(x : A)$, $\Gamma \vdash^{ctx} \Delta_2$ and $\Gamma;\Delta_2 \vdash A :$ **Type**.

  By induction hypothesis, $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$, that is to say, for all $x \in dom(\Delta_2)$, $\Gamma;\Delta \vdash \sigma(x) : \sigma(\Delta_0(x))$.

  We have $\Gamma;\Delta \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma} \sigma(A)$. Moreover, by Lemma 2.6.9, we have $\Gamma;\Delta \vdash \sigma(A) :$ **Type**. It follows, by conversion, that $\Gamma;\Delta \vdash \sigma(x) : \sigma(A)$.

  Thus, $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

□

We can now prove the theorem:

Proof of Theorem 3.2.1. Let $(u \hookrightarrow v)$ be a strongly-well-formed rewrite rule for Γ_0 and let Γ be a well-typed extension of Γ_0. Assume that, for some Δ, σ and T, we have $\Gamma \vdash^{ctx} \Delta$ and $\Gamma; \Delta \vdash \sigma(u) : T$.

By hypothesis, for some Δ_0 and T_0, $\Gamma_0; \Delta_0 \vdash u : T_0$ and $\Gamma_0; \Delta_0 \vdash v : T_0$. By the main lemma (Lemma 3.2.2), $T \equiv_{\beta\Gamma} \sigma(T_0)$ and, for all $x \in FV(u) = dom(\Delta_0)$, $\Gamma; \Delta \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma} \sigma(\Delta_0(x))$. By Lemma 3.2.3, $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

Finally, by the property of well-typed substitutions (Lemma 2.6.9), we have $\Gamma; \Delta \vdash \sigma(v) : \sigma(T_0)$. It follows, by conversion, that $\Gamma; \Delta \vdash \sigma(v) : T$. □

3.3 Left-Hand Sides Need not be Algebraic

We now try to weaken the assumption of algebraicity for left-hand side in Theorem 3.2.1. First, remark that this assumption cannot be dropped completely. We have

$\Gamma; (x : \mathtt{nat} \longrightarrow \mathtt{nat})(y : \mathtt{nat}) \vdash \mathtt{S}(x\ y) : \mathtt{nat}$ and
$\Gamma; (x : \mathtt{nat} \longrightarrow \mathtt{nat})(y : \mathtt{nat}) \vdash x\ (\mathtt{S}\ y) : \mathtt{nat}$.

However, the following rewrite rule is not well-typed.

$\mathtt{S}\ (x\ y) \hookrightarrow x\ (\mathtt{S}\ y)$.

Indeed, using this rewrite rule, we have

$\mathtt{S}\ ((\lambda x : \mathtt{prop.0})\ \mathtt{true}) \rightarrow (\lambda x : \mathtt{prop.0})\ (\mathtt{S}\ \mathtt{true})$.

While $\mathtt{S}\ ((\lambda x : \mathtt{prop.0})\ \mathtt{true})$ has type $\mathtt{nat}$, $(\lambda x : \mathtt{prop.0})\ (\mathtt{S}\ \mathtt{true})$ is obviously ill-typed.

The assumption of algebraicity is used to ensure that we cannot substitute the variables with terms of the *wrong* type without losing well-typedness. In fact, the types of the free variables of a well-typed algebraic term are uniquely defined. Moreover, the type of an algebraic term as well as the types of its free variables can be inferred. This is basically what is used in the proof of Lemma 3.2.2. This property is not specific to algebraic terms and it is closely related to type inference.

Algorithms able to reconstruct the type of a term with only partial information about the type of the free variables exist. They are usually based on bidirectional type systems. In these systems, the typing rules are split in two groups: typing rules to synthesize (infer) the type of a term and typing rules to check that a term has a given type.

Instead of requiring that the left-hand side of a rewrite rule is algebraic, we can require that its type as well as the types of its free variables can be inferred by some bidirectional typing system defined below.

Definition 3.3.1. *The relations $\Vdash_i$ and $\Vdash_c$ are defined inductively from the inference rules of Figure 3.1.*

The judgment $\Gamma; \Delta_1; \Sigma \Vdash_i t \Rightarrow T, \Delta_2$ is an inference judgment. It means: knowing the types of the variables in Δ_1 and Σ, we can synthesize the type T for t as well as the types of the variables in Δ_2. The variables in Δ_1 and Δ_2 are variables that are globally free in the term being explored (that is, the variables that will be matched) and the variables in Σ are variables that are bound.

The judgment $\Gamma; \Delta_1; \Sigma \Vdash_c t \Leftarrow T \mid \Delta_2$ is a checking judgment. It means: knowing the types of the variables in Δ_1 and Σ, we can check that t has type T and synthesize

(**Sort**)
$$\frac{}{\Gamma;\Delta;\Sigma \Vdash_i \mathbf{Type} \Rightarrow \mathbf{Kind}, \Delta}$$

(**Constant**)
$$\frac{(f : A) \in \Gamma}{\Gamma;\Delta;\Sigma \Vdash_i f \Rightarrow A, \Delta}$$

(**Σ-Variable**)
$$\frac{(x : A) \in \Sigma}{\Gamma;\Delta;\Sigma \Vdash_i x \Rightarrow A, \Delta}$$

(**Δ-Variable**)
$$\frac{(x : A) \in \Delta}{\Gamma;\Delta;\Sigma \Vdash_i x \Rightarrow A, \Delta}$$

(**S-Application**)
$$\frac{\Gamma;\Delta_1;\Sigma \Vdash_i u \Rightarrow T, \Delta_2 \qquad T \rightarrow^*_{\beta\Gamma} \Pi x : A.B \qquad \Gamma;\Delta_2;\Sigma \Vdash_c v \Leftarrow A \mid \Delta_3}{\Gamma;\Delta_1;\Sigma \Vdash_i u\,v \Rightarrow B[x/u], \Delta_3}$$

(**S-Abstraction**)
$$\frac{\Gamma;\Delta_1;\Sigma \Vdash_c A \Leftarrow \mathbf{Type} \mid \Delta_2 \qquad \Gamma;\Delta_2;\Sigma(x : A) \Vdash_i u \Rightarrow B, \Delta_3}{\Gamma;\Delta_1;\Sigma \Vdash_i \lambda x : A.u \Rightarrow \Pi x : A.B, \Delta_3}$$

(**Product**)
$$\frac{\Gamma;\Delta_1;\Sigma \Vdash_c A \Leftarrow \mathbf{Type} \mid \Delta_2 \qquad \Gamma;\Delta_2;\Sigma(x : A) \Vdash_c B \Leftarrow s \mid \Delta_3}{\Gamma;\Delta_1;\Sigma \Vdash_i \Pi x : A.B \Rightarrow s, \Delta_3}$$

(**Free Variable**)
$$\frac{x \notin dom(\Delta\Sigma) \qquad FV(A) \cap dom(\Sigma) = \emptyset}{\Gamma;\Delta;\Sigma \Vdash_c x \Leftarrow A \mid \Delta(x : A)}$$

(**Inversion**)
$$\frac{\Gamma;\Delta_1;\Sigma \Vdash_i u \Rightarrow A_2, \Delta_2 \qquad A_1 \equiv_{\beta\Gamma} A_2}{\Gamma;\Delta_1;\Sigma \Vdash_c u \Leftarrow A_1 \mid \Delta_2}$$

(**C-Abstraction**)
$$\frac{\begin{array}{ll} T \rightarrow^*_{\beta\Gamma} \Pi x : A_2.B & \Gamma;\Delta_1;\Sigma \Vdash_c A_1 \Leftarrow \mathbf{Type} \mid \Delta_2 \\ A_1 \equiv_{\beta\Gamma} A_2 & \Gamma;\Delta_2;\Sigma(x : A_1) \Vdash_c u \Leftarrow B \mid \Delta_3 \end{array}}{\Gamma;\Delta_1;\Sigma \Vdash_c \lambda x : A_1.u \Leftarrow T \mid \Delta_3}$$

(**C-Application**)
$$\frac{(x : A) \in \Sigma \qquad \Gamma;\Delta_1;\Sigma \Vdash_c u \Leftarrow \Pi x : A.B \mid \Delta_2}{\Gamma;\Delta_1;\Sigma \Vdash_c u\ x \Leftarrow B \mid \Delta_2}$$

Figure 3.1: Bidirectional typing rules for rewrite rules

a type for the variables in Δ_2.

Remark 3.3.2. *The typing system of Figure 3.1 is very close to an algorithm able to infer the type of a term together with the types of its free variables. Indeed, for each term, at most one inference rule of $\Vdash_c$ or $\Vdash_i$ can be applied. Moreover, we can see $\Vdash_c$ and $\Vdash_i$ as two mutually recursively defined functions. $\Vdash_i$ is a function taking two local contexts Δ_1 and Σ as well as a term t as arguments and producing a term T and a local context Δ_2 such that the judgment $\Gamma;\Delta_1;\Sigma \Vdash_i t \Rightarrow T,\Delta_2$ is derivable. $\Vdash_c$ is a function taking two local contexts Δ_1 and Σ and two terms t and T as arguments and produces a local context Δ_2 such that the judgment $\Gamma;\Delta_1;\Sigma \Vdash_c t \Leftarrow T \mid \Delta_2$ is derivable. Assuming that the side conditions (for instance the congruence) are decidable, this gives us an algorithm to infer the type of a term when the type of some free variables is missing.*

Theorem 3.3.3. *Let Γ be a global context. If, for some Δ and T, we have:*

- $\Gamma;\emptyset;\emptyset \Vdash_i u \Rightarrow T,\Delta$,
- *and* $\Gamma;\Delta \vdash v : T$,

then $(u \hookrightarrow v)$ is permanently well-typed in Γ.

As previously, we first prove a main lemma.

Lemma 3.3.4 (Main Lemma for Theorem 3.3.3)**.** *Let Γ_0 be a global context and Γ be a well-formed extension of Γ_0. Assume that:*

- $\Gamma \vdash^{ctx} \Delta\sigma(\Sigma)$,
- $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(t) : T$,
- $dom(\Sigma) \cap dom(\sigma) = \emptyset$,
- $dom(\Sigma) \cap codom(\sigma) = \emptyset$
- *and* $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta$.

We have

- *if* $\Gamma_0;\Delta_1;\Sigma \Vdash_i t \Rightarrow T_0,\Delta_0$ *then*
  - $T \equiv_{\beta\Gamma} \sigma(T_0)$,
  - $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$ *or* $T_0 =$ **Kind**
  - *and* $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$*;*
- *if* $\Gamma_0;\Delta_1;\Sigma \Vdash_c t \Leftarrow T_0 \mid \Delta_0$, $T \equiv_{\beta\Gamma} \sigma(T_0)$ *and* $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, *then* $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

Proof. We proceed by induction on $\Gamma_0;\Delta_1;\Sigma \Vdash_i t \Rightarrow T_0,\Delta_0$ and $\Gamma_0;\Delta_1;\Sigma \Vdash_c t \Leftarrow T_0 \mid \Delta_0$.

- Case $\Gamma_0;\Delta_1;\Sigma \Vdash_i t \Rightarrow T_0,\Delta_0$.
  - **(Sort)** By inversion on $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(t) : T$, we have $T =$ **Kind** $= \sigma(T_0)$.
  - **(Constant)** By inversion, $T \equiv_{\beta\Gamma} \Gamma(f) = \Gamma_0(f) = \sigma(\Gamma_0(f)) = \sigma(T_0)$.
  - **(Σ-Variable)** By inversion, $T \equiv_{\beta\Gamma} \sigma(\Sigma(x)) = \sigma(T_0)$. Since $\Delta\sigma(\Sigma)$ is well-formed, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\Sigma(x)) :$ **Type**.

– **(Δ-Variable)** By hypothesis, we have $\Gamma;\Delta \vdash \sigma(x) : \sigma(\Delta_1(x))$, $\Delta_0 = \Delta_1$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(x) : T$.

  We have $\Gamma;\Delta \vdash \sigma(x) : \sigma(\Delta_1(x))$, and, by weakening, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(x) : \sigma(\Delta_1(x))$.

  Therefore, by uniqueness of types, $\sigma(\Delta_1(x)) \equiv_{\beta\Gamma} T$.

  Finally, by stratification, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\Delta_1(x)) : \mathbf{Type}$.

– **(S-Application)** Suppose that $t = uv$, $T_0 = B_0[x/v]$, $\Gamma_0;\Delta_1;\Sigma \Vdash_i u \Rightarrow P, \Delta_2$, $P \rightarrow^*_{\beta\Gamma} \Pi x : A_0.B_0$ and $\Gamma_0;\Delta_2;\Sigma \Vdash_c v \Leftarrow A \mid \Delta_0$.

  By inversion, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(u) : \Pi x : A.B$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(v) : A$ and $T \equiv_{\beta\Gamma} B[x/\sigma(v)]$.

  By induction hypothesis on u, we have $\sigma(\Pi x : A_0.B_0) \equiv_{\beta\Gamma} \Pi x : A.B, \Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\Pi x : A_0.B_0) : s$, and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By product-compatibility of Γ, $\sigma(A_0) \equiv_{\beta\Gamma} A$ and $\sigma(B_0) \equiv_{\beta\Gamma} B$.

  By induction hypothesis on v, $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

  Moreover, $\sigma(T_0) = \sigma(B_0[x/v]) \equiv_{\beta\Gamma} B[x/\sigma(v)] \equiv_{\beta\Gamma} T$.

– **(S-Abstraction)** Suppose that $t = \lambda x : A_0.u$, $T_0 = \Pi x : A_0.B_0$, $\Gamma_0;\Delta_1;\Sigma \Vdash_c A_0 \Leftarrow \mathbf{Type} \mid \Delta_2$ and $\Gamma_0;\Delta_2;\Sigma(x : A_0) \Vdash_i u \Rightarrow B_0, \Delta_0$.

  By inversion, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(A_0) : \mathbf{Type}$, $\Gamma;\Delta\sigma(\Sigma(x : A_0)) \vdash \sigma(u) : B$ and $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).B$.

  By induction hypothesis on A_0, $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By induction hypothesis on u, $\sigma(B_0) \equiv_{\beta\Gamma} B$ and $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

  Thus, $\sigma(T_0) = \sigma(\Pi x : A_0.B_0) \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).B \equiv_{\beta\Gamma} T$.

– **(Product)** Suppose that $t = \Pi x : A.B$, $T_0 = s$, $\Gamma_0;\Delta_1;\Sigma \Vdash_c A \Leftarrow \mathbf{Type} \mid \Delta_2$ and $\Gamma_0;\Delta_2;\Sigma(x : A) \Vdash_c B \Leftarrow s \mid \Delta_0$.

  By inversion, $T = s$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(A) : \mathbf{Type}$ and $\Gamma;\Delta\sigma(\Sigma(x : A)) \vdash \sigma(B) : s$.

  By induction hypothesis on A, $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By induction hypothesis on B, $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

• Case $\Gamma_0;\Delta_1;\Sigma_0 \Vdash_c t \Leftarrow T_0 \mid \Delta_0$, $T \equiv_{\beta\Gamma} \sigma(T_0)$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$.

– **(C-Abstraction)** Suppose that $t = \lambda x : A_1.u$, $\sigma(T_0) \equiv_{\beta\Gamma} T$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, $T_0 \rightarrow^*_{\beta\Gamma} \Pi x : A_2.B_0$, $A_1 \equiv_{\beta\Gamma} A_2$, $\Gamma_0;\Delta_1;\Sigma \Vdash_c A_1 \Leftarrow \mathbf{Type} \mid \Delta_2$ and $\Gamma_0;\Delta_2;\Sigma(x : A_1) \Vdash_c u \Leftarrow B_0 \mid \Delta_0$.

  By inversion, $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_1).B$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(A_1) : \mathbf{Type}$ and $\Gamma;\Delta\sigma(\Sigma(x : A_1)) \vdash \sigma(u) : B$.

  Since $\sigma(\Pi x : A_2.B_0) \equiv_{\beta\Gamma} \sigma(T_0) \equiv_{\beta\Gamma} T \equiv_{\beta\Gamma} \Pi x : \sigma(A_1).B$, we have, by product-compatibility, $\sigma(B_0) \equiv_{\beta\Gamma} B$.

  By induction hypothesis on A_1, $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By induction hypothesis on u, $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

– **(C-Application)** Suppose that $t = ux$, $(x : A_0) \in \Sigma$ and $\Gamma_0;\Delta_1;\Sigma \Vdash_c u \Leftarrow \Pi x : A_0.T_0 \mid \Delta_0$.

  By inversion, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(u) : \Pi x : A.B$, $\Gamma;\Delta\sigma(\Sigma) \vdash x : A$, $A \equiv_{\beta\Gamma} \sigma(\Sigma(x)) = \sigma(A_0)$ and $T \equiv_{\beta\Gamma} B$.

  We have $\sigma(\Pi x : A_0.T_0) = \Pi x.\sigma(A_0).\sigma(T_0) \equiv_{\beta\Gamma} \Pi x : A.B$ and, by induction hypothesis, $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

– **(Free Variable)** Suppose that $t = x$ and $x \notin dom(\Delta_1\Sigma)$ and $\Delta_0 = \Delta_1(x : T_0)$. Since we have $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta$, to prove $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$, it suffices to show that $\Gamma; \Delta \vdash \sigma(x) : \sigma(T_0)$.

  We have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(x) : T$. Since $\sigma(T_0)$ is well-typed, by conversion, we have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(x) : \sigma(T_0)$.

  Since $dom(\Sigma) \cap codom(\sigma) = \emptyset$ and $FV(T_0) \cap dom(\Sigma) = \emptyset$, we have $\Gamma; \Delta \vdash \sigma(x) : \sigma(T_0)$.

– **(Inversion)** By induction hypothesis.

□

Using the main lemma, we can prove Theorem 3.3.3.

Proof of Theorem 3.3.3. Let Γ be a well-typed extension of Γ_0 and let $(u \hookrightarrow v)$ be a rewrite rule such that $\Gamma_0; \emptyset; \emptyset \Vdash_i u \Rightarrow T_0, \Delta_0$ and $\Gamma_0; \Delta_0 \vdash v : T_0$.

Assume that, for some Δ, σ and T, we have $\Gamma \vdash^{ctx} \Delta$ and $\Gamma; \Delta \vdash \sigma(u) : T$. By global weakening, we have $\Gamma; \Delta_0 \vdash v : T_0$.

By the main lemma (Lemma 3.3.4), $T \equiv_{\beta\Gamma} \sigma(T_0)$ and $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

Finally, by Lemma 2.6.9, we have $\Gamma; \Delta \vdash \sigma(v) : \sigma(T_0)$. It follows, by conversion, that $\Gamma; \Delta \vdash \sigma(v) : T$. □

As a corollary, we get that the first rewrite rule from the beginning of this chapter is well-typed.

Corollary 3.3.5. *The rewrite rule* $(\texttt{getCst}\ (\lambda x : \texttt{nat}.n) \hookrightarrow n)$ *is permanently well-typed.*

Proof. Using the rules **(Constant)**, **(Inversion)**, **(C-Abstraction)** and **(Free Variable)** we have:

$$\dfrac{\dfrac{\dfrac{(\texttt{nat} : \textbf{Type}) \in \Gamma}{\Gamma; \emptyset; \emptyset \Vdash_i \texttt{nat} \Rightarrow \textbf{Type}, \emptyset}}{\Gamma; \emptyset; \emptyset \Vdash_c \texttt{nat} \Leftarrow \textbf{Type} \mid \emptyset} \qquad \dfrac{n \neq x}{\Gamma; \emptyset; (x : \texttt{nat}) \Vdash_c n \Leftarrow \texttt{nat} \mid (n : \texttt{nat})}}{\Gamma; \emptyset; \emptyset \Vdash_c (\lambda x : \texttt{nat}.n) \Leftarrow \texttt{nat} \longrightarrow \texttt{nat} \mid (n : \texttt{nat})}$$

It follows, by the rule **(S-Application)**

$$\dfrac{\begin{array}{c}\Gamma; \emptyset; \emptyset \Vdash_i \texttt{getCst} \Rightarrow (\Pi x : \texttt{nat}.\texttt{nat}) \longrightarrow \texttt{nat}, \emptyset \\ \Gamma; \emptyset; \emptyset \Vdash_c (\lambda x : \texttt{nat}.n) \Leftarrow \texttt{nat} \longrightarrow \texttt{nat} \mid (n : \texttt{nat})\end{array}}{\Gamma; \emptyset; \emptyset \Vdash_i \texttt{getCst}\ (\lambda x : \texttt{nat}.n) \Rightarrow \texttt{nat}, (n : \texttt{nat})}$$

Moreover, we have $\Gamma; (n : \texttt{nat}) \vdash n : \texttt{nat}$. Therefore, by Theorem 3.3.3, the rewrite rule $(\texttt{getCst}\ (\lambda x : \texttt{nat}.n) \hookrightarrow n)$ is permanently well-typed. □

Chapter 4 also contains many examples of rewrite rules with non-algebraic left-hand sides that can be shown well-typed using Theorem 3.3.3.

$$\textbf{(Abs-No-Check)}\quad \frac{T \to^*_{\beta\Gamma} \Pi x : A_2.B \qquad \Gamma;\Delta_2;\Sigma(x:A_1) \Vdash^2_c u \Leftarrow B \mid \Delta_3}{\Gamma;\Delta_1;\Sigma \Vdash^2_c \lambda x : A_1.u \Leftarrow T \mid \Delta_3}$$

$$\textbf{(Inv-No-Check)}\quad \frac{\Gamma;\Delta_1;\Sigma \Vdash^2_i u \Rightarrow A_2,\Delta_2}{\Gamma;\Delta_1;\Sigma \Vdash^2_c u \Leftarrow A_1 \mid \Delta_2}$$

$$\textbf{(App-No-Check)}\quad \frac{\Gamma;\Delta_1;\Sigma \Vdash^2_i v \Rightarrow A,\Delta_2}{\Gamma;\Delta_1;\Sigma \Vdash^2_c u\ v \Leftarrow B \mid \Delta_2}$$

$$\textbf{(No-Check)}\quad \frac{}{\Gamma;\Delta_1;\Sigma \Vdash^2_c u \Leftarrow B \mid \Delta_1}$$

Figure 3.2: Additional typing rules for rewrite rules

3.4 Left-Hand Sides Need not be Well-Typed

The bidirectional type system of Figure 3.1 is sound with respect to $\vdash$.

Theorem 3.4.1. *Let Γ be a well-typed global context.*
If $\Gamma \vdash^{ctx} \Delta_1\Sigma$ and $\Gamma;\Delta_1;\Sigma \Vdash_i t \Rightarrow T,\Delta_2$, then $\Gamma \vdash^{ctx} \Delta_2\Sigma$ and $\Gamma;\Delta_2\Sigma \vdash t : T$.

Proof. By induction on $\Gamma;\Delta_1;\Sigma \Vdash_i t \Rightarrow T,\Delta_2$. □

Surprisingly, Theorem 3.4.1 is not used in the proof of Theorem 3.3.3. The assumption that the redex is well-typed is sufficient. Looking closely at the proof of Lemma 3.3.4, we remark that some premises of the rules of $\Vdash_i$ and $\Vdash_c$ are never used. In particular, in the rules **(Inversion)** and **(C-Abstraction)**, the hypothesis $A_1 \equiv_{\beta\Gamma} A_2$ is not used.

Since we are not interested in being sound with respect to $\vdash$, they can be removed. Moreover, we can make additional improvements. For instance, if some variable is not used in the right-hand side of a rewrite rule, it is not necessary to be able to infer its type since it will not be used. More generally, all the parts of the left-hand side that do not give any new information about the type of the term or the types of the free variables occurring in the right-hand side do not need to be inspected.

These considerations lead us to some improvements of the relations $\Vdash_i$ and $\Vdash_c$.

Definition 3.4.2. *The relations $\Vdash^2_i$ and $\Vdash^2_c$ are defined inductively from the typing rules of Figure 3.1 and Figure 3.2.*

$\Vdash^2_c$ features two new rules **(Inv-No-Check)** and **(Abs-No-Check)** similar to **(Inversion)** and **(C-Abstraction)** but without the unnecessary conversion test. We also add the rules **(No-Check)** and **(App-No-Check)** allowing to skip inspecting some subterm when its type (or the type of a superterm) is already known.

Using this new type system, Theorem 3.3.3 can be generalized one step further.

Theorem 3.4.3. *Let Γ be a global context. If $\Gamma;\emptyset;\emptyset \Vdash^2_i u \Rightarrow T,\Delta$ and $\Gamma;\Delta \vdash v : T$, then $(u \hookrightarrow v)$ is permanently well-typed.*

As usual, we first prove the corresponding main lemma. We will use the following notation.

Notation 3.4.4. *Let Δ be a local context and σ be a substitution. We write $\sigma(\Delta)$ for the context defined as follows:*

$$\begin{array}{lcll} \sigma(\emptyset) & = & \emptyset & \\ \sigma(\Delta_0(x:A)) & = & \sigma(\Delta_0) & \text{if } x \in dom(\sigma) \\ \sigma(\Delta_0(x:A)) & = & \sigma(\Delta_0)(x:\sigma(A)) & \text{if } x \notin dom(\sigma) \end{array}$$

Lemma 3.4.5. *Let Γ_0 be a global context and Γ be a well-typed extension of Γ_0. Assume that:*

- $\Gamma \vdash^{ctx} \Delta\sigma(\Sigma)$,
- $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(t) : T$,
- $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta$,
- $dom(\Sigma) \cap dom(\sigma) = \emptyset$
- *and* $dom(\Sigma) \cap codom(\sigma) = \emptyset$.

We have

- *if* $\Gamma_0; \Delta_1; \Sigma_1 \Vdash^2_i t \Rightarrow T_0, \Delta_0$, *then*
  - $T \equiv_{\beta\Gamma} \sigma(T_0)$,
  - $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$ *or* $T_0 =$ **Kind**,
  - $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$,
- *if* $\Gamma_0; \Delta_1; \Sigma_1 \Vdash^2_c t \Leftarrow T_0 \mid \Delta_0$, $T \equiv_{\beta\Gamma} \sigma(T_0)$ *and* $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, *then*
  - $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$,

Proof. As in the proof of Lemma 3.3.4, we proceed by induction on $\Gamma_0; \Delta_1; \Sigma_1 \Vdash_i t \Rightarrow T_0, \Delta_0$ and $\Gamma_0; \Delta_1; \Sigma_1 \Vdash_c t \Leftarrow T_0 \mid \Delta_0$. We only consider the new cases.

- **(Abs-No-Check)** Same as **(C-Abstraction)**.
- **(Inv-No-Check)** Same as **(Inversion)**.
- **(App-No-Check)** By induction hypothesis.
- **(No-Check)** Trivial.

□

Proof of Theorem 3.4.3. Same as Theorem 3.3.3 but using Lemma 3.4.5 instead of Lemma 3.3.4. □

We are now able to prove that the second example from the beginning of this chapter is well-typed.

Corollary 3.4.6. *The rewrite rule* (`head` n_1 (`vcons` n_2 e l) $\hookrightarrow$ e) *is permanently well-typed.*

Proof. Using the rules **(Constant)**, **(S-Application)** and **(Free Variable)**, we have, on the one hand,

$\Gamma;\emptyset;\emptyset \Vdash^2_i \texttt{head}\ n_1 \Rightarrow \texttt{Vector}\ (\texttt{S}\ n_1) \longrightarrow \texttt{term}, \Delta_1$

for $\Delta_1 = (n_1 : \texttt{nat})$ and, on the other hand,

$\Gamma;\Delta_1;\emptyset \Vdash^2_i \texttt{vcons}\ n_2\ e\ l \Rightarrow \texttt{Vector}\ (\texttt{S}\ n_2), \Delta_2$

for $\Delta_2 = (n_1 : \texttt{nat})(n_2 : \texttt{nat})(e : \texttt{term})(l : \texttt{Vector}\ n_2)$.
By the **(Inv-No-Check)** rule, we have

$\Gamma;\Delta_1;\emptyset \Vdash^2_c \texttt{vcons}\ n_2\ e\ l \Leftarrow \texttt{Vector}\ (\texttt{S}\ n_1) \mid \Delta_2$.

Remark that this rule permits changing the type of $\texttt{vcons}\ n_2\ e\ l$ from $\texttt{Vector}\ (\texttt{S}\ n_2)$ to $\texttt{Vector}\ (\texttt{S}\ n_1)$.
Finally, by the rule **(S-Application)**, we have

$\Gamma;\emptyset;\emptyset \Vdash^2_i \texttt{head}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \Rightarrow \texttt{term}, \Delta_2$.

On the other hand, we have $\Gamma;\Delta_2 \vdash e : \texttt{term}$. Therefore, by Theorem 3.4.3, the rewrite rule $(\texttt{head}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow e)$ is permanently well-typed. □

3.5 Taking Advantage of Typing Constraints

We have justified the linearization of the rewrite rule for the function `head`, which extracts the first element of a vector. Let us see if we can do the same for the function `tail`, which removes the first element of a vector.

$\texttt{tail} : \Pi n : \texttt{nat}.\texttt{Vector}\ (\texttt{S}\ n) \longrightarrow \texttt{Vector}\ n$.
$\texttt{tail}\ n\ (\texttt{vcons}\ n\ e\ l) \hookrightarrow l$.

The linearized version would be:

$\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow l$.

We have

$\Gamma;\emptyset;\emptyset \Vdash^2_i \texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \Rightarrow \texttt{Vector}\ n_1, \Delta$
and $\Gamma;\Delta \vdash l : \texttt{Vector}\ n_2$
for $\Delta = (n_1 : \texttt{nat})(n_2 : \texttt{nat})(e : \texttt{term})(l : \texttt{Vector}\ n_2)$.

But we cannot use Theorem 3.4.3, since $\texttt{Vector}\ n_1$ is not convertible with $\texttt{Vector}\ n_2$.

However, we can see that any substitution σ such that the redex $\sigma(\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l))$ is well-typed will verify the constraint $\sigma(\texttt{Vector}\ (\texttt{S}\ n_1)) \equiv_{\beta\Gamma} \sigma(\texttt{Vector}\ (\texttt{S}\ n_2))$ and, therefore, will verify $\sigma(n_1) \equiv_{\beta\Gamma} \sigma(n_2)$ and $\sigma(\texttt{Vector}\ n_1) \equiv_{\beta\Gamma} \sigma(\texttt{Vector}\ n_2)$. It follows that the rewrite rule is in fact well-typed.

In the previous section, we basically ignored the typing constraints (conversion tests) arising during the typechecking of the left-hand side of the rewrite rules. The example above shows that we need to extract information from them. This leads us to a new generalisation of Theorem 3.4.3.

We adapt the typing rules of Figure 3.2 to take into account the conversion tests and record them. The new inference relation, not only synthesizes the type of the left-hand side together with the types of its free variables, but also records the typing constraints that a well-typed redex will satisfy.

(**Sort**)
$$\frac{}{\Gamma;\Delta;\Sigma;\mathscr{C} \Vdash_i \mathbf{Type} \Rightarrow (\Delta, \mathbf{Kind}, \mathscr{C})}$$

(**Constant**)
$$\frac{(f : A) \in \Gamma}{\Gamma;\Delta;\Sigma;\mathscr{C} \Vdash_i f \Rightarrow (\Delta, A, \mathscr{C})}$$

(**Σ-Variable**)
$$\frac{(x : A) \in \Sigma}{\Gamma;\Delta;\Sigma;\mathscr{C} \Vdash_i x \Rightarrow (\Delta, A, \mathscr{C})}$$

(**Δ-Variable**)
$$\frac{(x : A) \in \Delta}{\Gamma;\Delta;\Sigma;\mathscr{C} \Vdash_i x \Rightarrow (\Delta, A, \mathscr{C})}$$

(**S-Application**)
$$\frac{\begin{array}{l}\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_i u \Rightarrow (\Delta_2, T_2, \mathscr{C}_2) \\ \Gamma;\Delta_2;\Sigma;\mathscr{C}_2 \Vdash_c v \Leftarrow A \mid (\Delta_3, \mathscr{C}_3) \qquad T_2 \rightarrow^*_{\beta\Gamma} \Pi x : A.B\end{array}}{\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_i u\ v \Rightarrow (\Delta_3, B[x/v], \mathscr{C}_3)}$$

(**S-Abstraction**)
$$\frac{\begin{array}{c}\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_c A \Leftarrow \mathbf{Type} \mid (\Delta_2, \mathscr{C}_2) \\ \Gamma;\Delta_2;\Sigma(x : A);\mathscr{C}_2 \Vdash_i u \Rightarrow (\Delta_3, B, \mathscr{C}_3)\end{array}}{\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_i \lambda x : A.u \Rightarrow (\Delta_3, \Pi x : A.B, \mathscr{C}_3)}$$

(**Product**)
$$\frac{\begin{array}{c}\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_c A \Leftarrow \mathbf{Type} \mid (\Delta_2, \mathscr{C}_2) \\ \Gamma;\Delta_2;\Sigma(x : A);\mathscr{C}_2 \Vdash_c B \Leftarrow s \mid (\Delta_3, \mathscr{C}_3)\end{array}}{\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_i \Pi x : A.B \Rightarrow (\Delta_3, s, \mathscr{C}_3)}$$

Figure 3.3: Bidirectional typing rules for pseudo-well-formed rewrite rules (Part 1: Synthesis)

3.5.1 Typing Constraints

Definition 3.5.1 (Constraints)**.** *A set of constraints is a set of pairs of terms.*

A set of constraints is also a unification problem modulo the relation $\equiv_{\beta\Gamma}$.

Definition 3.5.2 (Solutions of Constraints)**.** *Let Γ be a global context, V, a set of (implicitly bound) variables, and $\mathscr{C}$, a set of constraints.*

The set $Sol_\Gamma(V, \mathscr{C})$ of solutions of $\mathscr{C}$ in Γ is the set of substitutions σ such that:

- *$dom(\sigma) \subset V$*
- *and, for all $(A, B) \in \mathscr{C}$, $\sigma(A) \equiv_{\beta\Gamma} \sigma(B)$.*

The most general solution (MGS) (or most general unifier) is a classical notion of unification theory. A solution τ is most general when it subsumes all other solutions in the following sense: if σ is another solution, then $\sigma \equiv \sigma_0\tau$ for some σ_0. For our purpose, we also require a MGS to be idempotent and not to introduce new variables, which is equivalent to the following definition.

(**Free Variable**)
$$\frac{FV(A) \cap dom(\Sigma) = \emptyset \qquad x \notin dom(\Delta_1) \cup dom(\Sigma)}{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c x \Leftarrow A \mid (\Delta_1(x : A), \mathscr{C}_1)}$$

(**C-Abstraction**)
$$\frac{\begin{array}{c} T \to^*_{\beta\Gamma} \Pi x : A_2.B \\ \Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c A_1 \Leftarrow \mathbf{Type} \mid (\Delta_2, \mathscr{C}_2) \\ \Gamma; \Delta_2; \Sigma(x : A_1); \mathscr{C}_2 \Vdash_c u \Leftarrow B \mid (\Delta_3, \mathscr{C}_3) \end{array}}{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c \lambda x : A_1.u \Leftarrow T \mid (\Delta_3, \mathscr{C}_3 \cup \{(\lambda\Sigma.A_1, \lambda\Sigma.A_2)\})}$$

(**C-Application**)
$$\frac{(x : A) \in \Sigma \qquad \Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c u \Leftarrow \Pi x : A.B \mid (\Delta_2, \mathscr{C}_2)}{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c u\ x \Leftarrow B \mid (\Delta_2, \mathscr{C}_2)}$$

(**Inversion**)
$$\frac{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_i u \Rightarrow (\Delta_2, A_2, \mathscr{C}_2)}{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c u \Leftarrow A_1 \mid (\Delta_2, \mathscr{C}_2 \cup \{(\lambda\Sigma.A_1, \lambda\Sigma.A_2)\})}$$

(**App-No-Check**)
$$\frac{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_i v \Rightarrow (\Delta_2, A, \mathscr{C}_2)}{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c u\ v \Leftarrow B \mid (\Delta_2, \mathscr{C}_2)}$$

(**No-Check**)
$$\frac{}{\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c u \Leftarrow T \mid (\Delta_1, \mathscr{C}_1)}$$

Figure 3.4: Bidirectional typing rules for pseudo-well-formed rewrite rules (Part 2: Checking)

Definition 3.5.3 (Most General Solution (MGS))**.** *Let Γ be a global context, V be a set of variables and $\mathscr{C}$ be a set of constraints.*

A substitution $\tau \in Sol_\Gamma(V,\mathscr{C})$ is a most general solution *(MGS) for $\mathscr{C}$ in Γ if, for any $\sigma \in Sol_\Gamma(V,\mathscr{C})$, $\sigma \equiv_{\beta\Gamma} \sigma\tau$ on V.*

Since we are doing unification modulo $\equiv_{\beta\Gamma}$, MGSs do not always exist. For instance, the equation $\lambda x : A.y\ x \equiv_{\beta\Gamma} \lambda x : A.c\ x$ has two incomparable solutions $\{y \mapsto c\}$ and $\{y \mapsto \lambda x : A.c\ x\}$. Therefore, we introduce the weaker notion of pre-solution. A pre-solution is more general than any solution but need not be a solution itself.

Definition 3.5.4 (Pre-Solution)**.** *Let Γ be a global context, V be a set of variables and $\mathscr{C}$ be a set of constraints.*

A pre-solution *for $\mathscr{C}$ is a substitution τ such that, for any $\sigma \in Sol_\Gamma(V,\mathscr{C})$, $\sigma \equiv_{\beta\Gamma} \sigma\tau$ on V.*

We write $PreSol_\Gamma(V,\mathscr{C})$ the set of pre-solutions for $\mathscr{C}$ in Γ.

Remark 3.5.5. *The identity substitution is a pre-solution, for any set of constraints. Therefore, contrarily to the set of MGSs, the set of pre-solutions is never empty.*

If we take the set of constraints $\mathscr{C} = \{(\texttt{Vector}\ (\texttt{S}\ n_1), \texttt{Vector}\ (\texttt{S}\ n_2))\}$ and the set of variables $V = \{n_1, n_2\}$, then, by confluence of the reduction $\rightarrow_{\beta\Gamma}$, we know that, if $\sigma \in Sol_\Gamma(V,\mathscr{C})$, then $\sigma\ n_1 \equiv_{\beta\Gamma} \sigma\ n_2$. As a consequence, we have that the following substitutions are pre-solutions:

- $\{n_1 \mapsto n_2\}$,
- $\{n_2 \mapsto n_1\}$.

Since they are in $Sol_\Gamma(V,\mathscr{C})$, they are also MGSs.

3.5.2 Fine-Grained Typing of Rewrite Rules

We now generalize Theorem 3.4.3. We use the notion of pre-solution to take advantage of the information extracted from typing constraints obtained by typing the left-hand side of the rewrite rule, when typing the right-hand side.

Notation 3.5.6. *If t a term and Σ is a local context, then we write $\lambda\Sigma.t$ for the term $\lambda\vec{x} : \vec{A}.t$ where $\vec{x} = dom(\Sigma)$ and $A_i = \Sigma(x_i)$.*

Definition 3.5.7. *The relations $\Vdash_i$ and $\Vdash_c$ are defined inductively from the typing rules of Figure 3.3, and Figure 3.4.*

The inference rules for $\Vdash_i$ and $\Vdash_c$, are the same as for $\Vdash^2_i$ and $\Vdash^2_c$ except that, instead of ignoring the typing constraints, we record them.

Theorem 3.5.8. *Let Γ be a well-formed global context.*

If there exist Δ, T and τ such that

- $\Gamma;\emptyset;\emptyset;\emptyset \Vdash_i u \Rightarrow (\Delta, T, \mathscr{C})$,
- $\tau \in PreSol_\Gamma(dom(\Delta), \mathscr{C})$,
- $\Gamma \vdash^{ctx} \tau(\Delta)$
- *and* $\Gamma;\tau(\Delta) \vdash v : \tau(T)$,

then $(u \hookrightarrow v)$ *is well-typed in* Γ.

Remark 3.5.9. *If we take the identity substitution,* $\tau = id$*, for the pre-solution, then we get back exactly the assumptions of Theorem 3.4.3.*

As usual, we first prove the corresponding main lemma.

Lemma 3.5.10 (Main Lemma for Theorem 3.5.8)**.** *Let* Γ *a well-typed global context. Suppose that:*

- $\Gamma \vdash^{ctx} \Delta\sigma(\Sigma)$,
- $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(t) : T$,
- $\sigma \in Sol_\Gamma(dom(\Delta_1), \mathscr{C}_1)$,
- $dom(\Sigma) \cap dom(\sigma) = \emptyset$,
- $dom(\Sigma) \cap codom(\sigma) = \emptyset$
- *and* $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta$.

Then, we have:

- *if* $\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_i t \Rightarrow (\Delta_0, T_0, \mathscr{C}_0)$*, then*
  - $T \equiv_{\beta\Gamma} \sigma(T_0)$,
  - $T =$ **Kind** *or* $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$,
  - $\sigma \in Sol_\Gamma(dom(\Delta_0), \mathscr{C}_0)$
  - *and* $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$*;*
- *if* $\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c t \Leftarrow T_0 \mid (\Delta_0, \mathscr{C}_0)$ *and* $T \equiv_{\beta\Gamma} \sigma(T_0)$ *with* $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$*, then*
  - $\sigma \in Sol_\Gamma(dom(\Delta_0), \mathscr{C}_0)$
  - *and* $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

Proof. We prove both properties at the same time by induction on $\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_i t \Rightarrow (\Delta_0, T_0, \mathscr{C}_0)$ and $\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_c t \Leftarrow T_0 \mid (\Delta_0, \mathscr{C}_0)$.

- **(Sort)** $t =$ **Type**. By inversion, $T =$ **Kind** $= \sigma($**Kind**$) = \sigma(T_0)$.
- **(Constant)** $t = f$ is a constant. By inversion, $T \equiv_{\beta\Gamma} \Gamma(f) = \sigma(\Gamma(f)) = \sigma(T_0)$.
- **(Σ-Variable)** $t = x \in dom(\Sigma)$ is a variable and $T_0 = \Sigma(x)$. By inversion, $T \equiv_{\beta\Gamma} \sigma(\Sigma(x)) = \sigma(T_0)$.
- **(Δ-Variable)** $t = x \in dom(\Delta_1)$ is a variable and $T_0 = \Delta_1(x)$. We have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(x) : T$ and $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(x) : \sigma(\Delta_1(x))$.

  By uniqueness of types, $T \equiv_{\beta\Gamma} \sigma(\Delta_1(x)) = \sigma(T_0)$.

- **(S-Application)** Suppose that $t = uv$, $T_0 = B_2[x/v]$, $\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_i u \Rightarrow (\Delta_2, P, \mathscr{C}_2)$, $\Gamma;\Delta_2;\Sigma;\mathscr{C}_2 \Vdash_c v \Leftarrow A_2 \mid (\Delta_3, \mathscr{C}_3)$ and $P \rightarrow^*_{\beta\Gamma} \Pi x : A_2.B_2$.

  By inversion, we deduce $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(u) : \Pi x : A.B$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(v) : A$ and $T \equiv_{\beta\Gamma} B[x/\sigma(v)]$.

  By induction hypothesis on u, we have, $\sigma(P) \equiv_{\beta\Gamma} \Pi x : A.B$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(P) : s$, $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_2)$ and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By stratification, $\Gamma;\Delta\sigma(\Sigma) \vdash \Pi x : A.B : s$. By subject reduction, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\Pi x : A_2.B_2) : s$. Thus, from $\sigma(\Pi x : A_2.B_2) \equiv_{\beta\Gamma} \Pi x : A.B$, by product compatibility, we deduce $\sigma(A_2) \equiv_{\beta\Gamma} A$ and $\sigma(B_2) \equiv_{\beta\Gamma} B$.

  By induction hypothesis on v, $\sigma \in Sol(dom(\Delta_3), \mathscr{C}_3)$ and $\sigma : \Delta_3 \rightsquigarrow_\Gamma \Delta$.

  Moreover, $T \equiv_{\beta\Gamma} B[x/\sigma(v)] \equiv_{\beta\Gamma} \sigma(B_2)[x/\sigma(v)] = \sigma(T_0)$.

  Finally, since, by inversion, $\Gamma;\Delta\sigma(\Sigma)(x : \sigma(A_2)) \vdash \sigma(B_2) : s$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(v) : \sigma(A_2)$, we have, by substitution, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$.

- **(S-Abstraction)** Suppose that $t = \lambda x : A_0.u$, $T_0 = \Pi x : A_0.B_0$, $\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_c A_0 \Leftarrow \mathbf{Type} \mid (\Delta_2, \mathscr{C}_2)$ and $\Gamma;\Delta_2;\Sigma(x : A_0);\mathscr{C}_2 \Vdash_i u \Rightarrow (\Delta_3, B_0, \mathscr{C}_3)$.

  By inversion, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(A_0) : \mathbf{Type}$, $\Gamma;\Delta\sigma(\Sigma(x : A_0)) \vdash \sigma(u) : B$, $B \neq \mathbf{Kind}$ and $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).B$.

  By induction hypothesis on A_0, we have, $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_2)$ and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By induction hypothesis on u, $\sigma(B_0) \equiv_{\beta\Gamma} B$, $\Gamma;\Delta\sigma(\Sigma)(x : \sigma(A_0)) \vdash \sigma(B_0) : s$, $\sigma \in Sol(dom(\Delta_3), \mathscr{C}_3)$ and $\sigma : \Delta_3 \rightsquigarrow_\Gamma \Delta$.

  Finally, $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).B \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).\sigma(B_0) = \sigma(T_0)$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\Pi x : A_0.B_0) : s$.

- **(Product)** Suppose that $t = \Pi x : A_0.B_0$, $T_0 = s$, $\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_c A_0 \Leftarrow \mathbf{Type} \mid (\Delta_2, \mathscr{C}_2)$ and $\Gamma;\Delta_2;\Sigma(x : A_0);\mathscr{C}_2 \Vdash_c B_0 \Leftarrow s \mid (\Delta_3, \mathscr{C}_3)$.

  By inversion, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(A_0) : \mathbf{Type}$, $\Gamma;\Delta\sigma(\Sigma(x : A_0)) \vdash \sigma(B_0) : s$ and $T = s$.

  By induction hypothesis on A_0, we have $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_2)$ and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

  By induction hypothesis on u, $\sigma \in Sol(dom(\Delta_3), \mathscr{C}_3)$ and $\sigma : \Delta_3 \rightsquigarrow_\Gamma \Delta$.

  Finally, $T = s = \sigma(T_0)$.

- **(Free Variable)** Suppose that $t = x$, $\sigma(T_0) \equiv_{\beta\Gamma} T$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, $x \notin dom(\Delta_1) \cup dom(\Sigma)$ and $\Delta_0 = \Delta_1(x : T_0)$.

  We have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(x) : T$ and, by conversion, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(x) : \sigma(T_0)$.

  Since $dom(\Sigma) \cap codom(\sigma) = \emptyset$ and $FV(T_0) \cap dom(\Sigma) = \emptyset$, we also have $\Gamma;\Delta \vdash \sigma(x) : \sigma(T_0)$.

  Therefore, since $\sigma : \Delta_1 \rightsquigarrow_\Gamma \Delta$, we also have $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

- **(C-Abstraction)** Suppose that $t = \lambda x : A_1.u$, $\sigma(T_0) \equiv_{\beta\Gamma} T$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, $T_0 \rightarrow^*_{\beta\Gamma} \Pi x : A_0.B_0$, $\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_c A_1 \Leftarrow \mathbf{Type} \mid (\Delta_2, \mathscr{C}_2)$ and $\Gamma;\Delta_2;\Sigma(x : A_1);\mathscr{C}_2 \Vdash_c u \Leftarrow B_0 \mid (\Delta_3, \mathscr{C}_3)$.

  From $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\lambda x : A_1.u) : T$, by inversion, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(A_1) : \mathbf{Type}$, $\Gamma;\Delta\sigma(\Sigma(x : A_1)) \vdash \sigma(u) : B$, $B \neq \mathbf{Kind}$ and $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_1).B$.

Therefore, we have $\Pi x : \sigma(A_1).B \equiv_{\beta\Gamma} \sigma(\Pi x : A_0.B_0)$. Since $\Gamma;\Delta\sigma(\Sigma) \vdash \Pi x : \sigma(A_1).B : s$ and, by subject reduction, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(\Pi x : A_0.B_0) : s$, we have, by product compatibility, $\sigma(A_1) \equiv_{\beta\Gamma} \sigma(A_0)$ and $B \equiv_{\beta\Gamma} \sigma(B_0)$.

By induction hypothesis on A_1, $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_2)$ and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

By induction hypothesis on u, $\sigma \in Sol(dom(\Delta_3), \mathscr{C}_3)$ and $\sigma : \Delta_3 \rightsquigarrow_\Gamma \Delta$.

Moreover, since $\mathscr{C}_0 = \mathscr{C}_3 \cup \{(\lambda\Sigma.A_1, \lambda\Sigma.A_0)\}$, we have $\sigma \in Sol(\Delta_3, \mathscr{C}_0)$.

- **(C-Application)** Suppose that $t = u\ x$, $\sigma(T_0) \equiv_{\beta\Gamma} T$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, $(x : A_0) \in \Sigma$ and $\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_c u \Leftarrow \Pi x : A_0.T_0 \mid (\Delta_2, \mathscr{C}_2)$.

  From $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(u\ x) : T$, by inversion, we deduce $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(u) : \Pi x : A.B$ $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(x) : A$ and $T \equiv_{\beta\Gamma} B$. Moreover, since $x \notin dom(\sigma)$, we have $\sigma(x) = x$ and $A \equiv_{\beta\Gamma} \sigma(\Sigma(x)) = \sigma(A_0)$.

  Therefore, we have $\Pi x : A.B \equiv_{\beta\Gamma} \sigma(\Pi x : A_0.T_0)$.

  By induction hypothesis, we have, $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_2)$ and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$.

- **(Inversion)** Suppose that we have $T \equiv_{\beta\Gamma} \sigma(T_0)$, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_0) : s$, $\mathscr{C}_0 = \mathscr{C}_2 \cup \{(\lambda\Sigma.T_0, \lambda\Sigma.T_2)\}$ and $\Gamma;\Delta_1;\Sigma;\mathscr{C}_1 \Vdash_i t \Rightarrow (\Delta_2, T_2, \mathscr{C}_2)$.

  By induction hypothesis, $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_2)$, and $\sigma : \Delta_2 \rightsquigarrow_\Gamma \Delta$, $T \equiv_{\beta\Gamma} \sigma(T_2)$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(T_2) : s$.

  Since $\sigma(T_0) \equiv_{\beta\Gamma} T \equiv_{\beta\Gamma} \sigma(T_2)$, we have $\sigma \in Sol(dom(\Delta_2), \mathscr{C}_0)$.

- **(App-No-Check)** By induction hypothesis.

- **(No-Check)** Trivial.

□

We can now prove Theorem 3.5.8.

Proof of Theorem 3.5.8. Let Γ be a well-formed global context.

Suppose that $\Gamma;\emptyset;\emptyset,\emptyset \Vdash_i u \Rightarrow (\Delta_0, T_0, \mathscr{C})$, $\tau \in PreSol_\Gamma(dom(\Delta_0), \mathscr{C})$, $\Gamma \vdash^{ctx} \Delta_0$ and $\Gamma;\tau(\Delta_0) \vdash v : \tau(T_0)$.

Suppose that, for some Δ, σ and T, we have $\Gamma;\Delta \vdash \sigma(u) : T$ and $\Gamma \vdash^{ctx} \Delta$. Since $FV(u) \cup FV(v) \subset dom(\Delta)$, we can assume *wlog* that $dom(\sigma) \subset dom(\Delta)$.

By Lemma 3.5.10, $\sigma \in Sol_\Gamma(dom(\Delta_0), \mathscr{C})$ and $\sigma : \Delta_0 \rightsquigarrow_\Gamma \Delta$.

Therefore, we have, for all $x \in dom(\tau(\Delta_0))$, $\Gamma;\Delta \vdash \sigma(x) : T_x$ with $T_x = \sigma(\Delta_0(x)) \equiv_{\beta\Gamma} \sigma\tau(\Delta_0(x))$. By Lemma 3.2.3, we have then $\sigma : \tau(\Delta_0) \rightsquigarrow_\Gamma \Delta$.

By Lemma 2.6.9, we have $\Gamma;\Delta \vdash \sigma(v) : \sigma\tau(T_0)$.

Finally, since $\sigma\tau(T_0) \equiv_{\beta\Gamma} \sigma(T_0) \equiv_{\beta\Gamma} T$, then we have, by conversion, $\Gamma;\Delta \vdash \sigma(v) : T$. □

It is now possible to show that the rewrite rule for `tail` is well-typed.

Corollary 3.5.11. *The rewrite rule* (`tail` n_1 (`vcons` n_2 e l) $\hookrightarrow$ l) *is well-typed.*

Proof. Using the rules **(Constant)**, **(S-Application)** and **(Free Variable)**, we have, on the one hand,

$\Gamma;\emptyset;\emptyset;\emptyset \Vdash_i$ `tail` $n_1 \Rightarrow (\Delta_1, $`Vector (S` $n_1) \longrightarrow$ `Vector` $n_1, \emptyset)$

for $\Delta_1 = (n_1 :$ `nat`$)$ and, on the other hand,

$\Gamma;\Delta_1;\emptyset,\emptyset \Vdash_i \texttt{vcons}\ n_2\ e\ l \Rightarrow (\Delta_2, \texttt{Vector}\ (\texttt{S}\ n_2), \emptyset)$

for $\Delta_2 = (n_1 : \texttt{nat})(n_2 : \texttt{nat})(e : \texttt{term})(l : \texttt{Vector}\ n_2)$.
By the **(Inversion)** rule, we have

$\Gamma;\Delta_1;\emptyset;\emptyset \Vdash_c \texttt{vcons}\ n_2\ e\ l \Leftarrow \texttt{Vector}\ (\texttt{S}\ n_1) \mid (\Delta_2, \mathscr{C})$

for $\mathscr{C} = \{(\texttt{Vector}\ (\texttt{S}\ n_2), \texttt{Vector}\ (\texttt{S}\ n_1))\}$.
Finally, by the rule **(S-Application)**, we have

$\Gamma;\emptyset;\emptyset;\emptyset \Vdash_i \texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \Rightarrow (\Delta_2, \texttt{Vector}\ n_1, \mathscr{C})$.

As we have already seen, $\tau = \{n_1 \mapsto n_2\} \in \mathit{PreSol}_\Gamma(\mathit{dom}(\Delta_2), \mathscr{C})$ and we have

$\Gamma;\tau(\Delta_2) \vdash l : \tau(\texttt{Vector}\ n_1)$ with $\tau(\Delta_2) = (n_2 : \texttt{nat})(e : \texttt{term})(l : \texttt{Vector}\ n_2)$.

Therefore, by Theorem 3.5.8, the rewrite rule $(\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow l)$ is well-typed in Γ. □

3.6 Weakly Well-Formed Global Contexts

So far, we have proved that the rewrite rules satisfying the assumptions of Theorem 3.5.8 are well-typed but not that they are *permanently* well-typed. In fact, these rewrite rules are not permanently well-typed in general.

Consider the rewrite rule on `tail`:

$\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow l$.

We know that it satisfies the assumptions of Theorem 3.5.8 for Γ. Now, consider the following extension Γ_2:

`NonEmptyVector` : **Type**.
$\texttt{Vector}\ (\texttt{S}\ n) \hookrightarrow \texttt{NonEmptyVector}$.

In the local context $\Delta = (n_1 : \texttt{nat})(n_2 : \texttt{nat})(e : \texttt{term})(l : \texttt{Vector}\ n_2)$, since $\texttt{Vector}\ (\texttt{S}\ n_1) \equiv_{\beta\Gamma_2}$ $\texttt{NonEmptyVector} \equiv_{\beta\Gamma_2} \texttt{Vector}\ (\texttt{S}\ n_2)$, we have:

$\Gamma_2;\Delta \vdash \texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) : \texttt{Vector}\ n_1$.

and the following reduction:

$\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \to_{\Gamma_2} l$.

However, we do not have $\Gamma_2;\Delta \vdash l : \texttt{Vector}\ n_1$ since $\texttt{Vector}\ n_1$ and $\texttt{Vector}\ n_2$ are not convertible. Subject reduction is broken and the rewrite rule is not permanently well-typed.

The problem is that the pre-solution $\tau = \{n_1 \mapsto n_2\}$ for $\mathscr{C} = \{(\texttt{Vector}\ (\texttt{S}\ n_1), \texttt{Vector}\ (\texttt{S}\ n_2))\}$ in Γ in $V = \{n_1, n_2, e, l\}$) is no more a pre-solution for $\mathscr{C}$ in Γ_2. That is to say, $\tau \in \mathit{PreSol}_\Gamma(V, \mathscr{C})$, but $\tau \notin \mathit{PreSol}_{\Gamma_2}(V, \mathscr{C})$. Theorem 3.5.8 do not apply anymore.

Therefore, to prove that a rewrite rule is permanently well-typed we need a stronger notion of pre-solution. We need to characterize the substitutions that remain pre-solutions in any extension of the global context.

Definition 3.6.1 (Permanent Pre-Solution (First Attempt))**.** *Let Γ be a global context, V be a set of variables and $\mathscr{C}$ be a set of constraints.*

A permanent pre-solution *for $\mathscr{C}$ is a substitution τ such that, for any well-typed extension Γ_2 of Γ and any $\sigma \in \mathit{Sol}_{\Gamma_2}(V, \mathscr{C})$, $\sigma \equiv_{\beta\Gamma_2} \sigma\tau$ on V.*

Unfortunately, this new definition is useless. When $\mathscr{C}$ is a set of constraints originating from a typing problem in the $\lambda\Pi$-Calculus Modulo, then the only permanent pre-solution in the sense of Definition 3.6.1 is the identity substitution.

Indeed, the (satisfiable) typing constraints arising when typing a term in the $\lambda\Pi$-Calculus Modulo can be decomposed into constraints of the form $(C\vec{v}, C\vec{u})$ where C is a constant. The problem is that, for permanent pre-solutions, we need to consider every possible extensions of the global context, in particular the extension where the constraint is trivially verified, for example if we have the rewrite rule $(C\vec{v} \hookrightarrow C\vec{u})$. It follows that the identity is the only permanent pre-solution. Therefore, we are back to Section 3.4...

A simple solution to this problem is to restrict the possible extensions of the global context to contexts verifying some properties that we can use to find non-trivial permanent pre-solutions.

Let us look back at our example where $\mathscr{C} = \{(\texttt{Vector}\ (\texttt{S}\ n_1), \texttt{Vector}\ (\texttt{S}\ n_2))\}$. To prove that $\tau = \{n_1 \mapsto n_2\}$ is a pre-solution, we need, first, to remark that there are no rewrite rules whose head symbol is `Vector` or `S` and, second, that we use the confluence of $\rightarrow_{\beta\Gamma}$. We call safe the global contexts verifying these conditions.

3.6.1 Safe Global Context

We consider that we have a partition of the set of constants in two (infinite) sets: $\mathscr{C} = \mathscr{C}_S \uplus \mathscr{C}_D$.

Definition 3.6.2 (Static and Definable Symbols)**.** *Constants in $\mathscr{C}_S$ are called* static symbols. *Constants in $\mathscr{C}_D$ are called* definable symbols.

Static symbols are meant to be not rewritten; we will forbid the rewrite rules whose head constant is a static symbol. On the contrary, there are no restrictions for definable symbols.

For our examples on vectors, the type constant `Vector` and the object constants 0 and `S` are not meant to be rewritten: they should be static symbols. On the other hand, there are rewrite rules defining the constants `head` and `tail`; therefore, they should be declared as definable symbols.

Definition 3.6.3 (Safe Global Context)**.** *A global context Γ is* safe *if:*

- *Γ is well-typed;*
- *$\rightarrow_{\beta\Gamma}$ is confluent;*
- *for all $(u \hookrightarrow v) \in \Gamma$, $u = f\,\vec{w}$ where f is a definable symbol.*

3.6.2 Weakly Well-Formed Rewrite Rules

Now we can give a revised definition of permanent pre-solutions.

Definition 3.6.4 (Permanent Pre-Solution)**.** *Let Γ be a global context, V be a set of variables and $\mathscr{C}$ be a set of constraints.*

A permanent pre-solution *for $\mathscr{C}$ in Γ is a substitution τ such that, for any safe extension Γ_2 of Γ and any $\sigma \in Sol_{\Gamma_2}(V, \mathscr{C})$, we have $\sigma \equiv_{\beta\Gamma_2} \sigma\tau$.*

We write $PreSol^{per}_{\Gamma}(V, \mathscr{C})$ the set of permanent pre-solutions for $\mathscr{C}$ in Γ.

We now use the notion of permanent pre-solution to define weakly well-formed rewrite rules.

Definition 3.6.5 (Weakly Well-Formed Rewrite Rule)**.** *A rewrite rule* $(u \hookrightarrow v)$ *is* weakly-well-formed *in* Γ *if there exist* Δ, T *and* τ *such that*

- $\Gamma; \emptyset; \emptyset, \emptyset \Vdash_i u \Rightarrow (\Delta, T, \mathscr{C})$,
- $\tau \in PreSol_\Gamma^{per}(dom(\Delta), \mathscr{C})$,
- $\Gamma \vdash^{ctx} \tau(\Delta)$
- *and* $\Gamma; \tau(\Delta) \vdash v : \tau(T)$

We write $\Gamma \vdash^w u \hookrightarrow v$ *if* $(u \hookrightarrow v)$ *is weakly well-formed in* Γ.

As expected, weakly well-formed rewrite rules remain well-typed as long as confluence is preserved and no rewrite rule is associated to a static symbol.

Theorem 3.6.6. *Let* Γ *be a global context and* Γ_2 *be a safe extension of* Γ.
If a rewrite rule is weakly-well-formed in Γ*, then it is well-typed in* Γ_2.

The proof follows the lines of the proof of Theorem 3.5.8. We only state the corresponding main lemma.

Lemma 3.6.7 (Main Lemma for Theorem 3.6.6)**.** *Let* Γ *be a global context and* Γ_2 *be a safe extension of* Γ. *Suppose that:*

- $\Gamma_2; \Delta\sigma(\Sigma) \vdash \sigma(t) : T$,
- $\sigma \in Sol_{\Gamma_2}(dom(\Delta_1), \mathscr{C}_1)$,
- $dom(\Sigma) \cap dom(\sigma) = \emptyset$,
- $dom(\Sigma) \cap codom(\sigma) = \emptyset$
- *and* $\sigma : \Delta_1 \rightsquigarrow_{\Gamma_2} \Delta$.

We have

- *if* $\Gamma; \Delta_1; \Sigma; \mathscr{C}_1 \Vdash_i t \Rightarrow (\Delta_2, T_2, \mathscr{C}_2)$, *then*
  - $T \equiv_{\beta\Gamma_2} \sigma(T_2)$,
  - $T =$ **Kind** *or* $\Gamma_2; \Delta\sigma(\Sigma) \vdash \sigma(T_2) : s$,
  - $\sigma \in Sol_{\Gamma_2}(dom(\Delta_2), \mathscr{C}_2)$,
  - *and* $\sigma : \Delta_2 \rightsquigarrow_{\Gamma_2} \Delta$;
- *if* $\Gamma; \Delta_1; \Sigma; \sigma_1 \Vdash_c t \Leftarrow T_1 \mid (\Delta_2, \sigma_2)$ *and* $T \equiv_{\beta\Gamma_2} \sigma(T_1)$ *with* $\Gamma_2; \Delta\sigma(\Sigma) \vdash \sigma(T_1) : s$, *then*
  - $\sigma \in Sol_{\Gamma_2}(dom(\Delta_2), \mathscr{C}_2)$,
  - *and* $\sigma : \Delta_2 \rightsquigarrow_{\Gamma_2} \Delta$.

(Empty Global Context)

$$\frac{}{\emptyset \ \textbf{wwf}}$$

(Object Declaration)

$$\frac{\Gamma \ \textbf{wwf} \qquad \Gamma;\emptyset \vdash \underline{U} : \textbf{Type} \qquad \underline{c} \notin dom(\Gamma)}{\Gamma(\underline{c} : \underline{U}) \ \textbf{wwf}}$$

(Type Declaration)

$$\frac{\Gamma \ \textbf{wwf} \qquad \Gamma;\emptyset \vdash K : \textbf{Kind} \qquad C \notin dom(\Gamma)}{\Gamma(C : K) \ \textbf{wwf}}$$

(Rewrite Rules)

$$\frac{\begin{array}{c}\Gamma \ \textbf{wwf} \qquad \rightarrow_{\beta\Gamma\Xi} \text{ is confluent} \qquad \Xi = (f_1 \vec{u_1} \hookrightarrow v_1) \ldots (f_n \vec{u_n} \hookrightarrow v_n) \\ (\forall i) \Gamma \vdash^{w} u_i \hookrightarrow v_i \\ (\forall i)\, f_i \text{ is a definable symbol}\end{array}}{\Gamma\Xi \ \textbf{wwf}}$$

Figure 3.5: Weakly well-formedness rules for global contexts

3.6.3 Weakly Well-Formed Global Contexts

We now define a variant of the notion of strongly well-formed global context based on the notion of weakly well-formed rewrite rule instead of that of strongly well-formed rewrite rule.

Definition 3.6.8 (Weakly Well-Formed Global Contexts)**.** *A global context* Γ *is* weakly well-formed *if the judgment* Γ **wwf** *is derivable from the inference rules of Figure 3.5.*

Theorem 3.6.9. *Weakly well-formed global contexts are safe.*

Proof. By induction on the derivation of **wwf**, we prove that:

- the declarations are well-typed;
- $\rightarrow_{\beta\Gamma}$ is confluent;
- for all $(u \hookrightarrow v) \in \Gamma$, $u = c\vec{w}$ where c is a definable symbol;
- and the rewrite rules are weakly well-typed in some prefix of Γ.

By Theorem 2.6.11, product compatibility holds. By Theorem 3.6.6, the rewrite rules are well-typed. Therefore Γ is safe. □

Corollary 3.6.10. *Weakly well-formed global contexts are well-typed.*

3.6.4 Examples

We now look at some examples of weakly well-formed global contexts.

Functions on Vectors Suppose that `Vector` and `S` are static symbols. Using Theorem 3.6.6, we can show that the following context is weakly well-formed.
The rewrite rule for `tail`:

$\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow l.$

The rewrite rules for `vmap`, the function that applies a function to any element of a vector, are:

$\texttt{vmap} : \Pi x : \texttt{nat}.(\texttt{term} \longrightarrow \texttt{term}) \longrightarrow \texttt{Vector}\ n \longrightarrow \texttt{Vector}\ n.$
$\texttt{vmap}\ 0\ f\ \texttt{vnil} \hookrightarrow \texttt{vnil}.$
$\texttt{vmap}\ (S\ n_1)\ f\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow \texttt{vcons}\ n_2\ (f\ e)\ (\texttt{vmap}\ n_2\ f\ l).$

The rewrite rules for `vappend`, the function that concatenates two vectors, are:

$\texttt{vappend} : \Pi n_1 : \texttt{nat}.\Pi n_2 : \texttt{nat}.\texttt{Vector}\ n_1 \longrightarrow \texttt{Vector}\ n_2 \longrightarrow \texttt{Vector}\ (\texttt{plus}\ n_1\ n_2).$
$\texttt{vappend}\ 0\ n\ \texttt{vnil}\ l \hookrightarrow l.$
$\texttt{vappend}\ (\texttt{S}\ n_1)\ m\ (\texttt{vcons}\ n_2\ e\ l_1)\ l_2 \hookrightarrow \texttt{vcons}\ (\texttt{plus}\ n_2\ m)\ e\ (\texttt{vappend}\ n_2\ m\ l_1\ l_2).$

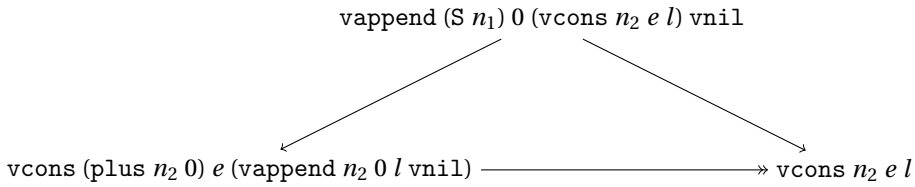
Remark 3.6.11. *We could write the second rewrite rule on* `vappend` *differently:*

$\texttt{vappend}\ (\texttt{S}\ n_1)\ m\ (\texttt{vcons}\ n_2\ e\ l_1)\ l_2 \hookrightarrow \texttt{vcons}\ (\texttt{plus}\ n_1\ m)\ e\ (\texttt{vappend}\ n_1\ m\ l_1\ l_2).$

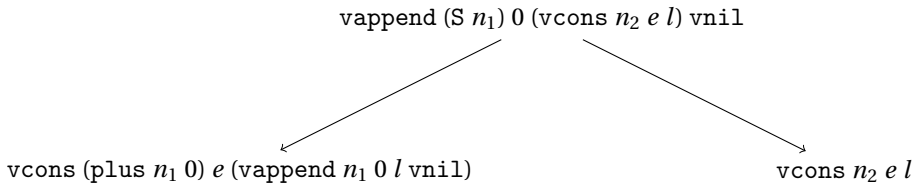
These two rules should be equivalent since, morally, n_1 and n_2 are convertible. However, consider the following rewrite rule:

$\texttt{vappend}\ n\ 0\ l\ \texttt{vnil} \hookrightarrow l.$

In both cases we have a new critical peak. With the first version of the rule on `vappend`*, the critical peak is joinable and the confluence is preserved (Theorem 1.4.7).*



With the second version of the rule, the peak is not joinable; confluence is lost.



Therefore, the way we linearize a rewrite rule matters for proving confluence.

The Simply Typed λ-Calculus Let us define an encoding of the simply typed λ-calculus. Simple types are the terms of type `stype` built from the base type ι and the constructor `arrow`. λ-terms of simple type A are encoded by the terms of type `lterm` A. The symbols `lapp` and `labs` correspond, respectively, to application and

abstraction in the simply typed λ-calculus.

$\texttt{stype} : \textbf{Type}$.
$\iota : \texttt{stype}$.
$\texttt{arrow} : \texttt{stype} \longrightarrow \texttt{stype} \longrightarrow \texttt{stype}$.
$\texttt{lterm} : \texttt{stype} \longrightarrow \textbf{Type}$.
$\texttt{lapp} : \Pi x : \texttt{stype}.\Pi y : \texttt{stype}.\texttt{lterm}\ (\texttt{arrow}\ x\ y) \longrightarrow \texttt{lterm}\ x \longrightarrow \texttt{lterm}\ y$.
$\texttt{labs} : \Pi x : \texttt{stype}.\Pi y : \texttt{stype}.(\texttt{lterm}\ x \longrightarrow \texttt{lterm}\ y) \longrightarrow \texttt{lterm}\ (\texttt{arrow}\ x\ y)$.

β-reduction in the simply typed λ-calculus is simulated using the following rewrite rule:

$\texttt{lapp}\ a_1\ b_1\ (\texttt{labs}\ a_2\ b_2\ t_1)\ t_2 \hookrightarrow t_1\ t_2$.

Assuming that `lterm` and `arrow` are static symbols, by Theorem 3.6.6, this rewrite rule is weakly well-typed.

3.6.5 Optimizing Pattern Matching

So far, we have motivated the use of ill-typed left-hand side of rewrite rules to turn non-left-linear rewrite rules into left-linear rewrite rules to recover the confluence of the rewriting relation. Very similar ideas have been used [BMM03] to optimize dependent pattern matching and the representation of terms in memory in the context of dependent type theory with inductive types and elimination operators. One of the ideas is that the rule:

$\texttt{vmap}\ (\texttt{S}\ n)\ f\ (\texttt{vcons}\ n\ e\ l) \hookrightarrow \texttt{vcons}\ n\ (f\ e)\ (\texttt{vmap}\ n\ f\ l)$.

can be replaced by:

$\texttt{vmap}\ n_1\ f\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow \texttt{vcons}\ n_2\ (f\ e)\ (\texttt{vmap}\ n_2\ f\ l)$.

since, at *runtime*, because of typing constraints, n_1 can only be convertible with $\texttt{S}\ n_2$. Here we have done more than just *linearizing* the rule since we have also deleted the constant `S`, resulting in an optimization of the pattern matching. This last rewrite rule is weakly well-formed; therefore this notion can also be used to justify the type-safety of such optimizations.

These ideas have been implemented in Agda using an algorithm similar to ours [Nor07].

3.7 Characterisation of Well-Typedness of Rewrite Rules

Even if the notion of weakly well-formed rewrite rule is much more general than the notion of strongly well-formed rewrite rule we started with, it does not coincide with the notion of well-typed rewrite rule. Consider the following rewrite rules.

$y \hookrightarrow y$.	**(Identity)**
$(\lambda x : \texttt{nat}.y\ x) \hookrightarrow y$.	**(η-reduction on** `nat`**)**
$(\lambda x : \texttt{nat}.y)\ 0 \hookrightarrow y$.	**(Trivial β-redex)**

The three rewrite rules are well-typed, but none of them is weakly well-formed. The reason is that, in each case, it is not possible to infer the type of y because it is not uniquely defined. For instance, $\lambda x : \texttt{nat}.\texttt{S}\ x$ and $\lambda : x : \texttt{nat}.\texttt{isZero}\ x$ are two instances of the η-reduction rule. In the first case, y is substituted by the term `S` of type $\texttt{nat} \longrightarrow \texttt{nat}$ and, in the second case, by the term `isZero` of type $\texttt{nat} \longrightarrow \texttt{prop}$.

(**Sort**)
$$\frac{}{\Gamma;\Sigma \Vdash^e \mathbf{Type} : (\emptyset, \emptyset, \mathbf{Kind})}$$

(**Constant**)
$$\frac{(c : A) \in \Gamma}{\Gamma;\Sigma \Vdash^e c : (\emptyset, \emptyset, A)}$$

(**Σ-Variable**)
$$\frac{(x : A) \in \Sigma}{\Gamma;\Sigma \Vdash^e x : (\emptyset, \emptyset, A)}$$

(**Free Variable**)
$$\frac{x \notin dom(\Sigma)}{\Gamma;\Sigma \Vdash^e x : (\{x\}, \emptyset, X_x)}$$

(**O-Application**)
$$\frac{\begin{array}{ll} & \{\vec{x}\} = dom(\Sigma) \\ \Gamma;\Sigma \Vdash^e u : (\mathcal{V}_1, \mathscr{C}_1, T) & X \text{ is a fresh type variable} \\ \Gamma;\Sigma \Vdash^e v : (\mathcal{V}_2, \mathscr{C}_2, A) & \mathscr{C}_3 = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.T = \lambda\Sigma.\Pi y : A.X\ \vec{x}\ y)\} \end{array}}{\Gamma;\Sigma \Vdash^e u\ v : (\mathcal{V}_1 \cup \mathcal{V}_2, \mathscr{C}_3, X\ \vec{x}\ v)}$$

(**T-Application**)
$$\frac{\begin{array}{l} \Gamma;\Sigma \Vdash^e U : (\mathcal{V}_1, \mathscr{C}_1, \Pi x : A_1.B) \\ \Gamma;\Sigma \Vdash^e v : (\mathcal{V}_2, \mathscr{C}_2, A_2) \qquad \mathscr{C}_3 = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.A_1 = \lambda\Sigma.A_2)\} \end{array}}{\Gamma;\Sigma \Vdash^e U\ v : (\mathcal{V}_1 \cup \mathcal{V}_2, \mathscr{C}_3, B[x/v])}$$

(**Abstraction**)
$$\frac{\begin{array}{ll} & B \neq \mathbf{Kind} \\ \Gamma;\Sigma \Vdash^e A : (\mathcal{V}_1, \mathscr{C}_1, \mathbf{Type}) & \Gamma;\Sigma(x : A) \Vdash^e u : (\mathcal{V}_2, \mathscr{C}_2, B) \end{array}}{\Gamma;\Sigma \Vdash^e \lambda x : A.u : (\mathcal{V}_1 \cup \mathcal{V}_2, \mathscr{C}_1 \cup \mathscr{C}_2, \Pi x : A.B)}$$

(**T-Product**)
$$\frac{\Gamma;\Sigma \Vdash^e A : (\mathcal{V}_1, \mathscr{C}_1, \mathbf{Type}) \qquad \Gamma;\Sigma(x : A) \Vdash^e B : (\mathcal{V}_2, \mathscr{C}_2, s)}{\Gamma;\Sigma \Vdash^e \Pi x : A.B : (\mathcal{V}_1 \cup \mathcal{V}_2, \mathscr{C}_1 \cup \mathscr{C}_2, s)}$$

Figure 3.6: Typing constraints for terms

In this section, we give an exact characterization of well-typed rewrite rules as a problem of inclusion between sets of solutions to unification constraints.

3.7.1 Typing All Terms

We give the definition of a type system similar to Definition 3.5.7 but that is able to *type* more terms.

Definition 3.7.1. *The relation $\Vdash^e$ is defined by induction from the inference rules of Figure 3.6.*

Remark that in the rule **(Free Variable)** and **(O-Application)**, we make use of type-level variables. Moreover, we assume that every (object-level) variable x is associated to a type-level variable X_x.

As for $\Vdash$, the type system $\Vdash^e$ records the typing constraints of the term. But, to be able to synthesize a type in any case, it may introduce fresh type variables when the type cannot be inferred.

The judgment $\Gamma;\Sigma \Vdash^e A : (\mathcal{V},\mathscr{C},T)$ means that we are exploring the term A in the local context Σ. $\mathcal{V}$ is the set of variables free in A that are not in $dom(\Sigma)$ and $\mathscr{C}$ is the set of typing constraints that should be verified for A to be well-typed of type T.

Remark 3.7.2. *The case of type-level application* **(T-Application)** *is different from that of object-level application* **(O-Application)**. *Indeed, since kind-level product types are only convertible to other product types, we may assume that the type of the term in function position is already a product-type. The only constraint we need to add then is that the expected type and inferred type of the argument match.*

Remark 3.7.3. *For any Γ, Σ and t, there exists at most one tuple $(\mathcal{V},\mathscr{C},T)$ (up to fresh variable renaming) such that $\Gamma;\Sigma \Vdash^e t : (\mathcal{V},\mathscr{C},T)$. We say that the pair $(\mathcal{V},\mathscr{C})$ is a unification problem.*

Notation 3.7.4. *When $\mathscr{C}$ is a set of constraints, we write $FTV(\mathscr{C})$ for the set of free type variables occurring in $\mathscr{C}$.*

3.7.2 Solutions of a Set of Typing Constraints

In the previous sections, a solution of a set of constraints was a substitution σ equating all the constraints. Here we also consider a substitution τ substituting the type variables introduced by the rules **(Free Variable)** and **(O-Application)**. To type these substitutions we also need to consider a local context Δ.

Definition 3.7.5. *Let Γ be a well-typed global context, $\mathcal{V}$ be a set of variables and $\mathscr{C}$ be a set of constraints.*

The triple (Δ,σ,τ) is a solution of $(\mathcal{V},\mathscr{C})$ if

- $\Gamma \vdash^{ctx} \Delta$;
- $\mathcal{V} = dom(\sigma)$;
- *τ is a type-level substitution such that*
  - $dom(\tau) = \{X_x | x \in dom(\sigma)\} \cup FTV(\mathscr{C})$,
  - *for all* $x \in dom(\sigma), \Gamma;\Delta \vdash \sigma(x) : \tau(X_x)$,
  - $\sigma\tau(X_x) = \tau(X_x)$
  - *and, for all* $(T_1,T_2) \in \mathscr{C}$, $\sigma\tau(T_1) \equiv_{\beta\Gamma} \sigma\tau(T_2)$.

We write $Sol(\mathcal{V},\mathscr{C})$ the set of solutions of $(\mathcal{V},\mathscr{C})$.

Remark 3.7.6. *If $(\Delta,\sigma,\tau) \in Sol(\mathcal{V},\mathscr{C})$, then $codom(\sigma) \cup codom(\tau) \subset dom(\Delta)$. This follows from the fact that, for all $x \in dom(\sigma), \Gamma;\Delta \vdash \sigma(x) : \tau(X_x)$,*

Lemma 3.7.7. *Let Γ be a well-typed global context.*

If $(\Delta,\sigma_{|\mathcal{V}_1},\tau_1) \in Sol(\mathcal{V}_1,\mathscr{C}_1)$, $(\Delta,\sigma_{|\mathcal{V}_2},\tau_2) \in Sol(\mathcal{V}_2,\mathscr{C}_2)$ and $FTV(\mathscr{C}_1) \cap FTV(\mathscr{C}_2) \subset \{ X_x \mid x \in dom(\sigma) \}$, then there exists τ_3 such that $(\Delta,\sigma_{|\mathcal{V}_1\cup\mathcal{V}_2},\tau_3) \in Sol(\mathcal{V}_1\cup\mathcal{V}_2,\mathscr{C}_1\cup\mathscr{C}_2)$.

Moreover, for all $X \in dom(\tau_1)$, we have $\tau_3(X) \equiv_{\beta\Gamma} \tau_1(X)$ and, for all $X \in dom(\tau_2)$, we have $\tau_3(X) \equiv_{\beta\Gamma} \tau_2(X)$.

Proof. Since, for all $x \in dom(\sigma), \Gamma; \Delta \vdash \sigma(x) : \tau_1(X_x)$ and $\Gamma; \Delta \vdash \sigma(x) : \tau_2(X_x)$, by uniqueness of types, we have $\tau_1(X) \equiv_{\beta\Gamma} \tau_2(X)$ for all $X \in dom(\tau_1) \cap dom(\tau_2)$. Therefore, we can take

$$\tau_3(X) = \begin{cases} \tau_1(X) \text{ if } X \in dom(\tau_1), \\ \tau_2(X) \text{ if } X \in dom(\tau_2) \setminus dom(\tau_1) \end{cases}$$

□

3.7.3 A Characterisation of Well-Typed Rewrite Rules

The following theorem says that the well-typedness problem for rewrite rules corresponds to an inclusion problem between sets of solutions of unification constraints.

Theorem 3.7.8. *Let Γ be a well-typed global context such that $\rightarrow_{\beta\Gamma}$ is confluent. Suppose that $\Gamma; \emptyset \Vdash^e u : (\mathcal{V}_u, \mathscr{C}_u, T_u)$ and $\Gamma; \emptyset \Vdash^e v : (\mathcal{V}_v, \mathscr{C}_v, T_v)$. The two following propositions are equivalent:*

- *the rewrite rule $(u \hookrightarrow v)$ is well-typed for Γ;*
- *for all $(\Delta, \sigma, \tau_1) \in Sol(\mathcal{V}_u, \mathscr{C}_u)$, there exists τ_2 such that $(\Delta, \sigma, \tau_2) \in Sol(\mathcal{V}_u \cup \mathcal{V}_v, \mathscr{C}_u \cup \mathscr{C}_v \cup \{(T_u = T_v)\})$.*

Before proving the theorem, we need to prove several lemmas.
First, as in the previous sections, we prove the *main lemma.*

Lemma 3.7.9. *Let Γ be a well-typed global context such that $\rightarrow_{\beta\Gamma}$ is confluent.*
Suppose that:

- $\Gamma; \Sigma \Vdash^e t : (\mathcal{V}, \mathscr{C}, T_0)$,
- $\Gamma \vdash^{ctx} \Delta$,
- $dom(\Sigma) \cap dom(\sigma) = \emptyset$,
- $codom(\sigma) \cap dom(\Sigma) = \emptyset$
- *and* $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(t) : T$.

Then, there exists τ such that:

- $(\Delta, \sigma_{|\mathcal{V}}, \tau) \in Sol(\mathcal{V}, \mathscr{C})$
- *and* $\sigma\tau(T_0) \equiv_{\beta\Gamma} T$.

Proof. By induction on $\Gamma; \Sigma \Vdash^e t : (\mathcal{V}, \mathscr{C}, T_0)$.

- **(Sort)** Trivial.
- **(Constant)** Trivial.
- **(Σ-Variable)** Trivial.
- **(Free Variable)** Suppose $t = x \notin dom(\Sigma)$ and $T_0 = X_x$. We have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(x) : T$. Since $codom(\sigma) \cap dom(\Sigma) = \emptyset$ (and $dom(\Sigma) = dom(\sigma(\Sigma))$), we have $FV(\sigma(x)) \cap dom(\sigma(\Sigma)) = \emptyset$; therefore, there exists a reduct T_2 of T such that $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(x) : T_2$ and $FV(T_2) \cap dom(\sigma(\Sigma)) = \emptyset$. It follows that we also have $\Gamma; \Delta \vdash \sigma(x) : T_2$.

  We can take $\tau = \{X_x \mapsto T_2\}$.

- **(O-Application)** Suppose that $t = u\ v$, $T_0 = X\ \vec{x}\ v$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.P = \lambda\Sigma.\Pi y : A_0.X\ \vec{x}\ y)\}$, $\Gamma; \Sigma \Vdash^e u : (\mathcal{V}_1, \mathscr{C}_1, P)$ and $\Gamma; \Sigma \Vdash^e v : (\mathcal{V}_2, \mathscr{C}_2, A_0)$.

  By inversion, we have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(u) : \Pi y : A.B$ and $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(v) : A$ and $T \equiv_{\beta\Gamma} B[y/\sigma(v)]$.

  By induction hypothesis, $(\Delta, \sigma_{|\mathcal{V}_1}, \tau_1) \in Sol(\mathcal{V}_1, \mathscr{C}_1)$, $(\Delta, \sigma_{|\mathcal{V}_2}, \tau_2) \in Sol(\mathcal{V}_2, \mathscr{C}_2)$, $\Pi y : A.B \equiv_{\beta\Gamma} \sigma\tau_1(P)$ and $A \equiv_{\beta\Gamma} \sigma\tau_2(A_0)$.

  By Lemma 3.7.7, there exists τ_3 such that $(\Delta, \sigma_{|\mathcal{V}}, \tau_3) \in Sol(\mathcal{V}, \mathscr{C}_1 \cup \mathscr{C}_2)$.

  We take $\tau = \tau_3 \uplus \{X \mapsto \lambda\vec{x} : \Sigma(\vec{x}).\lambda y : A.B\}$.

  We have $T \equiv_{\beta\Gamma} B[y/\sigma(v)] \equiv_{\beta\Gamma} (\lambda\vec{x} : \Sigma(\vec{x}).\lambda y : A.B)\ \vec{x}\ \sigma(v) = \sigma\tau(X\ \vec{x}\ v)$.

- **(T-Application)** Suppose that $t = U\ v$, $T_0 = B_1[x/v]$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.A_1 = \lambda\Sigma.A_2)\}$, $\Gamma; \Sigma \Vdash^e U : (\mathcal{V}_1, \mathscr{C}_1, \Pi x : A_1.B_1)$ and $\Gamma; \Sigma \Vdash^e v : (\mathcal{V}_2, \mathscr{C}_2, A_2)$.

  By inversion, we have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(U) : \Pi x : A.B$ and $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(v) : A$ and $T \equiv_{\beta\Gamma} B[x/\sigma(v)]$.

  By induction hypothesis, $(\Delta, \sigma_{|\mathcal{V}_1}, \tau_1) \in Sol(\mathcal{V}_1, \mathscr{C}_1)$, $(\Delta, \sigma_{|\mathcal{V}_2}, \tau_2) \in Sol(\mathcal{V}_2, \mathscr{C}_2)$, $\Pi x : A.B \equiv_{\beta\Gamma} \sigma\tau_1(\Pi x : A_1.B_1)$ and $A \equiv_{\beta\Gamma} \sigma\tau_2(A_2)$.

  By confluence, $A \equiv_{\beta\Gamma} \sigma\tau_1(A_1)$ and $B \equiv_{\beta\Gamma} \sigma\tau_1(B_1)$.

  By Lemma 3.7.7, there exists τ such that $(\Delta, \sigma_{|\mathcal{V}}, \tau) \in Sol(\mathcal{V}, \mathscr{C}_1 \cup \mathscr{C}_2)$; for all $X \in dom(\tau_1)$, we have $\tau(X) \equiv_{\beta\Gamma} \tau_1(X)$ and, for all $X \in dom(\tau_2)$, we have $\tau(X) \equiv_{\beta\Gamma} \tau_2(X)$.

  We have: $\sigma\tau(A_2) \equiv_{\beta\Gamma} \sigma\tau_2(A_2) \equiv_{\beta\Gamma} A \equiv_{\beta\Gamma} \sigma\tau_1(A_1) \equiv_{\beta\Gamma} \sigma\tau(A_1)$. Therefore $(\Delta, \sigma_{|\mathcal{V}}, \tau) \in Sol(\mathcal{V}, \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.A_1 = \lambda\Sigma.A_2)\})$.

  Moreover, $T \equiv_{\beta\Gamma} B[x/\sigma(v)] \equiv_{\beta\Gamma} \sigma\tau(B_1[x/v])$.

- **(Abstraction)** Suppose that $t = \lambda x : A_0.u$, $T_0 = \Pi x : A_0.B_0$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$, $\Gamma; \Sigma \Vdash^e A_0 : (\mathcal{V}_1, \mathscr{C}_1, \mathbf{Type})$ and $\Gamma; \Sigma(x : A_0) \Vdash^e u : (\mathcal{V}_2, \mathscr{C}_2, B_0)$.

  By inversion, we have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(A_0) : \mathbf{Type}$, $\Gamma; \Delta\sigma(\Sigma(x : A_0)) \vdash u : B$ and $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).B$.

  By induction hypothesis, $(\Delta, \sigma_{|\mathcal{V}_1}, \tau_1) \in Sol(\mathcal{V}_1, \mathscr{C}_1)$, $(\Delta, \sigma_{|\mathcal{V}_2}, \tau_2) \in Sol(\mathcal{V}_2, \mathscr{C}_2)$ and $B \equiv_{\beta\Gamma} \sigma\tau_2(B_0)$.

  By Lemma 3.7.7, there exists τ such that $(\Delta, \sigma_{|\mathcal{V}}, \tau) \in Sol(\mathcal{V}, \mathscr{C})$; for all $X \in dom(\tau_1)$, we have $\tau(X) \equiv_{\beta\Gamma} \tau_1(X)$ and, for all $X \in dom(\tau_2)$, we have $\tau(X) \equiv_{\beta\Gamma} \tau_2(X)$.

  Moreover, $T \equiv_{\beta\Gamma} \Pi x : \sigma(A_0).B \equiv_{\beta\Gamma} \sigma\tau(\Pi x : A_0.B_0)$ with $\sigma\tau(A_0) = \sigma(A_0)$.

- **(Product)** Suppose that $t = \Pi x : A.B$, $T_0 = s$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$, $\Gamma; \Sigma \Vdash^e A : (\mathcal{V}_1, \mathscr{C}_1, \mathbf{Type})$ and $\Gamma; \Sigma(x : A) \Vdash^e B : (\mathcal{V}_2, \mathscr{C}_2, s)$.

  By inversion, we have $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(A) : \mathbf{Type}$ and $\Gamma; \Delta\sigma(\Sigma(x : A)) \vdash B : s$.

  By induction hypothesis, $(\Delta, \sigma_{|\mathcal{V}_1}, \tau_1) \in Sol(\mathcal{V}_1, \mathscr{C}_1)$ and $(\Delta, \sigma_{|\mathcal{V}_2}, \tau_2) \in Sol(\mathcal{V}_2, \mathscr{C}_2)$.

  By Lemma 3.7.7, there exists τ_3 such that $(\Delta, \sigma_{|\mathcal{V}}, \tau_3) \in Sol(\mathcal{V}, \mathscr{C})$.

□

We now prove the converse.

Lemma 3.7.10. *Let Γ be a well-typed global context such that $\to_{\beta\Gamma}$ is confluent. Suppose that:*

- $\Gamma;\Sigma \Vdash^e t : (\mathscr{V}, \mathscr{C}, T_0)$,
- $dom(\sigma) \cap dom(\Sigma) = \emptyset$,
- $codom(\sigma) \cap dom(\Sigma) = \emptyset$,
- $(\Delta, \sigma, \tau) \in Sol(\mathscr{V}, \mathscr{C})$
- *and no type variable occurs in t or Σ.*

Then, we have $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(t) : T$ for $T \equiv_{\beta\Gamma} \sigma\tau(T_0)$.

Proof. By induction on $\Gamma;\Sigma \Vdash^e t : (\mathscr{V}, \mathscr{C}, T)$.

- **(Sort)** Suppose that $t = \mathbf{Type}$ and $T_0 = \mathbf{Kind}$. We have $\Gamma;\Delta\sigma(\Sigma) \vdash \mathbf{Type} : \mathbf{Kind}$, $\sigma(\mathbf{Type}) = \mathbf{Type}$ and $\sigma\tau(\mathbf{Kind}) = \mathbf{Kind}$.
- **(Constant)** Suppose that $t = c$ and $T_0 = \Gamma(c)$. We have $\sigma(c) = c$, $\sigma\tau(\Gamma(c)) = \Gamma(c)$ (since $\Gamma(c)$ is closed) and $\Gamma;\Delta\sigma(\Sigma) \vdash c : \Gamma(c)$.
- **(Σ-Variable)** Suppose that $t = x \in dom(\Sigma)$ and $T_0 = \Sigma(x)$. Since $dom(\Sigma) \cap dom(\sigma) = \emptyset$, we have $\sigma(t) = x$. Moreover, since there are no type variables in T_0, we have $\tau(T_0) = T_0$ therefore, $\Gamma;\Delta\sigma(\Sigma) \vdash x : \sigma\tau(\Sigma(x))$.
- **(Free Variable)** Suppose that $t = x \notin dom(\Sigma)$ and $T_0 = X_x$. By hypothesis, $\Gamma;\Delta \vdash \sigma(x) : \tau(X_x)$. Since $\sigma\tau(X_x) = \tau(X_x)$, by local weakening, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(x) : \sigma\tau(X_x)$.
- **(O-Application)** Suppose that $t = u\ v$, $T_0 = X\ \vec{x}\ v$, $\Gamma;\Sigma \Vdash^e u : (\mathscr{V}_1, \mathscr{C}_1, P)$, $\Gamma;\Sigma \Vdash^e v : (\mathscr{V}_2, \mathscr{C}_2, A)$, $\mathscr{V} = \mathscr{V}_1 \cup \mathscr{V}_2$ and $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.P = \lambda\Sigma.\Pi y : A.X\ \vec{x}\ y)\}$.

  We have $(\sigma, \Delta, \tau) \in Sol(\mathscr{V}_1, \mathscr{C}_1)$ and $(\sigma, \Delta, \tau) \in Sol(\mathscr{V}_2, \mathscr{C}_2)$; therefore, by induction hypothesis, we get $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(p) : T_1$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(q) : T_2$ with $T_1 \equiv_{\beta\Gamma} \sigma\tau(P)$ and $T_2 \equiv_{\beta\Gamma} \sigma\tau(A)$.

  Moreover, we have $\sigma\tau(P) \equiv_{\beta\Gamma} \sigma\tau(\Pi y : A.X\ \vec{x}\ y)$. It follows that $T_1 \equiv_{\beta\Gamma} \sigma\tau(\Pi y : A.X\ \vec{x}\ y)$

  By confluence, there exist A_0 and B_0 such that $T_2 \to_{\beta\Gamma} A_0$, $T_1 \to_{\beta\Gamma} \Pi y : A_0.B_0$ and $\sigma\tau(X\ \vec{x}\ y) \to^*_{\beta\Gamma} B_0$.

  By conversion, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(p) : \Pi y : A_0.B_0$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(q) : A_0$.

  It follows that $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(p\ q) : B_0[y/\sigma(q)]$ with $B_0[y/\sigma(q)] \equiv_{\beta\Gamma} \sigma\tau(X\ \vec{x}\ q)$.
- **(T-Application)** Suppose that $t = U\ v$, $T_0 = B_1[x/v]$, $\mathscr{V} = \mathscr{V}_1 \cup \mathscr{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{(\lambda\Sigma.A_1 = \lambda\Sigma.A_2)\}$, $\Gamma;\Sigma \Vdash^e U : (\mathscr{V}_1, \mathscr{C}_1, \Pi x : A_1.B_1)$ and $\Gamma;\Sigma \Vdash^e v : (\mathscr{V}_2, \mathscr{C}_2, A_2)$.

  We have $(\sigma, \Delta, \tau) \in Sol(\mathscr{V}_1, \mathscr{C}_1)$ and $(\sigma, \Delta, \tau) \in Sol(\mathscr{V}_2, \mathscr{C}_2)$, therefore, by induction hypothesis, $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(U) : T_1$ and $\Gamma;\Delta\sigma(\Sigma) \vdash \sigma(v) : T_2$ with $T_1 \equiv_{\beta\Gamma} \sigma\tau(\Pi x : A_1.B_1)$ and $T_2 \equiv_{\beta\Gamma} \sigma\tau(A_2)$.

  Moreover, $\sigma\tau(A_1) \equiv_{\beta\Gamma} \sigma\tau(A_2)$.

  By confluence, there exist A_0 and B_0 such that $T_2 \to_{\beta\Gamma} A_0$, $T_1 \to_{\beta\Gamma} \Pi x : A_0.B_0$ and $\sigma\tau(B_1) \to^*_{\beta\Gamma} B_0$.

By conversion, $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(U) : \Pi x : A_0.B_0$ and $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(v) : A_0$.

Therefore, we have and $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(U\ v) : B_0[x/\sigma(v)]$ with $B_0[x/\sigma(v)] \equiv_{\beta\Gamma} \sigma\tau(B_1[x/v])$.

- **(Abstraction)** Suppose that $t = \lambda x : A_0.u$, $T_0 = \Pi x : A_0.B_0$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$, $\Gamma; \Sigma \Vdash^e A_0 : (\mathcal{V}_1, \mathscr{C}_1, \mathbf{Type})$ and $\Gamma; \Sigma(x : A_0) \Vdash^e u : (\mathcal{V}_2, \mathscr{C}_2, B_0)$.

  We have $(\sigma, \Delta, \tau) \in Sol(\mathcal{V}_1, \mathscr{C}_1)$ and $(\sigma, \Delta, \tau) \in Sol(\mathcal{V}_2, \mathscr{C}_2)$, therefore, by induction hypothesis, $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(A_0) : \mathbf{Type}$ and $\Gamma; \Delta\sigma(\Sigma(x : A_0)) \vdash \sigma(u) : T_2$ with $T_2 \equiv_{\beta\Gamma} \sigma\tau(B_0)$.

  It follows that $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(\lambda x : A_0.u) : \Pi x : \sigma(A_0).T_2$ and $\Pi x : \sigma(A_0).T_2 \equiv_{\beta\Gamma} \sigma\tau(\Pi x : A_0.B_0)$.

- **(Product)** Suppose that $t = \Pi x : A.B$, $T_0 = s$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$, $\Gamma; \Sigma \Vdash^e A : (\mathcal{V}_1, \mathscr{C}_1, \mathbf{Type})$ and $\Gamma; \Sigma(x : A) \Vdash^e B : (\mathcal{V}_2, \mathscr{C}_2, s)$.

  We have $(\sigma, \Delta, \tau) \in Sol(\mathcal{V}_1, \mathscr{C}_1)$ and $(\sigma, \Delta, \tau) \in Sol(\mathcal{V}_2, \mathscr{C}_2)$; therefore, by induction hypothesis, $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(A) : \mathbf{Type}$ and $\Gamma; \Delta\sigma(\Sigma(x : A)) \vdash \sigma(B) : s$.

  It follows that $\Gamma; \Delta\sigma(\Sigma) \vdash \sigma(\Pi x : A.B) : s$.

□

Finally, we can now prove Theorem 3.7.8.

Proof of Theorem 3.7.8. Let Γ be a well-typed global context such that $\to_{\beta\Gamma}$ is confluent. Suppose that $\Gamma; \emptyset \Vdash^e u : (\mathcal{V}_u, \mathscr{C}_u, T_u)$ and $\Gamma; \emptyset \Vdash^e v : (\mathcal{V}_v, \mathscr{C}_v, T_v)$.

- Assume that $(u \hookrightarrow v)$ is well-typed for Γ. Let $(\Delta, \sigma, \tau_u) \in Sol(\mathcal{V}_u, \mathscr{C}_u)$.

  By Lemma 3.7.10, we have $\Gamma; \Delta \vdash \sigma(u) : \sigma\tau_u(T_u)$ and, by well-typedness of the rewrite rule, $\Gamma; \Delta \vdash \sigma(v) : \sigma\tau_u(T_u)$.

  By Lemma 3.7.9, there exists τ_v such that $(\Delta, \sigma, \tau_v) \in Sol(\mathcal{V}_v, \mathscr{C}_v)$ and $\sigma\tau_v(T_v) \equiv_{\beta\Gamma} \sigma\tau_u(T_u)$.

  By Lemma 3.7.7, there exists τ such that $(\Delta, \sigma, \tau) \in Sol(\mathcal{V}, \mathscr{C}_u \cup \mathscr{C}_v)$, for all $X \in dom(\tau_u)$, we have $\tau(X) \equiv_{\beta\Gamma} \tau_u(X)$ and, for all $X \in dom(\tau_v)$, we have $\tau(X) \equiv_{\beta\Gamma} \tau_v(X)$.

  It follows that $\sigma\tau(T_v) \equiv_{\beta\Gamma} \sigma\tau(T_u)$ and $(\Delta, \sigma, \tau_{uv}) \in Sol(\mathcal{V}, \mathscr{C}_u \cup \mathscr{C}_v \cup \{(T_u = T_v)\})$.

- Now assume that, if $(\Delta, \sigma, \tau_u) \in Sol(\mathcal{V}_u, \mathscr{C}_u)$, then there exists τ_{uv} such that $(\Delta, \sigma, \tau_{uv}) \in Sol(\mathcal{V}_u \cup \mathcal{V}_v, \mathscr{C}_u \cup \mathscr{C}_v \cup \{(T_u = T_v)\})$.

  Let $(u \hookrightarrow v)$ be a rewrite rule and suppose that $\Gamma; \Delta \vdash \sigma(u) : T$.

  By Lemma 3.7.9, for some τ_u, we have $(\Delta, \sigma, \tau_u) \in Sol(\mathcal{V}_u, \mathscr{C}_u)$ and $T \equiv_{\beta\Gamma} \sigma\tau_u(T_u)$.

  Thus, by hypothesis, there exists τ_{uv} such that $(\Delta, \sigma, \tau_{uv}) \in Sol(\mathcal{V}_u \cup \mathcal{V}_v, \mathscr{C}_u \cup \mathscr{C}_v \cup \{(T_u = T_v)\})$.

  Therefore, $(\Delta, \sigma, \tau_{uv}) \in Sol(\mathcal{V}_v, \mathscr{C}_v)$ and, by Lemma 3.7.10, $\Gamma; \Delta \vdash \sigma(v) : V$ for $V \equiv_{\beta\Gamma} \sigma\tau_{uv}(T_v) \equiv_{\beta\Gamma} \sigma\tau_{uv}(T_u) \equiv_{\beta\Gamma} T$.

  It follows, by the conversion rule, that $\Gamma; \Delta \vdash \sigma(v) : T$.

□

3.7.4 Applications

Using Theorem 3.7.8, we are now able to prove that the rewrite rules from the beginning of the section are well-typed.

Corollary 3.7.11. *The rewrite rules* **(Identity)**, **(η-reduction)** *and* **(Trivial β-redex)** *are well-typed.*

Proof.

- **(Identity)** We have:
  $\Gamma; \emptyset \Vdash^e y : (\{y\}, \emptyset, X_y)$ and $Sol(\{y\}, \emptyset) \subset Sol(\{y\}, \{(X_y = X_y)\})$.
  Therefore, by Theorem 3.7.8, the rewrite rule $y \hookrightarrow y$ is well-typed.

- **(η-reduction)** We have:
  $\Gamma; \emptyset \Vdash^e \lambda x : \mathtt{nat}.y\ x : (\{y\}, \mathscr{C}, \Pi x : \mathtt{nat}.X\ x)$
  with $\mathscr{C} = \{(\mathbf{Type} = \mathbf{Type}), (X_y = \Pi x : \mathtt{nat}.X\ x)\}$
  and $\Gamma; \emptyset \Vdash^e y : (\{y\}, \emptyset, X_y)$.

  Since $\mathscr{C} \cup \emptyset \cup \{(X_y = \Pi x : \mathtt{nat}.X\ x)\}) = \mathscr{C}$, by Theorem 3.7.8, the rewrite rule $\lambda x : \mathtt{nat}.y\ x \hookrightarrow y$ is well-typed.

- **(Trivial β-redex)** We have:
  $\Gamma; \emptyset \Vdash^e (\lambda x : \mathtt{nat}.y)\ 0 : (\{y\}, \mathscr{C}, X\ 0)$
  with $\mathscr{C} = \{(\mathbf{Type} = \mathbf{Type}), (\Pi x : \mathtt{nat}.X_y\ x = \Pi x : \mathtt{nat}.X\ x)\}$
  and $\Gamma; \emptyset \Vdash^e y : (\{y\}, \emptyset, X_y)$.

  By product compatibility, $Sol(\{y\}, \{(\mathbf{Type} = \mathbf{Type}), (\Pi x : \mathtt{nat}.X_y = \Pi x : \mathtt{nat}.X\ x)\}) \subset Sol(\{y\}, \{(\mathbf{Type} = \mathbf{Type}), (\Pi x : \mathtt{nat}.X_y\ x = \Pi x : \mathtt{nat}.B\ x), (X\ 0 = X_y)\})$.

  Therefore, by Theorem 3.7.8, the rewrite rule $(\lambda x : \mathtt{nat}.y)\ 0 \hookrightarrow y$ is well-typed.

□

3.7.5 Undecidability

We now show the undecidability of well-typedness of rewrite rules.

Lemma 3.7.12 (Undecidability of Unification)**.** *Let Γ be a global context without rewrite rule and containing the declaration $(\iota : \mathbf{Type})$. Unification modulo $\equiv_{\beta\Gamma}$ $(= \equiv_\beta)$ is undecidable.*

More precisely, there exists a set **Pol** *of terms of type $\mathtt{CN}^n \longrightarrow \mathtt{CN}$ (with $\mathtt{CN} = \iota \longrightarrow (\iota \longrightarrow \iota) \longrightarrow \iota$), for some n, such that it is not decidable, given two terms in* **Pol** *p and q of type $\mathtt{CN}^{n_1} \longrightarrow \mathtt{CN}$ and $\mathtt{CN}^{n_2} \longrightarrow \mathtt{CN}$ respectively, to tell if there exist two vectors $\vec{u}_1$ of size n_1 and $\vec{u}_2$ of size n_2 such that $p\ \vec{u}_1 \equiv_{\beta\Gamma} q\ \vec{u}_2$.*

Proof. We can easily reduce Hilbert tenth problem as it is done for proving the undecidability of higher-order unification in [Dow01]. The set **Pol** corresponds to a set of polynomials represented by Church's numerals. □

Theorem 3.7.13 (Undecidablity of Well-Typedness for Rewrite Rules)**.** *Well-typedness of a rewrite rule is undecidable.*

Proof. We reduce the problem of unification modulo $\equiv_{\beta\Gamma}$ of Lemma 3.7.12. Let Γ be the global context containing the following declarations:

ι : **Type**.

E : $\mathtt{CN} \longrightarrow$ **Type**.
$\mathtt{f}$: $\Pi \vec{x} : \mathtt{CN}.E\ (p\ \vec{x}) \longrightarrow \iota$.
$\mathtt{g}$: $\Pi \vec{y} : \mathtt{CN}.E\ (q\ \vec{y})$.

where p and q are two arbitrary terms in **Pol**. Consider the rewrite rule:

$\mathtt{f}\ \vec{x}\ (\mathtt{g}\ \vec{y}) \hookrightarrow \lambda z : \iota.z$.

According to Theorem 3.7.8, it is well-typed if and only if for all $(\sigma, \Delta, \tau_1) \in Sol_\Gamma(\{\vec{x}, \vec{y}\}, \{(E\ (p\ \vec{x}) = E\ (q\ \vec{y}))\})$, there exists τ_2 such that $(\sigma, \Delta, \tau_2) \in Sol_\Gamma(\{\vec{x}, \vec{y}\}, \{(E\ (p\ \vec{x}) = E\ (q\ \vec{y})), (\iota = \iota \longrightarrow \iota)\})$.

Since the equation $\iota \equiv_{\beta\Gamma} \iota \longrightarrow \iota$ is not satisfiable, the rewrite rule is well-typed if and only if $Sol_\Gamma(\{\vec{x}, \vec{y}\}, \{(E\ (p\ \vec{x}) = E\ (q\ \vec{y}))\}) = \emptyset$.

By Lemma 3.7.12, we cannot decide such equality. □

3.8 Conclusion

We have studied the property of well-typedness of rewrite rules. Starting from the notion of strongly well-formed rewrite rule, we have generalized gradually the criterion of well-typedness to allow non-algebraic and ill-typed left-hand sides. In particular, the new criterion justifies the linearization of left-hand sides of rewrite rules when non-linearity arises from typing constraints. Keeping left-linear rewrite rules is important to preserve confluence of the rewrite system. Indeed, the combination of non left-linear rewrite rules with β-reduction is often non-confluent as the examples of Section 1.4.3 show. Confluence can then be used to prove, among others, product compatibility. Finally, we have given an exact characterisation of the well-typedness property for rewrite rules as a problem of inclusion between solutions of two unification problems and we have used this characterisation to show the undecidability of well-typedness of rewrite rules.

Chapter 4

Rewriting Modulo β

Résumé Ce chapitre définit une notion de réécriture modulo β pour le $\lambda\Pi$-Calcul Modulo. En partant des observations que (1) la confluence du système de réécriture est une propriété vivement souhaitée car elle a pour conséquence la propriété de la compatibilité du produit ainsi que, avec la terminaison, la décidabilité de la congruence et que (2) la confluence est facilement perdue lorsque les règles de réécriture filtrent sous les abstractions, on propose une nouvelle notion de réécriture qui réconcilie confluence et filtrage sous les abstractions. Cette nouvelle notion est définie à travers un encodage des termes vers un *système de réécriture d'ordre supérieur*. Ceci permet d'importer dans le $\lambda\Pi$-Calcul Modulo les résultats de confluence conçus pour les systèmes d'ordre supérieur. On détaille aussi comment la réécriture modulo β peut être efficacement implémentée par la compilation des règles de réécriture en arbres de décision.

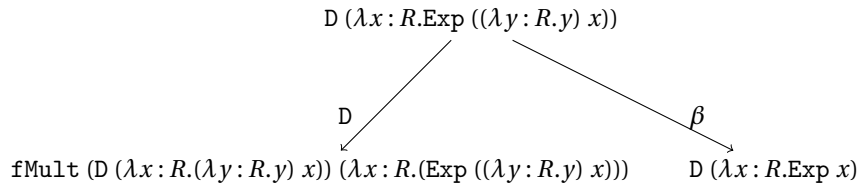
4.1 Introduction

In this chapter, we are interested in the confluence of rewriting systems. In particular, we study a problem arising from the combination of rewrite rules with β-reduction. Remember that confluence is a highly desirable property of the $\lambda\Pi$-Calculus Modulo for several reasons. First, confluence is the most direct way to prove the product compatibility property (Theorem 2.6.11). Second, as soon as the rewrite relation is also strongly normalizing, confluence entails the decidability of the congruence: two terms are convertible if and only if they have the same normal form. Third, confluence has also been used in the previous chapter for proving that weakly-well-formed rewrite rules are permanently well-typed. More generally, any property based on unification will require confluence. Lastly, confluence is used to prove strong normalization when there are type-level rewrite rules [Bla05b].

One case where confluence is easily lost is if one allows rewrite rules with λ-abstractions on their left-hand side. For instance, consider the following rewrite rule (which reflects the mathematical equality $(e^f)' = f' * e^f$):

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ (f\ x)) \hookrightarrow \mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.f\ x))\ (\lambda x : R.\mathtt{Exp}\ (f\ x)).$$

This rule introduces a non-joinable critical peak when combined with β-reduction:



A way to recover confluence is to consider a generalized rewriting relation where matching is done modulo β-reduction. In this setting, $\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x)$ is reducible because it is β-equivalent to the redex $\mathtt{D}\ (\lambda x : R.\mathtt{Exp}((\lambda y : R.y)\ x))$ and, as we will see, this allows closing the critical peak.

In this chapter, we formalize the notion of *rewriting modulo* β in the context of the $\lambda\Pi$-Calculus Modulo. We achieve this by encoding the $\lambda\Pi$-Calculus Modulo into Nipkow's Higher-Order Rewrite Systems [Nip91]. This encoding allows us, first, to properly define matching modulo β using the notion of higher-order rewriting and, secondly, to make the confluence results for higher-order rewriting available for the $\lambda\Pi$-Calculus Modulo.

Then, we prove a version of the subject reduction property for rewriting modulo β and that the confluence of the new rewriting relation implies the product compatibility property, generalizing the results of the previous chapters.

4.2 A Naive Definition of Rewriting Modulo β

As already mentioned, our goal is to give a notion of rewriting modulo β in the setting of the $\lambda\Pi$-Calculus Modulo. We first exhibit the issues arising from a naive definition of this notion.

In an untyped setting, we could define rewriting modulo β in this manner: t_1 rewrites to t_2 if, for some rewrite rule $(u \hookrightarrow v)$ and substitution σ, $\sigma(u) \equiv_\beta t_1$ and $\sigma(v) \equiv_\beta t_2$. This definition is not satisfactory for several reasons.

It breaks subject reduction. For the rewrite rule of Section 4.1, taking $\sigma = \{f \mapsto \lambda y : \Omega.y\}$ where Ω is some ill-typed term, we have

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x) \longrightarrow \mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.(\lambda y : \Omega.y)\ x))\ (\lambda x : R.\mathtt{Exp}\ ((\lambda y : \Omega.y)\ x))$$

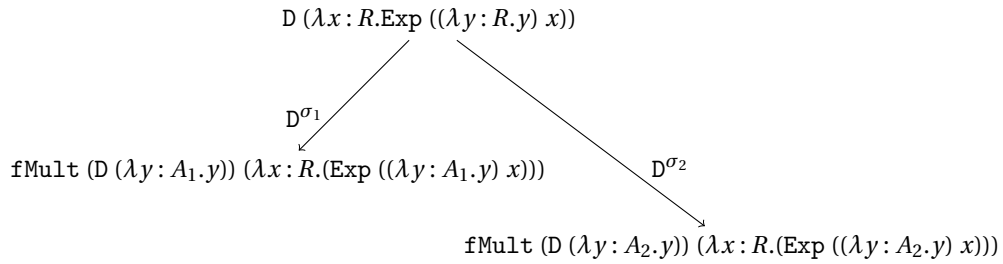
and, even if $\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x)$ is well-typed, its reduct is ill-typed since it contains an ill-typed subterm.

It may introduce free variables. In the example above, Ω has no reason to be closed.

It does not provide confluence. If we consider the following variant of the rewrite rule

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ (f\ x)) \hookrightarrow \mathtt{fMult}\ (\mathtt{D}\ f)\ (\lambda x : R.\mathtt{Exp}\ (f\ x))$$

and take $\sigma_1 = \{f \mapsto \lambda y : A_1.y\}$ and $\sigma_2 = \{f \mapsto \lambda y : A_2.y\}$ where A_1 and A_2 are two non-convertible types, then we have:



and the peak is not joinable.

Therefore, we need to find a definition that takes care of these issues. We will achieve this using an embedding of λΠ-Calculus Modulo into Higher-Order Rewrite Systems.

4.3 Higher-Order Rewrite Systems

In 1991, Nipkow [Nip91] introduced Higher-Order Rewrite Systems (HRS) in order to lift termination and confluence results from first-order rewriting to rewriting over λ-terms. More generally, the goal was to study rewriting over terms with bound variables such as programs, theorems and proofs.

Unlike the λΠ-Calculus Modulo, in HRSs β-reduction and rewriting do not operate at the same level. Rewriting is defined as a relation between the $\beta\eta$-equivalence classes of simply typed λ-terms: the λ-calculus is used as a meta-language.

Higher-Order Rewrite Systems are based upon the (pre)terms of the simply-typed λ-calculus built from a signature. A signature is a set of base types $\mathscr{B}$ and a set of typed constants. A simple type is either a base type $b \in \mathscr{B}$ or an arrow $A \longrightarrow B$ where A and B are simple types.

Definition 4.3.1 (Preterm)**.** *A* preterm *of type A is*

- *either a* variable *x of type A (we assume given for each simple type A an infinite number of variables of this type),*
- *or a* constant *f of type A,*
- *or an* application *$t(u)$ where t is a preterm of type $B \longrightarrow A$ and u is a preterm of type B,*
- *or, if $A = B \longrightarrow C$, an* abstraction *$\underline{\lambda}x.t$ where x is a variable of type B and t is a preterm of type C.*

In order to distinguish the abstraction of HRSs from the abstraction of λΠ-Calculus Modulo, we use the underlined symbol $\underline{\lambda}$ instead of λ. Similarly, we write the application $t(u)$ for HRSs (instead of tu). We use the abbreviation $t(u_1,\dots,u_n)$ for $t(u_1)\dots(u_n)$. If A is a simple type, we write A^1 for A and A^{n+1} for $A \longrightarrow A^n$.

Notice also that HRSs abstractions do not have type annotations because variables are typed.

β-reduction for preterms is defined as usual.

Definition 4.3.2 (Restricted η-expansion)**.** *The relation of restricted η-expansion (written $\rightarrow_{\overline{\eta}}$) is defined as follows:*

$$C[s] \rightarrow_{\overline{\eta}} C[\underline{\lambda}x.s\ x]$$

if the following conditions are satisfied:

- *s has type $A \longrightarrow B$, for some types A and B;*
- *x is a fresh variable of type A;*
- *s is not of the form $\underline{\lambda} z.s_0$;*
- *in the preterm $C[s]$, s does not occur as the left part of an application (it means that restricted η-expansions do not create β-redexes).*

Remark 4.3.3. *A preterm t is equivalent to a unique preterm in $\beta\overline{\eta}$-normal form. We write $\updownarrow_\beta^\eta t$ for the $\beta\overline{\eta}$-normal form of t.*

Definition 4.3.4 (HRS-Term)**.** *A* term *is a preterm in $\beta\overline{\eta}$-normal form.*

Definition 4.3.5 (Pattern)**.** *A term t is a* pattern *if every free occurrence of a variable F is in a subterm of t of the form $F\vec{u}$ such that $\vec{u}$ is η-equivalent to a list of distinct bound variables.*

The crucial result about patterns (due to Miller [Mil91]) is the decidability of their higher-order unification (unification modulo $\beta\eta$). Moreover, if two patterns are unifiable, then a most general unifier exists and is computable.

The notion of rewrite rule for HRSs is the following:

Definition 4.3.6 (Rewrite Rules)**.** *A* rewrite rule *is a pair of terms $(l \hookrightarrow r)$ such that l is a pattern not η-equivalent to a variable, $FV(r) \subset FV(l)$ and l and r have the same base type.*

The restriction to patterns for the left-hand side ensures that matching is decidable but also that, when it exists, the resulting substitution is unique. This way, the situation is very close to first-order (*i.e.* syntactic) matching.

Definition 4.3.7 (Higher-Order Rewriting System (HRS))**.** *A* Higher-Order Rewriting System *is a set R of rewrite rules.*

The rewrite relation $\rightarrow_R$ on terms is inductively defined as follows:

- *for any $(l \hookrightarrow r) \in R$ and any substitution σ, $\updownarrow_\beta^\eta \sigma(l) \rightarrow_R \updownarrow_\beta^\eta \sigma(r)$;*
- *if $t_1 \rightarrow_R t_2$ and x is a variable, then $x(\ldots, t_1, \ldots) \rightarrow_R x(\ldots, t_2, \ldots)$;*
- *if $t_1 \rightarrow_R t_2$ and f is a constant, then $f(\ldots, t_1, \ldots) \rightarrow_R f(\ldots, t_2, \ldots)$;*
- *if $t_1 \rightarrow_R t_2$, then $\underline{\lambda} x.t_1 \rightarrow_R \underline{\lambda} x.t_2$.*

The standard example of an HRS is the untyped λ-calculus. The signature involves a single base type `Term` and two constants:

$$\texttt{Lam} : (\texttt{Term} \longrightarrow \texttt{Term}) \longrightarrow \texttt{Term}$$

$$\texttt{App} : \texttt{Term} \longrightarrow \texttt{Term} \longrightarrow \texttt{Term}$$

and a single rewrite rule for β-reduction:

$$\text{(beta)} \quad \texttt{App}(\texttt{Lam}(\underline{\lambda} x.X(x)), Y) \hookrightarrow X(Y)$$

4.4 An Encoding of the $\lambda\Pi$-Calculus Modulo into Higher Order Rewrite Systems

4.4.1 Encoding of Terms

We now mimic the encoding of the untyped λ-calculus as an HRS and encode the terms of the $\lambda\Pi$-Calculus Modulo. First we specify the signature.

Definition 4.4.1. *The signature* **Sig**($\lambda\Pi$) *is composed of a single base type* `Term`*, the constants* `Type` *and* `Kind` *of type* `Term`*, the constant* `App` *of type* $\texttt{Term} \longrightarrow \texttt{Term} \longrightarrow \texttt{Term}$*, the constants* `Lam` *and* `Pi` *of type* $\texttt{Term} \longrightarrow (\texttt{Term} \longrightarrow \texttt{Term}) \longrightarrow \texttt{Term}$ *and the constants* `c` *of type* `Term` *for every constant* $c \in \mathscr{C}_O \cup \mathscr{C}_T$.

Then, we define the encoding of $\lambda\Pi$-terms.

Definition 4.4.2 (Encoding of $\lambda\Pi$-term)**.** *The function* $\|.\|$ *from* $\lambda\Pi$*-terms to HRS-terms in the signature* **Sig**($\lambda\Pi$) *is defined as follows:*

$$
\begin{array}{lcl}
\|\textbf{Kind}\| & := & \texttt{Kind} \\
\|\textbf{Type}\| & := & \texttt{Type} \\
\|x\| & := & x \text{ (variable of type } \texttt{Term}) \\
\|c\| & := & \texttt{c} \\
\|uv\| & := & \texttt{App}(\|u\|, \|v\|) \\
\|\lambda x : A.t\| & := & \texttt{Lam}(\|A\|, \underline{\lambda} x.\|t\|) \\
\|\Pi x : A.B\| & := & \texttt{Pi}(\|A\|, \underline{\lambda} x.\|B\|)
\end{array}
$$

This function is a bijection between the untyped terms of the $\lambda\Pi$-Calculus Modulo and well-typed terms of the corresponding HRS.

Lemma 4.4.3. *The function* $\|.\|$ *is a bijection from the* $\lambda\Pi$*-terms to HRS-terms of type* `Term`.

Proof. By induction on the $\beta\overline{\eta}$-normal form. □

Notation 4.4.4. *We write* $\|.\|^{-1}$ *for the inverse of* $\|.\|$.

Remark 4.4.5. $\|.\|$ *and* $\|.\|^{-1}$ *are compositional for substitution.*

- $\|t[x/u]\| = \|t\|[x/\|u\|]$;
- $\|t[x/u]\|^{-1} = \|t\|^{-1}[x/\|u\|^{-1}]$

4.4.2 Higher-Order Rewrite Rules

We have faithfully encoded the terms. The next step is to encode the rewrite rules.

First, we introduce a rule (beta) for β-reduction in the HRS.

$$\text{(beta)} \quad \texttt{App}(\texttt{Lam}(w, \underline{\lambda} x.y(x)), z) \hookrightarrow y(z)$$

We have the following correspondence:

Lemma 4.4.6. *If* $t_1 \to_\beta t_2$*, then* $\|t_1\| \to_{(beta)} \|t_2\|$.

Proof. We have $\|(\lambda x : A.t)u\| = \texttt{App}(\texttt{Lam}(\|A\|, \underline{\lambda} x.\|t\|), \|u\|) \to_{(\text{beta})} \updownarrow_\beta^\eta ((\underline{\lambda} x.\|t\|)(\|u\|)) = \|t\|[x/\|u\|] = \|t[x/u]\|$. Indeed, $\|t\|[x/\|u\|]$ is already a normal form since the variable x cannot have arguments in $\|t\|$ and therefore the substitution does not introduce β-redexes. □

Lemma 4.4.7. *If $t_1 \to_{(beta)} t_2$ and t_1, t_2 have type* `Term`*, then $\|t_1\|^{-1} \to_\beta \|t_2\|^{-1}$.*

Proof. If $t_1 = \updownarrow_\beta^\eta (\sigma(\texttt{App}(\texttt{Lam}(A, \underline{\lambda} x.t(x)), u))) \to_{(\text{beta})} t_2 = \updownarrow_\beta^\eta (\sigma(t(u)))$ and $x \in dom(\sigma)$, then $\|t_1\|^{-1} = (\lambda x : \|\sigma(A)\|^{-1}.\| \updownarrow_\beta^\eta (\sigma(t)\ x)\|^{-1})\|\sigma(u)\|^{-1} \to_\beta \| \updownarrow_\beta^\eta (\sigma(t)\ x)\|^{-1}[x/\|\sigma(u)\|^{-1}] = \| \updownarrow_\beta^\eta (\sigma(t(u)))\|^{-1} = \|t_2\|^{-1}$. □

By encoding rewrite rules in the obvious way (*i.e.*, translating $(u \hookrightarrow v)$ by $(\|u\| \hookrightarrow \|v\|)$), we would get a similar result for Γ-reduction. But, since we want to incorporate rewriting modulo β, we proceed differently.

First, we introduce the notion of uniform terms. These are terms verifying an arity constraint on their free variables.

Definition 4.4.8 (Uniform terms)**.** *A term t is* uniform *for a set V of variables if all occurrences of a variable in V and free in t is applied to the same number of arguments. A uniform term is a term uniform for its free variables.*

Now, we define an encoding for uniform terms.

Definition 4.4.9 (Encoding of uniform terms)**.** *Let V be a set of variables and t be a term uniform in V. The HRS-term $\|u\|_V$ of type* `Term` *is defined as follows:*

$$
\begin{array}{lcl}
\|\textbf{Kind}\|_V & := & \texttt{Kind} \\
\|\textbf{Type}\|_V & := & \texttt{Type} \\
\|x\|_V & := & x, \text{ if } x \notin V \text{ (variable of type } \texttt{Term}) \\
\|c\|_V & := & \texttt{c} \\
\|\lambda x : A.u\|_V & := & \texttt{Lam}(\|A\|_V, \underline{\lambda} x.\|u\|_{V \setminus \{x\}}) \\
\|\Pi x : A.B\|_V & := & \texttt{Pi}(\|A\|_V, \underline{\lambda} x.\|B\|_{V \setminus \{x\}}) \\
\|x\vec{v}\|_V & := & x(\|\vec{v}\|_V), \text{ if } x \in V \text{ (x of type } \texttt{Term}^{n+1} \text{ where } n = |\vec{v}|) \\
\|uv\|_V & := & \texttt{App}(\|u\|_V, \|v\|_V), \text{ if } uv \neq x\ \vec{w} \text{ for } x \notin V
\end{array}
$$

Remark 4.4.10. *The restriction to uniform terms in the previous definition comes from the fact that we need to type the HRS-variables and make sure that the encoding is a HRS-term of type* `Term`*.*

Now, we define an equivalent of patterns for the $\lambda\Pi$-Calculus Modulo.

Definition 4.4.11 ($\lambda\Pi$-patterns)**.** *A $\lambda\Pi$-pattern is a uniform term $t = f\,\vec{u}$ such that, for any variable x, if x occurs free in t then it occurs in a subterm of the form $x\vec{y}$ where $\vec{y}$ is a vector of pairwise distinct variables bound in t.*

Remark 4.4.12. *$\lambda\Pi$-patterns are defined in such a way that their encodings as uniform terms exactly match the definition of patterns in HRSs. If p is a $\lambda\Pi$-pattern, then $\|p\|_{FV(p)}$ is a pattern.*

We now define the encoding of rewrite rules.

Definition 4.4.13 (Encoding of Rewrite Rules)**.** *Let $(u \hookrightarrow v)$ be a rewrite rule such that*

- *u is a $\lambda\Pi$-pattern;*

- $FV(v) \subset FV(u)$;
- *all free occurrences of a variable in u or v are applied to the same number of arguments.*

The encoding of $(u \hookrightarrow v)$ *is* $\|u \hookrightarrow v\| = \|u\|_{FV(u)} \hookrightarrow \|v\|_{FV(u)}$.

The first and second assumptions ensure that $\|u \hookrightarrow v\|$ is indeed a rewrite rule for Definition 4.3.6. The third assumption says that u and v are uniform terms and that, moreover, the arity constraint is the same for u and v. This ensures that the encoding of a variable has a unique simple type.

Definition 4.4.14 (HRS(Γ))**.** *Let* Γ *be a global context whose rewrite rules satisfy the conditions of Definition 4.4.13. We write HRS(*Γ*) for the HRS* $\{\|u \hookrightarrow v\| \mid (u \hookrightarrow v) \in \Gamma\}$ *and HRS(*$\beta\Gamma$*) for* $HRS(\Gamma) \cup \{(\texttt{beta})\}$.

4.5 Rewriting Modulo β

4.5.1 Definition

We are now able to properly define rewriting modulo β. As for usual rewriting, rewriting modulo β is defined on all (untyped) terms of the $\lambda\Pi$-Calculus Modulo.

Definition 4.5.1 (Rewriting Modulo β)**.** *Let* Γ *be a global context whose rewrite rules satisfy the condition of Definition 4.4.13. We say that* t_1 rewrites to t_2 modulo β *(written* $t_1 \to_{\Gamma^b} t_2$*) if* $\|t_1\|$ *rewrites to* $\|t_2\|$ *in HRS(*Γ*). Similarly, we write* $t_1 \to_{\beta\Gamma^b} t_2$ *if* $\|t_1\|$ *rewrites to* $\|t_2\|$ *in HRS(*$\beta\Gamma$*).*

Lemma 4.5.2. $\to_{\beta\Gamma^b} = \to_{\Gamma^b} \cup \to_\beta$.

Proof. Follows from Lemma 4.4.6 and Lemma 4.4.7. □

Lemma 4.5.3. *If* $t_1 \to_\Gamma t_2$*, then* $t_1 \to_{\Gamma^b} t_2$.

Proof. Let $(u \hookrightarrow v) \in \Gamma$. Suppose that $t_1 = \sigma(u) \to_\Gamma \sigma(v) = t_2$. Then, we have $\|t_1\| = \underline{\sigma}(\|u\|) \to \underline{\sigma}(\|v\|) = \|t_2\|$ for $\underline{\sigma} = \{x \mapsto \|\sigma(x)\|\}$. □

4.5.2 Example

Let us look at the example from the introduction. Now we have :

$$\texttt{D}\ (\lambda x : R.\texttt{Exp}\ x) \to_{\Gamma^b} \texttt{fMult}\ (\texttt{D}\ (\lambda x : R.x))\ (\lambda x : R.\texttt{Exp}\ x)$$

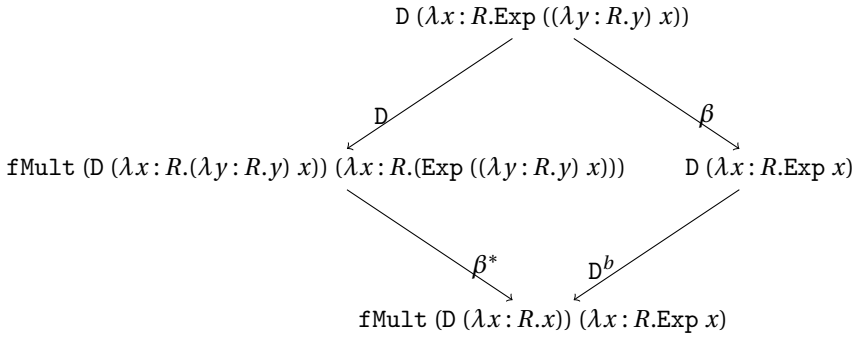
Indeed, for $\sigma = \{f \mapsto \underline{\lambda} y.y\}$ we have

$$\begin{aligned} \|\texttt{D}\ (\lambda x : R.\texttt{Exp}\ x)\| &= \texttt{App}(\texttt{D}, \texttt{Lam}(R, \underline{\lambda} x.\texttt{App}(\texttt{Exp}, x))) \\ &=\updownarrow_\beta^\eta \sigma(\texttt{App}(\texttt{D}, \texttt{Lam}(R, \underline{\lambda} x.\texttt{App}(\texttt{Exp}, f(x))))) \end{aligned}$$

and

$$\begin{aligned} &\|\texttt{fMult}\ (\texttt{D}\ (\lambda x : R.x))\ (\lambda x : R.\texttt{Exp}\ x)\| \\ &\quad = \texttt{App}(\texttt{fMult}, \texttt{App}(\texttt{D}, \texttt{Lam}(R, \underline{\lambda} x.x)), \texttt{Lam}(R, \underline{\lambda} x.\texttt{App}(\texttt{Exp}, x))) \\ &\quad =\updownarrow_\beta^\eta \sigma(\texttt{App}(\texttt{fMult}, \texttt{App}(\texttt{D}, \texttt{Lam}(R, \underline{\lambda} x.f(x))), \texttt{Lam}(R, \underline{\lambda} x.\texttt{App}(\texttt{Exp}, f(x))))) \end{aligned}$$

Therefore, the peak is now joinable and, anticipating a bit, the rewrite system is confluent.



4.5.3 Properties

Rewriting modulo β preserves typing.

Theorem 4.5.4 (Subject Reduction for $\rightarrow_{\Gamma^b}$)**.** *Let Γ be a well-formed global context and Δ a local context well-formed for Γ. If $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \rightarrow_{\Gamma^b} t_2$, then $\Gamma;\Delta \vdash t_2 : T$.*

It directly follows from the following lemma:

Lemma 4.5.5. *If $t_1 \rightarrow_{\Gamma^b} t_2$, then for some t_1' and t_2', we have $t_1 \leftarrow_\beta^* t_1' \rightarrow_\Gamma t_2' \rightarrow_\beta^* t_2$. Moreover, if t_1 is well-typed, then we can choose t_1' such that it is well-typed in the same context.*

Proof. The idea is to lift the β-reductions that occur at the HRS level to the $\lambda\Pi$-Calculus Modulo. Suppose $t_1 \rightarrow_{\Gamma^b} t_2$. For some rewrite rule $(u \hookrightarrow v)$ and (HRS) substitution σ, we have $\updownarrow_\beta^\eta \sigma(u) = \|t_1\|$ and $\updownarrow_\beta^\eta \sigma(v) = \|t_2\|$. We define the ($\lambda\Pi$) substitution $\hat{\sigma}$ as follows: $\hat{\sigma}(x) = \|\sigma(x)\|^{-1}$ if $\sigma(x)$ has type $\mathtt{Term}$; $\hat{\sigma}(x) = \lambda\vec{x} : \vec{A}.\|u\|^{-1}$ if $\sigma(x) = \underline{\lambda}\vec{x}.u$ has type $\mathtt{Term}^n \longrightarrow \mathtt{Term}$ where the A_i are arbitrary types. We have, at the $\lambda\Pi$ level, $\hat{\sigma}(u) \rightarrow_\Gamma \hat{\sigma}(v)$, $\hat{\sigma}(u) \rightarrow_\beta^* t_1$ and $\hat{\sigma}(v) \rightarrow_\beta^* t_2$. If t_1 is well-typed, then the A_i can be chosen so that $\hat{\sigma}(u)$ is also well-typed. □

Another consequence of this lemma is that the rewriting modulo β does not modify the congruence.

Theorem 4.5.6. *The congruence $\equiv_{\beta\Gamma^b}$ generated by $\rightarrow_{\beta\Gamma^b}$ is equal to $\equiv_{\beta\Gamma}$.*

Proof. Follows from Lemma 4.5.3 and Lemma 4.5.5. □

Theorem 4.5.7. *Let Γ be a global context. If HRS($\beta\Gamma$) is confluent, then product compatibility holds for Γ.*

Proof. Assume that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$. Then, by Theorem 4.5.6, $\Pi x : A_1.B_1 \equiv_{\beta\Gamma^b} \Pi x : A_2.B_2$. By confluence, there exist A_0 and B_0 such that $A_1 \rightarrow_{\beta\Gamma^b}^* A_0$, $A_2 \rightarrow_{\beta\Gamma^b}^* A_0$, $B_1 \rightarrow_{\beta\Gamma^b}^* B_0$ and $B_2 \rightarrow_{\beta\Gamma^b}^* B_0$. It follows, by Theorem 4.5.6, that $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$. □

(Empty Global Context)

$$\frac{}{\emptyset\ \mathbf{wf}^{\beta}}$$

(Object Declaration)

$$\frac{\Gamma\ \mathbf{wf}^{\beta} \qquad \Gamma;\emptyset \vdash \underline{U} : \mathbf{Type} \qquad \underline{c} \notin dom(\Gamma)}{\Gamma(\underline{c} : \underline{U})\ \mathbf{wf}^{\beta}}$$

(Type Declaration)

$$\frac{\Gamma\ \mathbf{wf}^{\beta} \qquad \Gamma;\emptyset \vdash K : \mathbf{Kind} \qquad C \notin dom(\Gamma)}{\Gamma(C : K)\ \mathbf{wf}^{\beta}}$$

(Rewrite Rules)

$$\frac{\begin{array}{c}\Gamma\ \mathbf{wf}^{\beta} \qquad \rightarrow_{\beta(\Gamma\Xi)^b} \text{ is confluent} \qquad \Xi = (f_1\,\vec{u_1} \hookrightarrow v_1)\ldots(f_n\,\vec{u_n} \hookrightarrow v_n) \\ (\forall i)\,f_i \text{ is a definable symbol} \\ (\forall i)\Gamma \vdash^{w} f\,\vec{u} \hookrightarrow v_i\end{array}}{\Gamma\Xi\ \mathbf{wf}^{\beta}}$$

Figure 4.1: Weakly well-formedness rules for global contexts

4.5.4 β-Well-Formed Global Contexts

Since product compatibility follows from confluence modulo β, we can update the notion of weakly well-formed global context defined in Section 3.6, weakening the assumption of confluence to confluence of rewriting modulo β. This gives us the notion of β-well-formed global context.

Definition 4.5.8 (β-Well-Formed Contexts)**.** *A global context Γ is* β-well-formed*, if the judgment $\Gamma\ \mathbf{wf}^{\beta}$ is derivable from the inference rules of Figure 4.1.*

Remark 4.5.9. *Weakly well-formed contexts are β-well formed.*

As expected, β-well-formed global contexts are well-typed.

Theorem 4.5.10. *β-well-formed global contexts are safe. Hence they are well-typed.*

Proof. Same proof as Theorem 3.6.9, but product compatibility is obtained by Theorem 4.5.7. □

4.6 Proving Confluence of Rewriting Modulo β

We have shown that the confluence of the rewriting modulo β relation has nice consequences on the type system. Now, we give a powerful criterion for proving confluence.

For Higher-Order Rewrite Systems, as for (first-order) Term Rewriting Systems, there are two general categories of rewrite systems for which confluence results are known: first, the terminating rewrite systems, for which confluence is decidable; second, the left-linear systems, for which several criteria for confluence exist.

Both results are based on the notion of overlapping patterns and higher-order critical pairs.

Notation 4.6.1. *We write $t_{|p}$ for the subterm of t at position p and $t[u]_p$ for the term t where its subterm at position p is replaced by u.*

Definition 4.6.2 (Overlapping Patterns)**.** *Let u and v be two patterns and p be a position in v. Suppose that:*

- *$\vec{x}$ are the free variables of $v_{|p}$ that are bound in v;*
- *σ is a substitution that maps every free variable z_1 of type A in u to $z_2(\vec{x})$ where z_2 is a fresh variable of type $\vec{B} \longrightarrow A$ and $\vec{B}$ are the types of $\vec{x}$;*
- *$\vec{y}$ are the variables free in $\updownarrow^{\eta}_{\beta}(\sigma(u))$ and $v_{|p}$;*
- *$v_{|p}$ is not a variable free in v.*

If there exists a substitution θ with domain $\vec{y}$ such that $\updownarrow^{\eta}_{\beta}(\theta(\underline{\lambda}\vec{x}.\sigma(u))) = \updownarrow^{\eta}_{\beta}(\theta(\underline{\lambda}\vec{x}.v_{|p}))$, then we say that u overlaps with v at position p.

Critical pairs arise from overlaps between two rewrite rules.

Definition 4.6.3 (Higher-Order Critical Pair)**.** *Let $(u_1 \hookrightarrow v_1)$ and $(u_2 \hookrightarrow v_2)$ be two HRS rewrite rules such that u_1 overlaps with u_2 at position p. Let θ be a most general unifier of $\underline{\lambda}\vec{x}.\sigma(u_1)$ and $\underline{\lambda}\vec{x}.u_{2|p}$ (for $\vec{x}$ and σ as in Definition 4.6.2). The pair $(\updownarrow^{\eta}_{\beta}(\theta(u_2[\sigma(u_1)]_p)), \updownarrow^{\eta}_{\beta}(\theta(v_2)))$ is called a critical pair.*

*When the overlap occurs at the root of u_2 (*i.e.*, $p = \epsilon$), then the pair is a root critical pair. Otherwise, it is an inner critical pair.*

We write $a \ltimes b$ when (a, b) is a critical pair.

Theorem 4.6.4 (Nipkow [Nip91])**.** *A terminating higher-order rewrite system is confluent if and only if its critical pairs are joinable.*

For our purpose, Theorem 4.6.4 is of little use since the (beta) rule is not terminating and there is no modularity result (Theorem 1.2.18) for HRSs.

Our second criteria applies to left-linear rewrite systems whose critical pairs are joinable by simultaneous reduction.

Definition 4.6.5 (Simultaneous Reduction)**.** *Let R be a HRS. The simultaneous reduction relation $\multimap$ is the relation on terms defined by:*

- *if $\vec{s} \multimap \vec{t}$ and x is a variable, then $x(\vec{s}) \multimap x(\vec{t})$;*
- *if $\vec{s} \multimap \vec{t}$ and f is a constant, then $f(\vec{s}) \multimap f(\vec{t})$;*
- *if $s \multimap t$, then $\underline{\lambda}x.s \multimap \underline{\lambda}x.t$;*
- *if $u \hookrightarrow v \in R$ and, for all $x, \theta_1(x) \multimap \theta_2(x)$, then $\updownarrow^{\eta}_{\beta}(\theta_1(u)) \multimap \updownarrow^{\eta}_{\beta}(\theta_2(v))$.*

Theorem 4.6.6 (Developpement Closure Theorem (V. van Oostrom [vO95]))**.** *Let R be a left-linear HRS such that:*

- *for every root critical pair $t_1 \ltimes t_2$, there exists t_3 such that $t_1 \multimap t_3$ and $t_2 \to_R t_3$*
- *and, for every inner critical pair $t_1 \ltimes t_2$, $t_1 \multimap t_2$,*

then R is confluent.

This criterion can be adapted for our purpose.

Corollary 4.6.7. *If Γ is a global context such that $HRS(\Gamma)$ satisfies the hypothesis of Theorem 4.6.6, then $\rightarrow_{\Gamma^b}$ is confluent.*

Proof. The rule (beta) is left-linear and cannot overlap with a rewrite rule in $HRS(\Gamma)$. □

4.7 Applications

4.7.1 Parsing and Solving Equations

The context declarations and rewrite rules of Figure 4.2 define a function `to_expr` that parses a function of type `Nat` to `Nat` into an expression of the form $a * x + b$ (represented by the term `mk_expr` a b) where a and b are constants. The left-hand sides of the rewrite rules for `to_expr` are $\lambda\Pi$-patterns. This allows defining `to_expr` by inspecting under the binders.

The function `solve` can then be used to solve the linear equation $a * x + b = 0$. The answer is either `None`, if there is no solution, `All`, if any x is a solution, or `One` m n, if $-m/(n+1)$ is the only solution.

For instance, we have (writing $-\frac{1}{3}$ for `One` 1 2):

$$\texttt{solve}\ (\texttt{to\_expr}(\lambda x : \texttt{Nat.plus}\ x\ (\texttt{plus}\ x\ (\texttt{S}\ x)))) \rightarrow^*_{\beta\Gamma} -\frac{1}{3}.$$

By Theorem 4.6.6 and Theorem 3.2.1, the global context of Figure 4.2 is β-well-formed.

4.7.2 Negation Normal Form

The example of Figure 4.3 is drawn from [Ter03]. It defines a rewrite system to normalize the propositions with respect to negations. It pushes negation inside the propositions and eliminates double negations. In order to push negation inside the quantifiers, the last two rewrite rules feature abstraction on their left-hand side. As previously, Theorem 4.6.6 and Theorem 3.2.1 can be used to show that this global context is well-formed.

4.7.3 Universe Reflection

In [Ass15], Assaf defines a version of the Calculus of Constructions with explicit universe subtyping thanks to an extended notion of conversion generated by a set of rewrite rules. This work can easily be adapted to fit in the framework of the $\lambda\Pi$-Calculus Modulo. Product compatibility holds because the rewriting system is confluent modulo β.

4.8 Compiling Rewrite Rules for Rewriting Modulo β

We now address the problem of compiling pattern matching modulo β to decision trees. This problem has been studied by Maranget [Mar08] in the case of *syntactic* pattern matching. We extend his work to handle pattern matching modulo β.

```
expr                                  :  Type.
mk_expr                               :  Nat ⟶ Nat ⟶ expr.
expr_S                                :  expr ⟶ expr.
expr_S (mk_expr a b)                  ↪  mk_expr a (S b).
expr_P                                :  expr ⟶ expr ⟶ expr.
expr_P (mk_expr a₁ b₁) (mk_expr a₂ b₂)
                                      ↪  mk_expr (plus a₁ a₂) (plus b₁ b₂).

to_expr                               :  (Nat ⟶ Nat) ⟶ expr.
to_expr (λx:Nat.0)                    ↪  mk_expr 0 0.
to_expr (λx:Nat.S (f x))              ↪  expr_S (to_expr (λx:Nat.f x)).
to_expr (λx:Nat.x)                    ↪  mk_expr (S 0) 0.
to_expr (λx:Nat.plus (f x) (g x))     ↪
                  expr_P (to_expr (λx:Nat.f x)) (to_expr (λx:Nat.g x)).

Solution                              :  Type.
All                                   :  Solution.
One                                   :  Nat ⟶ Nat ⟶ Solution.
None                                  :  Solution.
solve                                 :  expr ⟶ Solution.
solve (mk_expr 0 0)                   ↪  All.
solve (mk_expr 0 (S n))               ↪  None.
solve (mk_expr (S n) m)               ↪  One m n.
```

Figure 4.2: Parsing and solving linear equations

```
prop                                  :  Type
Term                                  :  Type
not                                   :  prop ⟶ prop
or                                    :  prop ⟶ prop ⟶ prop
and                                   :  prop ⟶ prop ⟶ prop
forall                                :  (Term ⟶ prop) ⟶ prop
exists                                :  (Term ⟶ prop) ⟶ prop
not (not p)                           ↪  p
not (and p₁ p₂)                       ↪  or (not p₁) (not p₂)
not (or p₁ p₂)                        ↪  and (not p₁) (not p₂)
not (forall (λx:Term.p x))            ↪  exists (λx:Term.not (p x))
not (exists (λx:Term.p x))            ↪  forall (λx:Term.not (p x))
```

Figure 4.3: Negation normal form

If we want to test if a term $\texttt{plus}\ u\ v$ matches one of the four rules below:

$\texttt{plus}\ 0\ n \hookrightarrow n.$
$\texttt{plus}\ n\ 0 \hookrightarrow n.$
$\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$
$\texttt{plus}\ n_1\ (\texttt{S}\ n_2) \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

the naive approach is to try to successively match the term $\texttt{plus}\ u\ v$ against each of the left-hand sides of the four rewrite rules. In the worst case (*i.e.*, if only the last rule is a match), we need to inspect u twice (first to see if it is 0 in the first rule and then to see if it is a successor in the third rule) and v twice as well (for the second and last rules).

Instead, we could inspect only once the shape of each argument u and v, and still be able to say which rewrite rule to apply. This is the idea behind the compilation to decision trees.

The situation is similar if the rewrite rules require matching modulo β:

$\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.0) \hookrightarrow \texttt{mk\_expr}\ 0\ 0.$
$\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.\texttt{S}\ (f\ x)) \hookrightarrow \texttt{expr\_S}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x)).$
$\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.x) \hookrightarrow \texttt{mk\_expr}\ (\texttt{S}\ 0)\ 0.$
$\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.\texttt{plus}\ (f\ x)\ (g\ x)) \hookrightarrow$
$\texttt{expr\_P}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x))\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.g\ x)).$

Here, again, if we try to match $\texttt{to\_expr}\ u$ against each rewrite rule, we will test four times that u is an abstraction.

To be able to deal with matching modulo β, the notion of decision tree must be modified for two reasons.

- First, since there may be abstractions occurring in the left-hand sides of the rewrite rules, we need to handle bound variables.
- Second, in decision trees for regular pattern matching, the leaves correspond to unification problems in *solved form*, that is, of the form $\{x_i = t_i\}$ where the x_i are distinct (if the rewrite rules are left-linear) and none of the x_i occur in any t_i. Such problems always have a unique solution. For matching modulo β the situation is different. The leaves correspond to a set of *flexible-rigid* equations of the form $x_i\ \vec{y}_i = t_i$ where $\vec{y}_i$ are bound variables and none of the x_i occur in any t_i. Such problems have a solution (which is unique) only if the free variables of t_i are among $\vec{y}_i$. The semantics of the decision trees has to be modified in consequence.

4.8.1 Decision Trees

We define the syntax of decision trees and their semantics.

Definition 4.8.1 (Decision Tree)**.** *A decision tree is:*

- *either the tree* **Leaf**(t, ρ, D) *where t is a term, ρ is a partial function from variables to pairs $(\vec{y}, n)$, where $\vec{y}$ is a vector of distinct variables and n is an integer, and D is a decision tree,*
- *or the tree* **Switch**(j, F, D) *where j is an integer, F is a partial function from $\{\lambda\} \cup ((\mathscr{C}_O \cup \mathscr{C}_T) \times \mathbb{N})$ to decision trees and D is a decision tree.*

$$\frac{\begin{array}{l}\rho = \{x \mapsto (\vec{y}_x, j_x)\} \\ \sigma = \{x \mapsto \underline{\lambda}\vec{y}_x.s_{j_x}\} \qquad \forall x \in dom(\rho),\ FV(s_{j_x}) \cap \mathcal{V} \subseteq \vec{y}_x\end{array}}{(\vec{s}, \mathcal{V}) \vdash \mathbf{Leaf}(v, \rho, D) \hookrightarrow v^{\sigma}_{\mathcal{V}}} \textbf{(Leaf)}$$

$$\frac{(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t}{(\vec{s}, \mathcal{V}) \vdash \mathbf{Leaf}(v, \rho, D) \hookrightarrow t} \textbf{(Leaf-Else)}$$

$$\frac{F(u,n) = D_u \qquad s_j = u\vec{v} \qquad |\vec{v}| = n \qquad (\vec{s}.\vec{v}, \mathcal{V}) \vdash D_u \hookrightarrow t}{(\vec{s}, \mathcal{V}) \vdash \mathbf{Switch}(j, F, D) \hookrightarrow t} \textbf{(Switch-C)}$$

$$\frac{F(\lambda) = D_\lambda \qquad s_j = \lambda x : A.u \qquad (\vec{s}.A.u, \mathcal{V} \cup \{x\}) \vdash D_\lambda \hookrightarrow t}{(\vec{s}, \mathcal{V}) \vdash \mathbf{Switch}(j, F, D) \hookrightarrow t} \textbf{(Switch-L)}$$

$$\frac{\begin{array}{c}s_j \neq \lambda x : A.u \\ (\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t \qquad s_j \neq u\vec{v} \text{ with } n = |\vec{v}| \text{ and } (u,n) \in dom(F)\end{array}}{(\vec{s}, \mathcal{V}) \vdash \mathbf{Switch}(j, F, D) \hookrightarrow t} \textbf{(Switch-Else)}$$

Figure 4.4: Operational semantics for decision trees

- *or the tree* **Fail**.

We now define the semantics of decision trees. We use the following notation to avoid, as much as possible, the reference to the construction of rewriting modulo β through HRS when working with $\lambda\Pi$-terms.

Notation 4.8.2. *Let $\mathcal{V}$ be a set of variables, t be a term uniform in $FV(t) \setminus \mathcal{V}$ and σ be a HRS substitution such that $dom(\sigma) \cap \mathcal{V} = \emptyset$ and $\sigma(\|t\|_{\mathcal{V}})$ is a preterm (of type* `Term`*).*

We write $t^{\sigma}_{\mathcal{V}}$ for the term $\| \updownarrow^{\eta}_{\beta} (\sigma(\|t\|_{\mathcal{V}}))\|^{-1}$.

For example, if $t =$ `Exp` $(f\ x), \mathcal{V} = \{x\}$ and $\sigma = f \mapsto \underline{\lambda} y.y$, then we have $t^{\sigma}_{\mathcal{V}} =$ `Exp` x.

Notation 4.8.3 (Head Reduction)**.** *We write $t_1 \to^{h}_{\Gamma^b} t_2$ if $t_1 \to_{\Gamma^b} t_2$ and the reduction occurs at the root of t_1 and not in a subterm.*

Remark 4.8.4. *$t_1 \to^{h}_{\Gamma^b} t_2$ for the rewrite rule $u \hookrightarrow v$ if and only if there is a HRS-substitution σ such that $t_1 = u^{\sigma}_{\emptyset}$ and $t_2 = v^{\sigma}_{\emptyset}$.*

Definition 4.8.5. *The operational semantics for decision trees is given Figure 4.4 in the form of a judgment $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$ where $\vec{s}$ is a vector of terms (the stack), $\mathcal{V}$ is a set of variables, D is a decision tree and t is a term.*

If the judgment $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$ is derivable, we say that D reduces to t in the context $(\vec{s}, \mathcal{V})$.

Intuitively, a decision tree is meant to be *executed* with a stack $\vec{s}$ (*i.e.*, a vector of terms s_j) and a set of variables $\mathcal{V}$.

The decision tree $\mathbf{Leaf}(v, \{x \mapsto (\vec{y}_x, j_x)\}, D)$ corresponds to a potential match. If the unification problem $\{\lambda\vec{z}.x(\vec{y}_x) \equiv_\beta \lambda\vec{z}.\|s_{j_x}\|\}$ with $\mathcal{V} = \{\vec{z}\}$ has a solution σ, then the result of the matching is $v^{\sigma}_{\mathcal{V}}$. This situation corresponds to the rule **(Leaf)**. If

there are no solutions, then we continue, executing the decision tree D **(Leaf-Else)**. Remark that if there are no bound variables (*i.e.*, $\mathcal{V}$ is empty), then a solution always exists.

The decision tree **Switch**(j, F, D) is a branching node in our search for a match. It tells us to look at the shape of s_j. If s_j is an abstraction and $F(\lambda) = D_\lambda$ is defined, then we continue with D_λ **(Switch-L)**. If $s_j = u\vec{v}$ is a constant application or a bound variable application, and $F(u, n) = D_u$ is defined for $n = |\vec{v}|$, then we continue with D_u **(Switch-C)**. Otherwise we continue with D **(Switch-Else)**.

The construction **Fail** corresponds to the situation where the term does not match any rewrite rule.

4.8.2 From Rewrite Rules to Decision Tree

Let Γ be a global context for which $\to_{\beta\Gamma}$ is well-defined. The goal of this section is to define a function $\mathbf{CC}_\Gamma$ from constants to decision trees such that:

$$(\vec{u}, \emptyset) \vdash \mathbf{CC}_\Gamma(f) \hookrightarrow t \text{ if and only if } f\,\vec{u} \to^{h}_{\Gamma^b} t.$$

We call the function $\mathbf{CC}_\Gamma$ the compilation function. It will use matrices.

Here, (pairs of) matrices are concise representations of sets of rewrite rules for the same constant and with the same number of arguments.

Definition 4.8.6 (Matrix Patterns)**.** *A* matrix pattern *is either a* $\lambda\Pi$*-pattern or the symbol '*$*$*' which we call* joker.

Definition 4.8.7 (Matrix)**.** *A* matrix *is a pair* $M = (U \hookrightarrow V)$ *where* U *is an array of matrix patterns of size* $m * n$ *(*m *lines,* n *columns) and* V *is a vector of terms of size* m*.*

The line i *of* $(U \hookrightarrow V)$ *is the pair* $(\vec{u} \hookrightarrow v)$ *where* $\vec{u} = U(i)$ *and* $v = V(i)$*.*

The size of the matrix $(U \hookrightarrow V)$ *is* $m * n$*.*

We now give the correspondence between rewrite rules and matrices.

Definition 4.8.8 (Rewrite Rules as a Matrix)**.** *A set of rewrite rules* $(f\,\vec{u}_i \hookrightarrow v_i)_{1 \le i \le m}$ *for the same constant* f *and with the same number* n *of arguments (for all* i*,* $|\vec{u}_i| = n$*) can be represented by the matrix* $(U \hookrightarrow V)$ *of size* $m * n$ *defined as follows:*

- $U(i, j) = u_{i,j}$*, for* $1 \le i \le m$ *and* $1 \le j \le n$*,*
- $V(i) = v_i$*, for* $1 \le i \le m$*.*

The rewrite rule $(f\,\vec{u}_i \hookrightarrow v_i)$ *corresponds to the line* $(\vec{u}_i \hookrightarrow v_i)$*. Remark that the constant* f *is not stored in the matrix.*

Examples The matrix representing the rewrite rules for `plus` is

$$M_{\texttt{plus}} = \begin{pmatrix} 0 & n \\ n & 0 \\ \texttt{S}\ n_1 & n_2 \\ n_1 & \texttt{S}\ n_2 \end{pmatrix} \hookrightarrow \begin{pmatrix} n \\ n \\ \texttt{S}\ (\texttt{plus}\ n_1\ n_2) \\ \texttt{S}\ (\texttt{plus}\ n_1\ n_2) \end{pmatrix}.$$

The matrix representing the rewrite rules for `to_expr` is

$$M_{\texttt{to\_expr}} = \begin{pmatrix} \lambda x : \texttt{Nat}.0 \\ \lambda x : \texttt{Nat}.\texttt{S}\ (f\ x) \\ \lambda x : \texttt{Nat}.x \\ \lambda x : \texttt{Nat}.\texttt{plus}\ (f\ x)\ (g\ x) \end{pmatrix} \hookrightarrow \begin{pmatrix} \texttt{mk\_expr}\ 0\ 0 \\ \texttt{expr\_S}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x)) \\ \texttt{mk\_expr}\ (\texttt{S}\ 0)\ 0 \\ \texttt{expr\_P}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x))\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.g\ x)) \end{pmatrix}.$$

We know how to see a set of rewrite rules as a matrix. Now, we explain how to transform a matrix into a decision tree. This compilation process uses two operations on matrices: the extraction of the default matrix and the specialization of a matrix.

Definition 4.8.9 (Default Matrix)**.** *Let $M = (U \hookrightarrow V)$ be a matrix of size $m * n$, $1 \leq j \leq n$ and $\mathcal{V}$ a set of variables.*

The default matrix *for M at Column j with bound variables $\mathcal{V}$ (written $\mathscr{D}(\mathcal{V}, j, U \hookrightarrow V)$) is the matrix built from the lines i of M where $U(i)(j)$ is either a joker or a term $x\vec{y}$ with x a variable not in $\mathcal{V}$.*

Examples

$$\mathscr{D}(\emptyset, 1, \begin{pmatrix} 0 & n \\ n & 0 \\ \mathtt{S}\ n_1 & n_2 \\ n_1 & \mathtt{S}\ n_2 \end{pmatrix} \hookrightarrow \begin{pmatrix} n \\ n \\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2) \\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2) \end{pmatrix}) = \begin{pmatrix} n & 0 \\ n_1 & \mathtt{S}\ n_2 \end{pmatrix} \hookrightarrow \begin{pmatrix} n \\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2) \end{pmatrix}$$

$$\mathscr{D}(\{x\}, 3, \begin{pmatrix} * & \mathtt{Nat} & 0 \\ * & \mathtt{Nat} & \mathtt{S}\ (f\ x) \\ * & \mathtt{Nat} & x \\ * & \mathtt{Nat} & \mathtt{plus}\ (f\ x)\ (g\ x) \end{pmatrix} \hookrightarrow \begin{pmatrix} \mathtt{mk\_expr}\ 0\ 0 \\ \mathtt{expr\_S}\ (\mathtt{to\_expr}\ (\lambda x : \mathtt{Nat}. f\ x)) \\ \mathtt{mk\_expr}\ (\mathtt{S}\ 0)\ 0 \\ \mathtt{expr\_P}\ \ldots \end{pmatrix}) = \emptyset$$

Definition 4.8.10 (Specialized Matrix)**.** *Let $M = (U \hookrightarrow V)$ be a matrix of size $m * n$, $1 \leq j \leq n$, $\mathcal{V}$ be a set of variables, u be a constant or a variable in $\mathcal{V}$ and k be an integer.*

The specialized matrix $\mathscr{S}_{(u,k)}(\mathcal{V}, j, M)$ *is built from M as follows:*

- *we keep only the lines i of M where $U(i, j)$ is either a joker, or a term $x\vec{y}$ with x a variable not in $\mathcal{V}$, or a term $u\vec{v}$ with $|\vec{v}| = k$;*
- *we add k new columns at the end of U: if $U(i, j) = u\vec{v}$, then we fill the new columns with $\vec{v}$; otherwise, we fill the new columns with jokers;*
- *if $U(i, j) = u\vec{v}$, then we replace it by a joker.*

The specialized matrix $\mathscr{S}_{\lambda}(\mathcal{V}, j, M)$ *is built from M as follows:*

- *we keep only the lines i of M where $U(i, j)$ is either a joker, or a term $x\vec{y}$ with x a variable not in $\mathcal{V}$, or an abstraction;*
- *we add 2 new columns at the end of U: if $U(i, j) = \lambda x : A.w$, then we fill the two new columns by A and w; otherwise, we fill the new columns with jokers;*
- *if $U(i, j)$ is an abstraction, we replace it by a joker.*

Remark 4.8.11. *Jokers are used to mark positions in the matrix that have already been matched or that should match anything.*

Examples

$$\mathscr{S}_{(\mathtt{S},1)}(\emptyset,1,\begin{pmatrix}0 & n\\ n & 0\\ \mathtt{S}\ n_1 & n_2\\ n_1 & \mathtt{S}\ n_2\end{pmatrix}\hookrightarrow\begin{pmatrix}n\\ n\\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2)\\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2)\end{pmatrix}) = \begin{pmatrix}n & 0 & *\\ * & n_2 & n_1\\ n_1 & \mathtt{S}\ n_2 & *\end{pmatrix}\hookrightarrow\begin{pmatrix}n\\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2)\\ \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2)\end{pmatrix}$$

$$\mathscr{S}_{\lambda}(\emptyset,1,\begin{pmatrix}\lambda x:\mathtt{Nat}.0\\ \lambda x:\mathtt{Nat}.\mathtt{S}\ (f\ x)\\ \lambda x:\mathtt{Nat}.x\\ \lambda x:\mathtt{Nat}.\mathtt{plus}\ (f\ x)\ (g\ x)\end{pmatrix}\hookrightarrow\begin{pmatrix}\mathtt{mk\_expr}\ \ldots\\ \mathtt{expr\_S}\ \ldots\\ \mathtt{mk\_expr}\ \ldots\\ \mathtt{expr\_P}\ \ldots)\end{pmatrix}) = \begin{pmatrix}* & \mathtt{Nat} & 0\\ * & \mathtt{Nat} & \mathtt{S}\ (f\ x)\\ * & \mathtt{Nat} & x\\ * & \mathtt{Nat} & \mathtt{plus}\ (f\ x)\ (g\ x)\end{pmatrix}\hookrightarrow\begin{pmatrix}\mathtt{mk\_expr}\ \ldots\\ \mathtt{expr\_S}\ \ldots\\ \mathtt{mk\_expr}\ \ldots\\ \mathtt{expr\_P}\ \ldots\end{pmatrix}$$

The compilation of matrices to decision trees is defined as follows. Since several decision trees may correspond to one matrix, we define the result of the compilation as a set of decision trees.

Definition 4.8.12 (Compilation of Matrices)**.** *Let $M = (U \hookrightarrow V)$ be a matrix and $\mathscr{V}$ be a set of variables.*

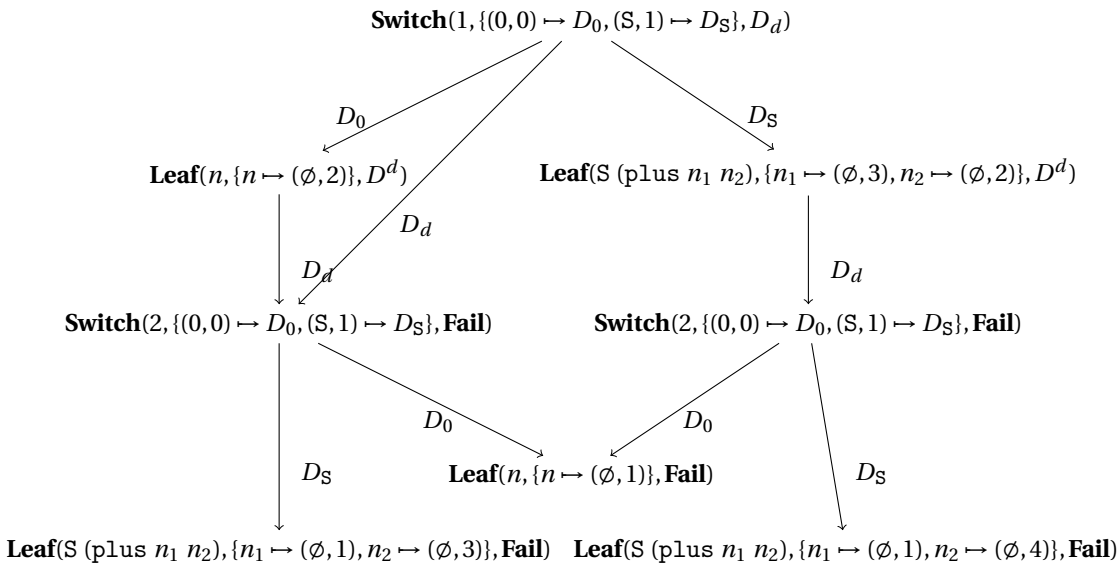
The set of decision trees $\mathbf{CC}(\mathscr{V}, M)$ is defined as follows:

- **Fail** $\in \mathbf{CC}(\mathscr{V}, \emptyset)$.
- **Leaf**$(v, \rho, D) \in \mathbf{CC}(\mathscr{V}, M)$*, if, for some line i of M:*
  - *for all j, either $U(i)(j)$ is a joker, or $U(i)(j) = x_j \vec{y}_j$ where x_j is a variable not in $\mathscr{V}$, $\vec{y}_j \subset \mathscr{V}$ and $\rho(x_j) = (\vec{y}_j, j)$;*
  - *$D \in \mathbf{CC}(\mathscr{V}, M_2)$ where M_2 is M without the line i,*
  - *and $v = V(i)$.*
- **Switch**$(j, F, D) \in \mathbf{CC}(\mathscr{V}, M)$*, if*
  - *$1 \leq j \leq n$,*
  - *F is a function with domain $\{(u, n) \mid \exists i.U(i)(j) = u\vec{v},$ where u is a constant or a variable in $\mathscr{V}$ and $|\vec{v}| = n\} \cup \{\lambda \mid \exists i.U(i)(j) = \lambda x : A.u\}$ such that*
    - *$F(u, n) \in \mathbf{CC}(\mathscr{V}, \mathscr{S}_{(u,n)}(\mathscr{V}, j, M))$*
    - *and $F(\lambda) \in \mathbf{CC}(\mathscr{V} \cup \{x\}, \mathscr{S}_{\lambda}(\mathscr{V}, j, M))$,*
  - *and $D \in \mathbf{CC}(\mathscr{V}, \mathscr{D}(\mathscr{V}, j, M))$.*

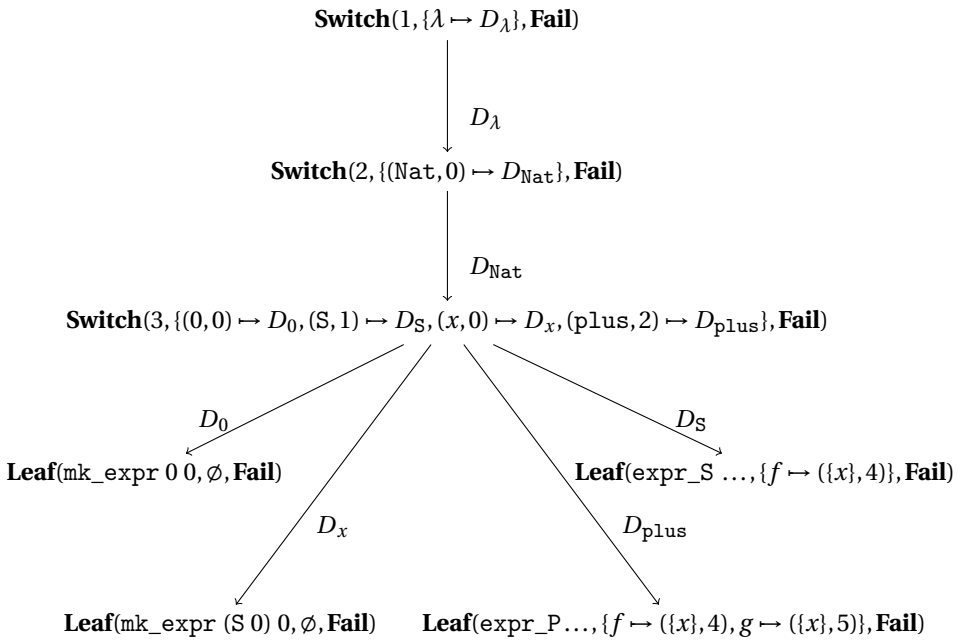
Two parameters make $\mathbf{CC}(\mathscr{V}, M)$ be a set and not a single decision tree. First, if M has several lines containing only jokers and free variables applications, then, depending of the chosen line, several **Leaf** decision trees are possible. Second, if the width of the matrix is strictly greater than one, then we can choose to *switch* on different columns. These two parameters may be used to optimize either the size of the decision tree or its depth [Mar08]. We do not detail this here.

We now give some examples of decision trees.

Symmetric Addition A decision tree in $\mathbf{CC}(\emptyset, M_{\texttt{plus}})$:



Parsing of Expressions A decision tree in $\mathbf{CC}(\emptyset, M_{\texttt{to\_expr}})$:



4.8.3 Soundness and Completeness

Recall that our goal is to define a function $\mathbf{CC}_\Gamma$ verifying the following properties:

- **(Soundness)** If $(\vec{u}, \emptyset) \vdash \mathbf{CC}_\Gamma(f) \hookrightarrow t$, then $f\,\vec{u} \to^h_{\Gamma_b} t$.

- **(Completeness)** If $f\,\vec{u} \to^{h}_{\Gamma^b} t$, then $(\vec{u}, \emptyset) \vdash \mathbf{CC}_\Gamma(f) \hookrightarrow t$.

Let Γ be a global context such that:

- $\to_{\Gamma^b}$ is well-defined (*i.e.*, the rewrite rules satisfy the assumptions of Definition 4.4.13);
- the rewrite rules are left-linear;
- all the rewrite rules for a single symbol have the same number of arguments.

We can take $\mathbf{CC}_\Gamma = f \mapsto D \in \mathbf{CC}(\emptyset, M_f)$ for M_f the matrix representing the rewrite rules whose head symbol is f.

We now prove the soundness and completeness properties for $\mathbf{CC}_\Gamma$, using a notion a matching for vectors of terms.

Definition 4.8.13 (Match)**.** *Let $\vec{s}$ be a vector of terms, $\mathcal{V}$ be a set of variables and $M = (U \hookrightarrow V)$ be a matrix.*

Match$(\vec{s}, \mathcal{V}, M)$ *is the set of terms t such that, for some integer i and HRS-substitution σ:*

- *for all j, $u = U(i,j)$ is either a joker or $s_j = u^\sigma_{\mathcal{V}}$;*
- *and $t = v^\sigma_{\mathcal{V}}$ for $v = V(i)$.*

Remark 4.8.14. *Let Γ be a global context and M be the matrix corresponding to the rules $(f\,\vec{u} \hookrightarrow v)$ in Γ.*

*We have $f\,\vec{s} \to_{\Gamma^b} t$ (by a toplevel reduction) if and only if $t \in$ **Match**$(\vec{s}, \emptyset, M)$.*

Notation 4.8.15. *We write $M_1 \subset M_2$ if M_2 contains at least all the lines of M_1.*

Remark 4.8.16. *If $M_1 \subset M_2$, then* **Match**$(\vec{s}, \mathcal{V}, M_1) \subset$ **Match**$(\vec{s}, \mathcal{V}, M_2)$.

Soundness for Matrix Specialization and Default Matrix

Lemma 4.8.17 (Soundness of $\mathscr{S}_{(u,n)}$)**.** *If $s_j = u\,\vec{v}$ and $|\vec{v}| = n$, with u a constant or a variable in $\mathcal{V}$, then* **Match**$(\vec{s}, \mathcal{V}, M) =$ **Match**$(\vec{s}.\vec{v}, \mathcal{V}, \mathscr{S}_{(u,n)}(\mathcal{V}, j, M))$.

Proof. By definition of $\mathscr{S}_{(u,n)}(\mathcal{V}, j, M)$. □

Lemma 4.8.18 (Soundness of $\mathscr{S}_\lambda$)**.** *If $s_j = \lambda x : A.u$, then* **Match**$(\vec{s}, \mathcal{V}, M) =$ **Match**$(\vec{s}.A.u, \mathcal{V} \cup \{x\}, \mathscr{S}_\lambda(V, j, M))$.

Proof. By definition of $\mathscr{S}_\lambda(\mathcal{V}, j, M)$. □

Lemma 4.8.19 (Soundness of $\mathscr{D}$)**.** *Suppose that:*

- *if $s_j = \lambda x : A.u$, then, for all i, $U(i,j)$ is not an abstraction;*
- *if $s_j = u\,\vec{v}$ with u a constant or a variable in $\mathcal{V}$, then, for all i, $U(i,j) \neq u\,\vec{w}$ with $|\vec{v}| = |\vec{w}|$.*

Then, we have **Match**$(\vec{s}, \mathcal{V}, M) =$ **Match**$(\vec{s}, \mathcal{V}, \mathscr{D}(\mathcal{V}, j, M))$.

Proof. By definition of $\mathscr{D}(\mathcal{V}, j, M)$. □

Soundness of the Compilation to Decision Trees

Theorem 4.8.20 (Soundness of **CC**)**.** *Let $\vec{s}$ be a vector of terms, $\mathcal{V}$ be a set of variables, D be a decision tree, $M = U \hookrightarrow V$ be a matrix and t be a term.*

If $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$ and $D \in \mathbf{CC}(\mathcal{V}, M)$, then $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$.

Proof. By induction on $\vdash$.

- **(Leaf)** Suppose that $D = \mathbf{Leaf}(v, \rho, D_0)$, $t = v^\sigma_{\mathcal{V}}$, $\rho = x \mapsto (\vec{y}_x, j_x)$, $\forall x, FV(s_{j_x}) \cap \mathcal{V} \subset \vec{y}_x$ and $\sigma = x \mapsto \underline{\lambda}.\vec{y}_x.s_{j_x}$.

  Since $D \in \mathbf{CC}(\mathcal{V}, M)$, there exists i such that, for all j, $U(i,j)$ is either a joker or $z\vec{y}_z$, for variables z and $\vec{y}_z$, and $v = V(i)$.

  It follows that, for all j, $u = U(i,j)$ is either a joker or $s_j = u^\sigma_{\mathcal{V}}$ and $t = v^\sigma_{\mathcal{V}} \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$.

- **(Leaf-Else)** Suppose that $D = \mathbf{Leaf}(v, \rho, D_0)$ and $(\vec{s}, \mathcal{V}) \vdash D_0 \hookrightarrow t$.

  Since $D \in \mathbf{CC}(\mathcal{V}, M)$, we have $D_0 \in \mathbf{CC}(\mathcal{V}, M_0)$ for $M_0 \subset M$.

  By induction hypothesis, $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M_0)$.

  By Remark 4.8.16, $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$.

- **(Switch-C)** Suppose that $D = \mathbf{Switch}(j, F, D_0)$, $F(u,n) = D_u$, $s_j = u\vec{v}$, $|\vec{v}| = n$ and $(\vec{s}.\vec{v}, \mathcal{V}) \vdash D_u \hookrightarrow t$.

  Since $D \in \mathbf{CC}(\mathcal{V}, M)$, we have $D_u \in \mathbf{CC}(\mathcal{V}, \mathscr{S}_{(u,n)}(\mathcal{V}, j, M))$.

  By induction hypothesis, we have $t \in \mathbf{Match}(\vec{s}.\vec{v}, \mathcal{V}, \mathscr{S}_{(u,n)}(\mathcal{V}, j, M))$.

  By Lemma 4.8.17, we have $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$.

- **(Switch-L)** Suppose that $D = \mathbf{Switch}(j, F, D_0)$, $F(\lambda) = D_\lambda$, $s_j = \lambda x : A.u$ and $(\vec{s}.A.u, \mathcal{V} \cup \{x\}) \vdash D_\lambda \hookrightarrow t$.

  Since $D \in \mathbf{CC}(\mathcal{V}, M)$, we have $D_\lambda \in \mathbf{CC}(\mathcal{V} \cup \{x\}, \mathscr{S}_\lambda(V, j, M))$.

  By induction hypothesis, $t \in \mathbf{Match}(\vec{s}.A.u, \mathcal{V} \cup \{x\}, \mathscr{S}_\lambda(V, j, M))$.

  By Lemma 4.8.18, $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$.

- **(Switch-Else)** Suppose that $D = \mathbf{Switch}(j, F, D_0)$ and $(\vec{s}, \mathcal{V}) \vdash D_0 \hookrightarrow t$.

  Since $D \in \mathbf{CC}(\mathcal{V}, M)$, we have $D_0 \in \mathbf{CC}(\mathcal{V}, \mathscr{D}(\mathcal{V}, j, M))$.

  By induction hypothesis, $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, \mathscr{D}(\mathcal{V}, j, M))$.

  Therefore, since $\mathscr{D}(\mathcal{V}, j, M) \subset M$, by Remark 4.8.16, $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$.

□

Corollary 4.8.21 (Soundness fo $\mathbf{CC}_\Gamma$)**.** *If $(\vec{s}, \emptyset) \vdash \mathbf{CC}_\Gamma(f) \hookrightarrow t$, then $f\vec{s} \to^h_{\Gamma b} t$.*

Proof. By Remark 4.8.14 and Theorem 4.8.20. □

Completeness of the Compilation to Decision Trees

Theorem 4.8.22 (Completeness of **CC**)**.** *If* $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M)$ *and* $D \in \mathbf{CC}(\mathcal{V}, M)$*, then* $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$.

Proof. We proceed by induction on $\mathbf{CC}(\mathcal{V}, M)$. Let $D \in \mathbf{CC}(\mathcal{V}, M)$.

- Since $\mathbf{Match}(\vec{s}, \mathcal{V}, M) \neq \emptyset$, then $M \neq \emptyset$ and $D \neq \mathbf{Fail}$.
- Suppose that $D = \mathbf{Leaf}(v, \rho, D_0)$; then, there exists i such that, for all $j, U(i,j)$ is either a joker or is equal to $x_j \vec{y}_j$ with $x_j \notin \mathcal{V}$ and $\vec{y}_j \subset \mathcal{V}$, $\rho(x_j) = (\vec{y}_j, j)$, and $D_0 \in \mathbf{CC}(\mathcal{V}, M_0)$ for M_0 the matrix M without the line i.

  If $\forall x_j, FV(s_j) \cap \mathcal{V} \subset \vec{y}_j$ and $t = v^{\sigma}_{\mathcal{V}}$, for $\sigma = x_j \mapsto \underline{\lambda}\, \vec{y}_j . s_j$, then, by **(Leaf)**, we have $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$.

  Otherwise, $\vec{s}$ does not match the line i and $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, M_0)$. By induction hypothesis, $(\vec{s}, \mathcal{V}) \vdash D_0 \hookrightarrow t$ and, by **(Leaf-Else)**, $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$.
- Suppose that $D = \mathbf{Switch}(j, F, D_0)$ with $D_0 \in \mathbf{CC}(\mathcal{V}, \mathscr{D}(\mathcal{V}, j, M))$, $dom(F) = \{ (u,n) \mid \text{for some } i,\ U(i,j) = u\vec{v},\ u \text{ is a constant or a variable in } \mathcal{V} \text{ and } |\vec{v}| = n \} \cup \{ \lambda \mid \text{for some } i,\ U(i,j) = \lambda x : A.u \}$ and, when it is defined, we have $F(u,n) \in \mathbf{CC}(\mathcal{V}, \mathscr{S}_{(u,n)}(\mathcal{V}, j, M))$ and $F(\lambda) \in \mathbf{CC}(\mathcal{V} \cup \{x\}, \mathscr{S}_{\lambda}(\mathcal{V}, j, M))$.
  - If $s_j = u\vec{v}$, $|\vec{v}| = n$ and $(u,n) \in dom(F)$, then, by Lemma 4.8.17, $t \in \mathbf{Match}(\vec{s}, \mathcal{V}, \mathscr{S}_{(u,n)}(\mathcal{V}, j, M))$.
    By induction hypothesis, we have $(\vec{s}, \mathcal{V}) \vdash F(u,n) \hookrightarrow t$.
    It follows, by **(Switch-C)**, that $(\vec{s}, \mathcal{V}) \vdash D \hookrightarrow t$.
  - if $s_j = \lambda x : A.u$ and $\lambda \in dom(F)$, then, by Lemma 4.8.18, $t \in \mathbf{Match}(\vec{s}.A.u, \mathcal{V} \cup \{x\}, \mathscr{S}_{\lambda}(\mathcal{V}, j, M))$.
    By induction hypothesis, we have $(\vec{s}, \mathcal{V} \cup \{x\}) \vdash F(\lambda) \hookrightarrow t$.
    It follows, by **(Switch-L)**, that $(\vec{s}, V) \vdash D \hookrightarrow t$.
  - otherwise, by Lemma 4.8.19, we have $t \in \mathbf{Match}(\vec{s}, V \cup \{x\}, \mathscr{D}(V, j, M))$
    By induction hypothesis, we have $(\vec{s}, V) \vdash D_0 \hookrightarrow t$.
    It follows, by **(Switch-Else)**, that $(\vec{s}, V) \vdash D \hookrightarrow t$.

□

Corollary 4.8.23 (Completeness for $\mathbf{CC}_{\Gamma}$)**.** *If* $f\vec{s} \to^{h}_{\Gamma_b} t$ *, then* $(\vec{s}, \emptyset) \vdash \mathbf{CC}_{\Gamma}(f) \hookrightarrow t$.

Proof. By Remark 4.8.14 and Theorem 4.8.22. □

4.9 Conclusion

We have defined a notion of rewriting modulo β for the $\lambda\Pi$-Calculus Modulo. We achieved this by encoding the $\lambda\Pi$-Calculus Modulo into the framework of Higher-Order Rewrite Systems. As a consequence, we also make the confluence results for HRSs available for the $\lambda\Pi$-Calculus Modulo. We proved that rewriting modulo β preserves typing and that confluence of rewriting modulo β implies product-compatibility, allowing us to generalize the notion of weakly well-formed global context to β-well-formed global contexts.

We have also studied an efficient implementation of rewriting modulo β through the compilation of rewrite rules to decision trees, extending the work of Maranget [Mar08].

A natural extension of this work would be to consider rewriting modulo $\beta\eta$ as in Higher-Order Rewrite Systems. This requires extending the conversion with η-reduction. But, as remarked in [Geu92] (attributed to Nederpelt), $\rightarrow_{\beta\eta}$ is not confluent on untyped terms as the following example shows:

$$\lambda y : B.y \leftarrow_{\eta} \lambda x : A.(\lambda y : B.y)x \rightarrow_{\beta} \lambda x : A.x$$

Therefore properties such as product compatibility need to be proved another way.

For the $\lambda\Pi$-calculus a notion of higher-order pattern matching has been proposed [Pie08] based on Contextual Type Theory (CTT) [NPP08]. This notion is similar to ours. However, it is defined using the notion of meta-variable (which is native in CTT) instead of a translation into HRSs.

In [Bla15], Blanqui studies the termination of the combination of β-reduction with a set of rewrite rules with matching modulo $\beta\eta$ in the polymorphic λ-calculus. His definition of rewriting modulo $\beta\eta$ is direct and does not use any encoding. This leads to a slightly different notion a rewriting modulo β. For instance, $\mathtt{D}(\lambda : R.\mathtt{Exp}\ x)$ would reduce to $\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.(\lambda y : R.y)\ x))\ (\lambda x : R.\mathtt{Exp}\ ((\lambda y : R.y)\ x))$ instead of $\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)$. It would be interesting to know whether the two definitions are equivalent with respect to confluence, that is to say if, for the same set of rewrite rules, the rewrite relation for his definition of rewriting modulo β is confluent if and only if the rewrite relation for our definition of rewriting modulo β is confluent.

Chapter 5

Non-Left-Linear Systems

Résumé Ce chapitre considère les règles de réécriture non linéaires à gauche. Combinées avec la β-réduction, ces règles génèrent généralement un système de réécriture non confluent. Ceci est un problème car la confluence est notre outil principal pour prouver la compatibilité du produit. On prouve que la propriété de compatibilité du produit est toujours vérifiée (même sans la confluence) lorsque les règles de réécriture sont toutes au niveau objet. Ensuite on étudie cette propriété en présence de règles non linéaires à gauche et de règles au niveau type. Pour cela, on introduit une variante du $\lambda\Pi$-Calcul Modulo où la conversion est contrainte par une notion de typage faible.

5.1 Introduction

In the previous chapters, we used the confluence of the rewrite relation each time we proved that a global context is well-typed, and, in particular, that it satisfies product compatibility. More precisely, we used either the confluence of $\rightarrow_{\beta\Gamma}$ (Theorem 2.6.11) or the confluence of $\rightarrow_{\beta\Gamma^b}$ (Theorem 4.5.7).

Relying on the confluence of the rewriting relation basically prevents us from considering non left-linear rewrite rules. Indeed, all the criteria that we have at hand for proving the confluence of $\rightarrow_{\beta\Gamma}$ (Theorem 1.4.7) or $\rightarrow_{\beta\Gamma^b}$ (Theorem 4.6.6) require the rewrite rules to be left-linear. Worse, the rewrite system generated by a non left-linear rewrite rule together with β-reduction is almost always non-confluent (see Section 1.4.3).

One part of this problem has been solved in Chapter 3, where we have shown how to eliminate non left-linearity of rewrite rules when it is due to typing constraints. For instance, the non left-linear rewrite rule

$\texttt{tail}\ n\ (\texttt{vcons}\ n\ e\ l) \hookrightarrow l.$

can be replaced by its linearized version

$\texttt{tail}\ n_1\ (\texttt{vcons}\ n_2\ e\ l) \hookrightarrow l.$

because both left-hand sides match the same *well-typed* terms. However, this technique does not apply if the non left-linearity is intended as in:

$\texttt{eq}\ n\ n \hookrightarrow \texttt{true}.$

Obviously, this rewrite rule cannot be replaced by:

$\texttt{eq}\ n_1\ n_2 \hookrightarrow \texttt{true}$.

since both rewrite rules do not have the same behavior.

Therefore, the question remains: can we prove product compatibility in presence of non left-linear rewrite rules or, more generally, when the rewrite system is not confluent? And, if so, how to proceed?

Barbanera, Geuvers and Fernández [BFG94] already addressed this question by proving, without any assumption on confluence, that when the rewrite rules are only at object level, product compatibility always holds. They proved this theorem for the Algebraic λ-Cube, but, as we will see, it can be adapted for the $\lambda\Pi$-Calculus Modulo. For instance, the following global context Γ satisfies product compatibility because all the rewrite rules are at object level:

$\texttt{nat} : \textbf{Type}$.
$\texttt{minus} : \texttt{nat} \longrightarrow \texttt{nat} \longrightarrow \texttt{nat}$.
$\texttt{minus}\ n\ 0 \hookrightarrow n$.
$\texttt{minus}\ (\texttt{S}\ n_1)\ (\texttt{S}\ n_2) \hookrightarrow \texttt{minus}\ n_1\ n_2$.
$\texttt{minus}\ 0\ n \hookrightarrow 0$.
$\texttt{minus}\ n\ n \hookrightarrow 0$.
$\texttt{minus}\ (\texttt{S}\ n)\ n \hookrightarrow \texttt{S}\ 0$.

However, the relation $\rightarrow_{\beta\Gamma}$ is non-confluent. Indeed, let Ω be the fix-point combinator introduced in Lemma 1.4.8. We have:

$\texttt{minus}\ (\Omega\ \texttt{S})\ (\Omega\ \texttt{S}) \rightarrow_\Gamma 0$ and $\texttt{minus}\ (\Omega\ \texttt{S})\ (\Omega\ \texttt{S}) \rightarrow_\beta^* \texttt{minus}\ (\texttt{S}\ (\Omega\ \texttt{S}))\ (\Omega\ \texttt{S}) \rightarrow_\Gamma \texttt{S}\ 0$.

However 0 and S 0 are not joinable.

But, what about global contexts containing at the same time non left-linear rewrite rules and type-level rewrite rules? Barbanera's result has been extended by Blanqui [Bla05a] to rewrite systems without product types in the right-hand side of rewrite rules. Still, the question remains for global contexts with non left-linear rewrite rules and rewrite rules with product types on the right-hand side.

In general, product compatibility does not hold if the reduction $\rightarrow_{\beta\Gamma}$ is not confluent. Consider the following extension of the global context above:

$A : \textbf{Type}$.
$T : \texttt{nat} \longrightarrow \textbf{Type}$.
$T\ 0 \hookrightarrow \texttt{nat} \longrightarrow A$.
$T\ (S\ 0) \hookrightarrow \texttt{nat} \longrightarrow \texttt{nat}$.

Since, as we have seen above, $0 \equiv_{\beta\Gamma} \texttt{S}\ 0$, we have:

$(\texttt{nat} \longrightarrow A) \leftarrow_\Gamma T\ 0 \equiv_{\beta\Gamma} T\ (S\ 0) \rightarrow_\Gamma (\texttt{nat} \longrightarrow \texttt{nat})$.

But, we do not have $A \equiv_{\beta\Gamma} \texttt{nat}$. Hence, product compatibility does not hold.

The problem here is that the non-confluence occurring at object level on terms of type nat is propagated at type level through the rewrite rules on the symbol T whose behaviour depends on its argument of type nat.

To avoid such a situation, we would like to use a criterion based on typing to *separate* the type-level part of the rewrite system from the non left-linear part.

However, this is not possible because conversions between two well-typed terms may contain ill-typed terms and, therefore, no argument based on typing can be applied. This problem is strongly connected to our choice of using an untyped reduc-

tion and an untyped conversion in the $\lambda\Pi$-Calculus Modulo.

A radical solution would be to consider a typed conversion. We leave this line of research for future work as we believe it deeply modifies the type system (see Section 5.7). Instead, we consider a variant of the $\lambda\Pi$-Calculus Modulo where the conversion respects a weak form of typing and for which we give a general criterion for product compatibility able to deal with global contexts containing at the same time non left-linear rewrite rules and rewrite rules with product types on their right-hand side. This can be thought as a first step toward the study of the subject reduction property in the $\lambda\Pi$-Calculus Modulo with a typed conversion. As we will see in Chapter 6, another interesting consequence of proving product compatibility for this variant is that it can be used to prove the soundness of the inference algorithm with respect to the *unmodified* $\lambda\Pi$-Calculus Modulo.

Convention In this chapter, we only consider rewrite rules of the form $(f\,\vec{u} \hookrightarrow v)$ where f is a constant. We call f the head symbol of the rule. This restriction is needed to prove the *Postponement Lemmas* and the *Commutations Lemma* (see below) and to define some of the notions introduced hereafter.

5.2 Object-Level Rewrite Systems

To begin with, we prove, by adapting the work of Barbanera, Geuvers and Fernández [BFG94] and Blanqui [Bla05a] to the $\lambda\Pi$-Calculus Modulo, that product compatibility always holds for global context without Π-*producing* rewrite rules.

Definition 5.2.1 (Π-Producing Rewrite Rules)**.** *A rewrite rule* $(u \hookrightarrow v)$ *is* Π*-producing if:*

- *there is a product type in* v*;*
- *this product type is not in the type annotation of an abstraction.*

Remark 5.2.2. *Object-level rewrite rules are not* Π*-producing.*

Remark 5.2.3. *Right-hand sides of* Π*-producing rewrite rules are of the form* $(\lambda\vec{x} : \vec{T}.\Pi y : A.B)\vec{t}$ *where* $\vec{x}$ *and* $\vec{t}$ *may be empty vectors.*

The main results of this section are the following:

Theorem 5.2.4. *Global contexts without any* Π*-producing rewrite rules satisfy the product compatibility property.*

Corollary 5.2.5. *Global contexts containing only object-level rewrite rules satisfy the product compatibility property.*

Proof. By Theorem 5.2.4 and Remark 5.2.2. □

The proof of Theorem 5.2.4 relies on two lemmas: a *postponement* lemma and a *commutation* lemma.

The *postponement* lemma says that Γ-reductions can be postponed after the β-steps occurring in head position (β^h-reduction) when we reduce toward a product type.

Lemma 5.2.6 (Postponement)**.** *Let* Γ *be a global context without any* Π*-producing rewrite rules.*

If $T_1 \to_\Gamma^* T_2 \to_{\beta^h}^* \Pi x : A_2.B_2$*, then there exist* A_1 *and* B_1 *such that* $T_1 \to_{\beta^h}^* \Pi x : A_1.B_1 \to_\Gamma^* \Pi x : A_2.B_2$.

Proof. Since there are no Π-producing rules, it is sufficient to show that if $T_1 \to_\Gamma^* T_2 \to_{\beta^h}^* P$ and P contains a product type (but not inside the type annotation of an abstraction), then, for some T_3, we have $T_1 \to_{\beta^h}^* T_3 \to_\Gamma^* P$.

We proceed by induction on the number of β^h steps. The base case is trivial.

- **(Inductive Step)** Suppose that $T_1 \to_\Gamma^* T_2 \to_{\beta^h}^k P_0 \to_{\beta^h} P$ and P contains a product type. Then $P_0 = (\lambda x : U_2.V_2)u_2$, $P = V_2[x/u_2]$ and V_2 contains a product type. By induction hypothesis, we have $T_1 \to_{\beta^h}^* T_0 \to_\Gamma^* (\lambda x : U_2.V_2)u_2 \to_{\beta^h} P$. Since rules are non-$\Pi$-producing, we have $T_0 = (\lambda x : U_1.V_1)u_1$ with $V_1 \to_\Gamma^* V_2$ and $u_1 \to_\Gamma^* u_2$. Thus, we have $T_1 \to_{\beta^h}^* T_0 = (\lambda x : U_1.V_1)u_1 \to_{\beta^h} V_1[x/u_1] \to_\Gamma^* V_2[x/u_2] = P$.

□

The *commutation* lemma says that $\to_\Gamma$ and $\to_{\beta^h}$ commute.

Lemma 5.2.7 (Commutation)**.** *Let* Γ *be a global context.*

If $t_1 \to_{\beta^h}^* t_2$ *and* $t_1 \to_\Gamma^* t_3$*, then, for some* t_4*, we have* $t_2 \to_\Gamma^* t_4$ *and* $t_3 \to_{\beta^h}^* t_4$.

Proof. We proceed by induction on the number of β^h-steps.

Suppose that $t_1 \to_{\beta^h}^* u_1 \to_{\beta^h} t_2$. By induction hypothesis, there exists u_2 such that $t_3 \to_{\beta^h}^* u_2$ and $u_1 \to_\Gamma^* u_2$.

Since u_1 is β^h-reducible, $u_1 = (\lambda x : A_1.f_1)a_1\vec{w}_1$ and $u_2 = (\lambda x : A_2.f_2)a_2\vec{w}_2$ with $A_1 \to_\Gamma^* A_2$, $f_1 \to_\Gamma^* f_2$, $a_1 \to_\Gamma^* a_2$ and $\vec{w}_1 \to_\Gamma^* \vec{w}_2$.

Moreover, we have $t_2 = (f_1[x/a_1])\vec{w}_1$ and we can take $t_4 = (f_2[x/a_2])\vec{w}_2$. □

Using these two lemmas, we can prove Theorem 5.2.4.

Proof of Theorem 5.2.4. Suppose that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$.

Since $(\equiv_{\beta\Gamma}) = (\downarrow_\Gamma \cup \downarrow_\beta)^*$, we also have $\Pi x : A_1.B_1\ (\downarrow_\Gamma \cup \downarrow_\beta)^*\ \Pi x : A_2.B_2$.

We prove, by induction on n, that, if $\Pi x : A_1.B_1\ (\downarrow_\Gamma \cup \downarrow_\beta)^n\ T$, then $T \to_{\beta^h}^* \Pi x : A_2.B_2$ with $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$.

It suffices to prove, for $\downarrow = \downarrow_\beta$ and $\downarrow = \downarrow_\Gamma$, that, if $T_1 \downarrow T_2$ and $T_1 \to_{\beta^h}^* \Pi x : A_1.B_1$, then, for some A_2 and B_2, $T_2 \to_{\beta^h}^* \Pi x : A_2.B_2$ with $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$.

If $\downarrow = \downarrow_\beta$ (see Figure 5.1), then it follows from

1. confluence of $\to_\beta$ (Theorem 1.3.5) and
2. standardization (Theorem 1.3.6).

If $\downarrow = \downarrow_\Gamma$ (see Figure 5.1), then it follows from

1. the commutation lemma (Lemma 5.2.7) and
2. the postponement lemma (Lemma 5.2.6).

□

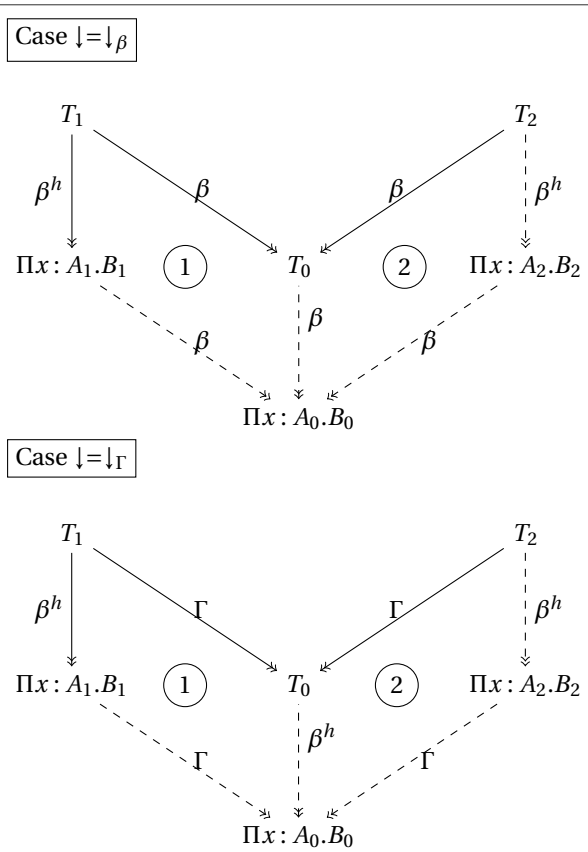


Figure 5.1: Proof of product compatibility

5.3 Towards a New Criterion For Product Compatibility

As already mentioned, our goal is to design a new criterion for product compatibility able to deal with global contexts containing Π-producing rewrite rules *and* non left-linear rewrite rules.

Suppose that we want to define polymorphic pairs. Following the encoding of polymorphism of Section 2.7.2, we can proceed as follows[2]:

$\mathtt{U}_T$: **Type**.
$\dot{\epsilon}$: $\mathtt{U}_T \longrightarrow$ **Type**.
$\dot{\Pi}$: $\Pi a : \mathtt{U}_T . \Pi x : (\dot{\epsilon}\ a \longrightarrow \mathtt{U}_T) . \mathtt{U}_T$.
$\dot{\epsilon}\ (\dot{\Pi}\ a\ b) \hookrightarrow \Pi x : \dot{\epsilon}\ a . \dot{\epsilon}\ (b\ x)$.

$\mathtt{Pair} : \mathtt{U}_T \longrightarrow \mathtt{U}_T \longrightarrow$ **Type**.
π_1 : $\Pi a : \mathtt{U}_T . \Pi b : \mathtt{U}_T . \mathtt{Pair}\ a\ b \longrightarrow \dot{\epsilon}\ a$.
π_2 : $\Pi a : \mathtt{U}_T . \Pi b : \mathtt{U}_T . \mathtt{Pair}\ a\ b \longrightarrow \dot{\epsilon}\ b$.
$\mathtt{mk\_pair} : \Pi . a : \mathtt{U}_T . \Pi . b : \mathtt{U}_T . \dot{\epsilon}\ a \longrightarrow \dot{\epsilon}\ b \longrightarrow \mathtt{Pair}\ a\ b$.
$\pi_1\ a_1\ b_1\ (\mathtt{mk\_pair}\ a_2\ b_2\ x\ y) \hookrightarrow x$.
$\pi_2\ a_2\ b_2\ (\mathtt{mk\_pair}\ a_2\ b_2\ x\ y) \hookrightarrow y$.

If we want the pairs to be surjective, we need to add this non left-linear rewrite rule:

$\mathtt{mk\_pair}\ a\ b\ (\pi_1\ a\ b\ p)\ (\pi_2\ a\ b\ p) \hookrightarrow p$.

We get a global context mixing Π-producing rewrite rules and non left-linear rewrite rules. However, the types involved in the Π-producing rewrite rule ($\mathtt{U}_T$ and **Type**) are different from the type involved in the non left-linear rewrite rule (`Pair`). Therefore, these rewrite rules should not interact. We will show that, in this case, product compatibility holds.

We will proceed as follows.

- First, in Section 5.4, we define a weak notion of typing, for which subject reduction is easy to prove.
- Then, in Section 5.5, we define a variant of the $\lambda\Pi$-Calculus Modulo, in which the conversion rule is assumed to be weakly well-typed.
- Finally, in Section 5.6, we prove a general criterion for product compatibility for this variant of the $\lambda\Pi$-Calculus Modulo. We introduce two rewriting relation $\rightarrow_{out}$ and $\rightarrow_{in}$ for which we prove a *postponement* lemma and a *commutation* lemma. This allows us to prove a theorem similar to Theorem 5.2.4, but allowing, under some assumptions, Π-producing rewrite rules.

5.4 Weak Typing

We now define formally the notions of *weak types* and *weak typing*. Recall that the goal is to have an alternative notion of typing for which subject reduction is easy to prove. We obtain the new notion of typing by dropping dependent types for simple types.

$$A, B \quad ::= \quad \textbf{Kind} \mid \textbf{Type} \mid \textbf{Black} \mid \textbf{White} \mid A \to B$$

Figure 5.2: Syntax for simple types

5.4.1 Weak Types

Definition 5.4.1 (Weak Types)**.** *Weak types are simple types built from the constants* **Kind**, **Type**, **Black** *and* **White**. *The concrete syntax for weak types is given Figure 5.2.*

Assuming that we have a function *Color*(.) giving a color (black or white) to any type constant, we can associate to any type or kind of the $\lambda\Pi$-Calculus Modulo a weak type. We simply ignore type dependencies.

Definition 5.4.2 (Stripping Function)**.** *The stripping function* $\|.\|$*, from types and kinds to weak types, is defined as follows:*

$$\begin{array}{lcl}
\|\textbf{Kind}\| & = & \textbf{Kind} \\
\|\textbf{Type}\| & = & \textbf{Type} \\
\|C\| & = & \text{Color}(C) \\
\|At\| & = & \|A\| \\
\|\lambda x : A.B\| & = & \|B\| \\
\|\Pi x : A.B\| & = & \|A\| \to \|B\|
\end{array}$$

Remark that the type constants in the $\lambda\Pi$-Calculus Modulo are collapsed, when translated into weak types, into only two constants: **Black** and **White**. The reason for this collapse (we could have kept all the type constants in weak types) is that it maximizes our chances to subsequently use weak subject reduction (Theorem 5.4.25). On the other side, we need at least two constants because, in Section 5.6, we will use weak typing to discriminate between two kinds of terms: roughly speaking, terms allowed to *be non confluent* and terms that are not.

The stripping function is invariant by substitutions since substitutions only concern objects.

Lemma 5.4.3. $\|\sigma(A)\| = \|A\|$

Proof. By induction on A. Remark that objects are ignored in the definition of $\|.\|$. □

We now define for weak types an equivalent of the relation $\to_{\beta\Gamma}$ for terms.

Definition 5.4.4. *Let* Γ *be a global context. The relation* $\to_\Gamma^w$ *on weak types is the smallest relation closed by subterm rewriting such that* $\|U\| \to_\Gamma^w \|V\|$ *for any type-level rewrite rule* $(U \hookrightarrow V) \in \Gamma$.

We write $\equiv_\Gamma^w$ *for the congruence generated by* $\to_\Gamma^w$.

The stripping function commutes with the reduction.

Lemma 5.4.5. *If* $A \to_{\beta\Gamma} B$*, then either* $\|A\| = \|B\|$ *or* $\|A\| \to_\Gamma^w \|B\|$.

Proof. By induction on $\to_{\beta\Gamma}$, using Lemma 5.4.3. □

As a corollary, we get that the stripping function commutes with the congruence.

[2] Remark the last two rewrite rules have been linearized as explained in Chapter 3.

Lemma 5.4.6. *If $A \equiv_{\beta\Gamma} B$, then $\|A\| \equiv^{w}_{\Gamma} \|B\|$.*

Proof. By induction on $\equiv_{\beta\Gamma}$, using Lemma 5.4.5. □

The color of weak types is defined as follows:

Definition 5.4.7 (Black and White Weak Types)**.** *The color of a weak type (***Black** *or* **White***) is defined as follows:*

Color(**Kind**)	=	**Black**
Color(**Type**)	=	**Black**
Color(**Black**)	=	**Black**
Color(**White**)	=	**White**
Color($A \to B$)	=	**Color**(B)

A type-level rewrite rule that preserves the color of weak types is called non-confusing.

Definition 5.4.8 (Non-Confusing Rewrite Rules)**.** *A type-level rewrite rule* $(U \hookrightarrow V)$ *is non-confusing if* **Color**($\|U\|$) = **Color**($\|V\|$).

Lemma 5.4.9. *Let Γ be a global context whose type-level rewrite rules are non-confusing. If A and B are weak types such that $A \equiv^{w}_{\Gamma} B$, then* **Color**(A) = **Color**(B).

Proof. By induction on the definition of $\equiv^{w}_{\Gamma}$ and on A. □

5.4.2 Weak Typing

We now show how to associate weak types to the terms of the $\lambda\Pi$-Calculus Modulo using a typing discipline close to the simply typed λ-calculus.

Definition 5.4.10 (Weak Typing)**.** *Let Γ be a global context. A term t has* weak type *T in the local context Δ if the judgment $\Gamma;\Delta \vdash_{w} t : T$ is derivable from the inference rules of Figure 5.3.*

A term t is weakly well-typed if such a T exists.

The notion of weak typing is an approximation of *usual* typing in the following sense:

Lemma 5.4.11. *If $\Gamma;\Delta \vdash t : T$, then $\Gamma;\Delta \vdash_{w} t : \|T\|$.*

Proof. By induction on the typing derivation.

- **(Type, Variable, Constant)** Trivial.
- **(Application)** By induction hypothesis and Lemma 5.4.3.
- **(Abstraction)** By induction hypothesis.
- **(Product)** By induction hypothesis.
- **(Conversion)** By induction hypothesis and Lemma 5.4.6.

□

The converse is not true. For instance `plus 0 true` is obviously ill-typed; however it is weakly well-typed if we take *Color*(`nat`) = *Color*(`prop`).

(Sort) $$\frac{}{\Gamma;\Delta \vdash_w \mathbf{Type} : \mathbf{Kind}}$$

(Variable) $$\frac{(x : A) \in \Delta}{\Gamma;\Delta \vdash_w x : \|A\|}$$

(Constant) $$\frac{(c : A) \in \Gamma}{\Gamma;\Delta \vdash_w c : \|A\|}$$

(Application)

$$\frac{\Gamma;\Delta \vdash_w t : A \to B \qquad \Gamma;\Delta \vdash_w u : A}{\Gamma;\Delta \vdash_w t\,u : B}$$

(Abstraction)

$$\frac{\Gamma;\Delta \vdash_w A : \mathbf{Type} \qquad \Gamma;\Delta(x : A) \vdash_w t : B \qquad B \neq \mathbf{Kind}}{\Gamma;\Delta \vdash_w \lambda x : A.t : \|A\| \to B}$$

(Product)

$$\frac{\Gamma;\Delta \vdash_w A : \mathbf{Type} \qquad \Gamma;\Delta(x : A) \vdash_w B : s}{\Gamma;\Delta \vdash_w \Pi x : A.B : s}$$

(Conversion)

$$\frac{\Gamma;\Delta \vdash_w t : A \qquad \Gamma;\Delta \vdash_w B : s \qquad A \equiv^w_\Gamma \|B\|}{\Gamma;\Delta \vdash_w t : \|B\|}$$

Figure 5.3: Weak typing rules for terms

(Empty Local Context) $$\frac{}{\Gamma \vdash^{ctx}_w \emptyset}$$

(Variable Declaration) $$\frac{\Gamma \vdash^{ctx}_w \Delta \qquad \Gamma;\Delta \vdash_w U : \mathbf{Type} \qquad x \notin dom(\Delta)}{\Gamma \vdash^{ctx}_w \Delta(x : U)}$$

Figure 5.4: Weak well-formedness rules for local contexts

Definition 5.4.12 (Weakly Well-Formed Local Contexts)**.** *A local context* Δ *is* weakly well-formed *with respect to a global context* Γ *if the judgment* $\Gamma \vdash^{ctx}_{w} \Delta$ *is derivable by the inference rules of Figure 5.4.*

Lemma 5.4.13. *If* $\Gamma \vdash^{ctx} \Delta$*, then* $\Gamma \vdash^{ctx}_{w} \Delta$*.*

Proof. By induction on the derivation, using Lemma 5.4.11. □

Definition 5.4.14 (Weakly Well-Typed Rewrite Rules)**.** *Let* Γ *be a global context.*

A rewrite rule $(u \hookrightarrow v)$ *is* weakly well-typed[1] *in* Γ *if, for any substitution* σ*, weak type* T *and weakly well-formed local context* Δ*, if* $\Gamma;\Delta \vdash_w \sigma(u) : T$*, then* $\Gamma;\Delta \vdash_w \sigma(v) : T$*.*

Remark 5.4.15. *Well-typed rewrite rules are not necessarily weakly well-typed. For instance the rule* $(\texttt{plus}\ 0\ \texttt{true} \hookrightarrow \lambda x : \texttt{nat}.x)$ *is well-typed because it is not possible to use it on a well-typed term. However, if we take* $\mathrm{Color}(\texttt{nat}) = \mathrm{Color}(\texttt{prop}) =$ **Black***, then the rule is not weakly well-typed because the weak-types of its left-hand side (***Black***) and right-hand side (***Black** $\longrightarrow$ **Black***) are different.*

In Section 5.4.3, we give a simple criterion for showing that a rewrite rule is weakly well-typed.

To prove weak subject reduction, we need a weak notion of product compatibility.

Definition 5.4.16 (Weak Product Compatibility)**.** *A global context* Γ *satisfies the* weak product compatibility *if, for any* $\Delta, A_1, A_2, B_1, B_2$ *such that:* $\Pi x : A_1.B_1 \equiv^{w}_{\Gamma} \Pi x : A_2.B_2$*, we have* $A_1 \equiv^{w}_{\Gamma} A_2$ *and* $B_1 \equiv^{w}_{\Gamma} B_2$*.*

Lemma 5.4.17. *Let* Γ *be a global context. If* $\rightarrow^{w}_{\Gamma}$ *is confluent, then weak product compatibility holds for* Γ*.*

Proof. If $\Pi x : A_1.B_1 \equiv^{w}_{\Gamma} \Pi x : A_2.B_2$, then, for some A_0 and B_0, we have $A_1 \rightarrow^{w}_{\Gamma} A_0$, $A_2 \rightarrow^{w}_{\Gamma} A_0$, $B_1 \rightarrow^{w}_{\Gamma} B_0$ and $B_2 \rightarrow^{w}_{\Gamma} B_0$. □

Remark 5.4.18. *It is much easier to prove weak product compatibility than to prove product compatibility. As we have seen, it follows from the confluence of the rewrite system obtained by* stripping *the type-level rewrite rules. It means that there is no need to consider object-level rewrite rules nor* β*-reduction. Therefore, the difficulty associated with non left-linear rewrite rules vanishes. Moreover, such a system is ground (no variable in the rewrite rules); therefore its confluence is decidable [CGN01] in polynomial time.*

Definition 5.4.19 (Weakly Well-Typed Global Context)**.** *A global context* Γ *is* weakly well-typed *if*

- *for all* $(c : A) \in \Gamma, \Gamma;\emptyset \vdash_w A : s$*;*
- *all rewrite rules in* Γ *are weakly-well typed;*
- *all type-level rewrite rules are non-confusing;*
- Γ *satisfies the weak product compatibility property.*

[1] Not to be confused with weakly well-formed rewrite rules (Definition 3.6.5). We never use the latter notion in this chapter.

5.4.3 Properties

Many typing properties of the $\lambda\Pi$-Calculus Modulo have a *weak* counterpart.

We begin by proving some simple lemmas.

Lemma 5.4.20 (Inversion)**.** *If* $\Gamma;\Delta \vdash_w t : T$ *then*

- *either* $t = \mathbf{Type}$ *and* $T = \mathbf{Kind}$*;*
- *or* $t = x$ *and there exists* A *such that* $(x : A) \in \Delta$ *and* $T \equiv_\Gamma^w \|A\|$*;*
- *or* $t = c$ *and there exists* A *such that* $(c : A) \in \Gamma$ *and* $T \equiv_\Gamma^w \|A\|$*;*
- *or* $t = f\,u$ *and there exist* A *and* B *such that* $\Gamma;\Delta \vdash_w f : A \longrightarrow B$, $\Gamma;\Delta \vdash_w u : A$ *and* $T \equiv_\Gamma^w B$*;*
- *or* $t = \lambda x : A.t$ *and there exists* $B \neq \mathbf{Kind}$ *such that* $\Gamma;\Delta \vdash_w A : \mathbf{Type}$, $\Gamma;\Delta(x : A) \vdash_w t : B$, *and* $T \equiv_\Gamma^w \|A\| \longrightarrow B$*;*
- *or* $t = \Pi x : A.B$ *and there exists a sort* s *such that* $\Gamma;\Delta \vdash_w A : \mathbf{Type}$, $\Gamma;\Delta(x : A) \vdash_w B : s$ *and* $T = s$.

Proof. By induction on the typing derivation. □

Lemma 5.4.21. *If* $\Delta_1 \to_{\beta\Gamma} \Delta_2$ *and* $\Gamma;\Delta_1 \vdash_w t : A$, *then* $\Gamma;\Delta_2 \vdash_w t : A$.

Proof. By induction on the typing derivation and Lemma 5.4.5. □

Lemma 5.4.22. *If* $\Gamma;\Delta(x : A) \vdash_w u : T$ *and* $\Gamma;\Delta \vdash_w v : \|A\|$, *then* $\Gamma;\Delta \vdash_w u[x/v] : T$.

Proof. By induction on the typing derivation. □

We now prove a weak subject reduction theorem, *i.e.*, that reduction preserves weak typing.

Lemma 5.4.23 (Weak Subject Reduction for $\to_\Gamma$)**.** *Let* Γ *be a global context whose rewrite rules are weakly well-typed.*

If $\Gamma;\Delta \vdash_w t_1 : T$ *and* $t_1 \to_\Gamma t_2$, *then* $\Gamma;\Delta \vdash_w t_2 : T$.

Proof. We proceed by induction on t_1 and follow the proof of usual subject reduction for $\to_\Gamma$ (Lemma 2.6.21). We use Lemma 5.4.20, Lemma 5.4.22 and Lemma 5.4.21. □

Lemma 5.4.24 (Weak Subject Reduction for $\to_\beta$)**.** *Let* Γ *be a global context that satisfies the weak product compatibility property.*

If $\Gamma;\Delta \vdash_w t_1 : T$ *and* $t_1 \to_\beta t_2$, *then* $\Gamma;\Delta \vdash_w t_2 : T$.

Proof. We proceed by induction on t_1 and follow the proof of usual subject reduction for $\to_\beta$ (Lemma 2.6.20). As previously, Lemma 5.4.22 and Lemma 5.4.21 are needed. We detail the redex case.

Suppose that $t_1 = (\lambda x : A_0.u)\,v$ and $t_2 = u[x/v]$. By inversion, on the one hand, $\Gamma;\Delta \vdash_w \lambda x : A_0.u : A \longrightarrow B$, $\Gamma;\Delta \vdash_w v : A$ and $T \equiv_\Gamma^w B$ and, on the other hand, $\Gamma;\Delta \vdash_w A_0 : \mathbf{Type}$, $\Gamma;\Delta(x : A_0) \vdash_w u : B_0$ and $A_0 \longrightarrow B_0 \equiv_\Gamma^w A \longrightarrow B$.

By weak product compatibility, $A_0 \equiv_\Gamma^w A$ and $B_0 \equiv_\Gamma^w B$. By Lemma 5.4.22, from $\Gamma;\Delta(x : A_0) \vdash_w u : T$ and $\Gamma;\Delta \vdash_w v : \|A_0\|$, we deduce $\Gamma;\Delta \vdash_w u[x/v] : T$. □

From these two lemmas, we get weak subject reduction.

Theorem 5.4.25 (Weak Subject Reduction)**.** *Let* Γ *be a global context that satisfies the weak product compatibility property and whose rewrite rules are weakly well-typed.*

If $\Gamma;\Delta \vdash_w t_1 : T$ *and* $t_1 \to_{\beta\Gamma} t_2$*, then* $\Gamma;\Delta \vdash_w t_2 : T$*.*

Proof. Follows from Lemma 5.4.24 and Lemma 5.4.23. □

From weak product compatibility, we can also prove the uniqueness of weak types.

Theorem 5.4.26 (Uniqueness of Weak Types)**.** *Let* Γ *be a global context satisfying the weak product compatibility property and let* Δ *be a weakly well-formed local context.*

If $\Gamma;\Delta \vdash_w t : T_1$ *and* $\Gamma;\Delta \vdash_w t : T_2$*, then* $T_1 \equiv^w_\Gamma T_2$*.*

Proof. By induction on the first typing derivation.

- **(Sort)**, **(Variable)**, **(Constant)** By inversion on the second typing derivation.
- **(Application)** Suppose that $t = u\,v$, $\Gamma;\Delta \vdash_w u : A_1 \longrightarrow T_1$, $\Gamma;\Delta \vdash_w v : A_1$.

  By inversion on the second typing derivation, $\Gamma;\Delta \vdash_w u : A_2 \longrightarrow B_2$, $\Gamma;\Delta \vdash_w v : A_2$ and $T_2 \equiv^w_\Gamma B_2$.

  By induction hypothesis, we have $A_1 \longrightarrow T_1 \equiv^w_\Gamma A_2 \longrightarrow B_2$.

  Finally, by weak product compatibility, $T_1 \equiv^w_\Gamma B_2 \equiv^w_\Gamma T_2$.
- **(Abstraction)** and **(Product)** By inversion on the second typing derivation and induction hypothesis.
- **(Conversion)** By induction hypothesis.

□

In the following, we focus on weakly well-typed rewrite rules and we give a simple criterion for showing that a rewrite rule is weakly well-typed.

Theorem 5.4.27. *Let* Γ *be a global context satisfying the weak product compatibility property. Suppose that:*

- $f\,\vec{u}$ *is algebraic;*
- $\Gamma \vdash^{ctx}_w \Delta$*;*
- $dom(\Delta) = FV(f\,\vec{u})$*;*
- $\Gamma;\Delta \vdash_w f\,\vec{u} : T$*;*
- $\Gamma;\Delta \vdash_w v : T$*.*

Then, $(f\,\vec{u} \hookrightarrow v)$ *is weakly well-typed in* Γ*.*

This theorem is a *weak* version of Theorem 3.3.3. The proofs are similar.

First, we need to introduce the notion of weakly well-typed substitution.

Definition 5.4.28 (Weakly Well-Typed Substitutions)**.** *A substitution* σ *is* weakly well-typed *from* Δ_1 *to* Δ_2 *in* Γ*, written* $\sigma : \Delta_1 \rightsquigarrow^w_\Gamma \Delta_2$*, if, for all* $x \in dom(\Delta_1)$*, we have* $\Gamma;\Delta_2 \vdash_w \sigma(x) : \|\Delta_1(x)\|$*.*

Lemma 5.4.29. *Let Γ be a global context whose declarations are closed. If $\sigma : \Delta_1 \rightsquigarrow^w_\Gamma \Delta_2$ and $\Gamma; \Delta_1 \vdash_w t : T$, then $\Gamma; \Delta_2 \vdash_w \sigma(t) : T$.*

Proof. By induction on the typing derivation. □

Then, we prove a *main lemma*.

Lemma 5.4.30. *Let Γ be a global context satisfying the weak product compatibility property. Assume that:*

- *t is algebraic;*
- $\Gamma; \Delta_0 \vdash_w t : T_0$,
- $\Gamma \vdash^{ctx}_w \Delta$,
- $\Gamma; \Delta \vdash_w \sigma(t) : T$,

We have

- $T \equiv^w_\Gamma T_0$,
- *and, for all $x \in FV(t), \Gamma; \Delta \vdash_w \sigma(x) : \|\Delta_0(x)\|$.*

Proof. We proceed by induction on t.

- Suppose that $t = f$ is a constant. By inversion (Lemma 5.4.20), we have $T_0 \equiv^w_\Gamma \|\Gamma(f)\| \equiv^w_\Gamma T$.
- Suppose that $t = uv$ with u and v algebraic terms. By inversion (Lemma 5.4.20), on the one hand, $\Gamma; \Delta_0 \vdash_w u : A_0 \longrightarrow B_0$, $\Gamma; \Delta_0 \vdash_w v : A_0$ and $T_0 \equiv^w_\Gamma B_0$.

  On the other hand, $\Gamma; \Delta \vdash_w \sigma(u) : A \longrightarrow B$, $\Gamma; \Delta \vdash_w \sigma(v) : A$, and $T \equiv^w_\Gamma B$.

  By induction hypothesis, $A \longrightarrow B \equiv^w_\Gamma A_0 \longrightarrow B_0$, $A \equiv^w_\Gamma A_0$ and, for all $x \in FV(t), \Gamma; \Delta \vdash_w \sigma(x) : \|\Delta_0(x)\|$.

  By weak product compatibility, $B \equiv^w_\Gamma B_0$. It follows that $T \equiv^w_\Gamma T_0$.
- Suppose that $t = ux$ with u algebraic and x a variable in Δ_0. By inversion (Lemma 5.4.20), on the one hand, $\Gamma; \Delta_0 \vdash_w u : A_0 \longrightarrow B_0$, $\Gamma; \Delta_0 \vdash_w x : A_0$ with $A_0 \equiv^w_\Gamma \Delta_0(x)$ and $T_0 \equiv^w_\Gamma B_0$.

  On the other hand, $\Gamma; \Delta \vdash_w \sigma(u) : A \longrightarrow B$, $\Gamma; \Delta \vdash_w \sigma(x) : A$ and $T \equiv^w_\Gamma B$.

  By induction hypothesis, $A \longrightarrow B \equiv^w_\Gamma A_0 \longrightarrow B_0$ and, for all $x \in FV(u), \Gamma; \Delta \vdash_w \sigma(x) : \|\Delta_0(x)\|$.

  By weak product compatibility, $A \equiv^w_\Gamma A_0$ and $B \equiv^w_\Gamma B_0$. It follows that $T \equiv^w_\Gamma B \equiv^w_\Gamma B_0 \equiv^w_\Gamma T_0$.

  Moreover, we have $\Gamma; \Delta \vdash_w \sigma(x) : A$ with $A \equiv^w_\Gamma A_0 \equiv^w_\Gamma \|\Delta_0(x)\|$.

□

Finally, we can prove the theorem.

Proof of Theorem 5.4.27. Suppose that we have $\Gamma; \Delta \vdash_w \sigma(f\vec{u}) : T$, $\Gamma; \Delta_0 \vdash_w f\vec{u} : T_0$ and $\Gamma; \Delta_0 \vdash_w v : T_0$.

By Lemma 5.4.30, we have $T \equiv^w_\Gamma T_0$ and, for all $x \in FV(t), \Gamma; \Delta \vdash_w \sigma(x) : \|\Delta_0(x)\|$.

By induction on $\Gamma \vdash^{ctx}_w \Delta_0$, we can conclude that $\sigma : \Delta_0 \rightsquigarrow^w_\Gamma \Delta$.

By Lemma 5.4.29, we have $\Gamma; \Delta \vdash_w \sigma(v) : T_0$ and, by conversion, $\Gamma; \Delta \vdash_w \sigma(v) : T$. □

(**Restricted Conversion**) $$\frac{\Gamma;\Delta\vdash' t:A \qquad \Gamma;\Delta\vdash' B:s \qquad \Gamma;\Delta\vdash^w A\equiv B}{\Gamma;\Delta\vdash' t:B}$$

Figure 5.5: Restricted conversion rule

In fact, the other criteria of Chapter 3 could also be adapted to prove the weak well-typedness of rewrite rules.

5.5 The Colored $\lambda\Pi$-Calculus Modulo

In this section, we define a variant of the $\lambda\Pi$-Calculus Modulo, where the conversion is assumed to be weakly well-typed.

5.5.1 Weakly Well-Typed Conversion

Definition 5.5.1 (Weakly Well-Typed Conversion)**.** *Let Γ be a global context and Δ be a local context. We write $\Gamma;\Delta\vdash^w t_1\equiv t_2$ if the pair (t_1,t_2) is in the reflexive, symmetric and transitive closure of the relation $\{\,(t_1,t_2)\mid \exists T.\Gamma;\Delta\vdash_w t_1:T$ and $t_1\to_{\beta\Gamma} t_2\,\}$.*

Remark 5.5.2. *If $\Gamma;\Delta\vdash^w t_1\equiv t_2$, then $t_1\equiv_{\beta\Gamma} t_2$.*

The converse is not true in general. However, it holds in the particular case where the rewriting relation is confluent.

Lemma 5.5.3. *Let Γ be a global context satisfying the weak product compatibility property, whose rewrite rules are weakly well-typed and such that $\to_{\beta\Gamma}$ is confluent.*

If $t_1\equiv_{\beta\Gamma} t_2$ and t_1 and t_2 are weakly well-typed in Δ, then $\Gamma;\Delta\vdash^w t_1\equiv t_2$.

Proof. By confluence, we have $t_1\downarrow_{\beta\Gamma} t_2$. By weak subject reduction (Theorem 5.4.25), all the reducts of t_1 and t_2 are weakly well-typed. □

The weakly well-typed conversion ensures that every term in the conversion has the same weak type.

Lemma 5.5.4. *Let Γ be a global context satisfying the weak product compatibility property and whose rewrite rules are weakly well-typed.*

If $\Gamma;\Delta\vdash_w t_1:T$ and $\Gamma;\Delta\vdash^w t_1\equiv t_2$, then, for every term t in the $\beta\Gamma$-path between t_1 and t_2, we have $\Gamma;\Delta\vdash_w t:T$.

Proof. By weak subject reduction (Theorem 5.4.25). □

5.5.2 The Colored $\lambda\Pi$-Calculus Modulo

Definition 5.5.5 (The Colored $\lambda\Pi$-Calculus Modulo)**.** *The relation $\vdash'$ is defined by the inference rules of Figure 2.4 (the typing rules for usual typing) where we replace the* **(Conversion)** *rule by the* **(Restricted Conversion)** *rule of Figure 5.5.*

The relation $\vdash'$ is contained in $\vdash$.

Lemma 5.5.6. *If $\Gamma;\Delta\vdash' t:T$, then $\Gamma;\Delta\vdash t:T$.*

Proof. Follows from Remark 5.5.2. □

Moreover, if the relation $\rightarrow_{\beta\Gamma}$ is confluent, then the typing relations $\vdash'$ and $\vdash$ are equal.

Lemma 5.5.7. *Let Γ be a global context satisfying the weak product compatibility property, whose rewrite rules are weakly well-typed and such that $\rightarrow_{\beta\Gamma}$ is confluent.*
We have $\Gamma;\Delta \vdash t : T$ if and only if $\Gamma;\Delta \vdash' t : T$.

Proof. By Lemma 5.5.6 and Lemma 5.5.3. □

5.6 A General Criterion for Product Compatibility for the Colored $\lambda\Pi$-Calculus Modulo

In this section we prove a general criterion for product compatibility (Theorem 5.6.5) in the Colored $\lambda\Pi$-Calculus Modulo defined in Section 5.5. We begin by some definitions.

The color of a rewrite rule is the color of its head symbol.

Definition 5.6.1 (Black and White Rewite Rules)**.** *Let Γ be a global context. The color of a rewrite rule $(f\,\vec{u} \hookrightarrow v)$ is the color of $\|\Gamma(f)\|$.*

Remark 5.6.2. *The distinction between black and white rewrite rules formalizes the distinction* safe *and* unsafe *rewrite rules that we introduced in Section 5.1.*

Notation 5.6.3. *We write $t_1 \rightarrow_{Black} t_2$ if $t_1 \rightarrow_\Gamma t_2$ using a black rewrite rule.*

Definition 5.6.4 (Black and White Positions)**.** *Let Γ be a global context. A position p in a term t is black (respectively white) if:*

- *either $t_{|p} = f\,\vec{u}$ and* **Color**$(\|\Gamma(f)\|) =$ **Black** *(respectively* **White***);*
- *or $p = q.i$, $t_{|q} = f\,\vec{u}$, $\|\Gamma(f)\| = A_1 \longrightarrow \ldots \longrightarrow A_n \longrightarrow B$ and* **Color**$(A_i) =$ **Black** *(respectively* **White***).*

Theorem 5.6.5 (Product Compatibility)**.** *Let Γ be a global context. Suppose that:*

- **(A1)** *Γ is weakly well-typed;*
- **(A2)** *black rewrite rules are left-linear;*
- **(A3)** *the relation generated by the black rewrite rules together with β-reduction is confluent;*
- **(A4)** *for any black rewrite rule $(u \hookrightarrow v) \in \Gamma$, all the non-variable subterms of u are at black positions;*

then, Γ satisfies the product compatibility in the Colored $\lambda\Pi$-Calculus Modulo.

5.6.1 The Rewriting Relations $\to_{in}$ and $\to_{out}$

To prove Theorem 5.6.5, we will mimic the proof of Theorem 5.2.4. However, the roles of $\to_{\beta^h}$ and $\to_\Gamma$ will be played by two new relations, $\to_{in}$ and $\to_{out}$ that we define in this section.

First, we need to introduce the notion of black and white terms.

Definition 5.6.6 (Black and White Terms)**.** *Let Γ be a global context. We say that a term t is* black *(respectively* white*) in a local context Δ if, for some T, we have $\Gamma;\Delta \vdash_w t : T$ with* $\mathbf{Color}(\|T\|) = \mathbf{Black}$ *(respectively* $\mathbf{Color}(\|T\|) = \mathbf{White}$*).*

The sets of black and white terms are disjoint. They form a partition of the weakly well-typed terms.

Lemma 5.6.7. *Let Γ be a weakly well-typed global context and Δ be a weakly well-formed local context for Γ.*

If we have $\Gamma;\Delta \vdash_w t : T_1$ and $\Gamma;\Delta \vdash_w t : T_2$, then $\mathbf{Color}(\|T_1\|) = \mathbf{Color}(\|T_2\|)$.

Proof. By uniqueness of weak types (Theorem 5.4.26) and Lemma 5.4.9. □

The sets of black and white terms are stable by reduction.

Lemma 5.6.8. *Let Γ be a weakly well-typed global context and Δ be a weakly well-formed local context for Γ.*

If t_1 is black (respectively white) in Δ and $t_1 \to_{\beta\Gamma} t_2$, then t_2 is black (respectively white) in Δ.

Proof. It follows from weak subject reduction (Theorem 5.4.25). □

Applying a term preserves its color.

Lemma 5.6.9. *Let Γ be a weakly well-typed global context and Δ be a weakly well-formed local context for Γ. If $\Gamma;\Delta \vdash_w u : T_1$ and $\Gamma;\Delta \vdash_w u\ v : T_2$, then $\mathbf{Color}(T_1) = \mathbf{Color}(T_2)$.*

Proof. By inversion (Lemma 5.4.20), $\Gamma;\Delta \vdash_w u : A \longrightarrow B$, $\Gamma;\Delta \vdash_w v : A$ and $T_2 \equiv^w_\Gamma B$.
Moreover, by uniqueness of weak types (Theorem 5.4.26), $T_1 \equiv^w_\Gamma A \longrightarrow B$.
Finally, by Lemma 5.4.9, $\mathbf{Color}(T_1) = \mathbf{Color}(A \longrightarrow B) = \mathbf{Color}(B) = \mathbf{Color}(T_2)$. □

Redexes of black (respectively white) rewrite rules are black (respectively white).

Lemma 5.6.10. *Let Γ be a weakly well-typed global context and Δ be a weakly well-formed local context for Γ.*

If t_1 is weakly well-typed in Δ and is a redex of a black (respectively white) rewrite rule in Γ, then it is black (respectively white).

Proof. By definition of black and white rewrite rules and Lemma 5.6.9. □

The subterms at a black (respectively white) position are black (respectively white).

Lemma 5.6.11. *Let Γ be a weakly well-typed global context and Δ be a weakly well-formed local context for Γ.*

If u is weakly well-typed in Δ and p is a black (respectively white) position in u, then $u_{|p}$ is black (respectively white) in Δ.

Proof. By definition of black and white positions. □

Using the notions of black and white terms, we can define the relations of internal reduction $\to_{in}$ and of external reduction $\to_{out}$.

Definition 5.6.12 (The Relation $\to_{in}$)**.** *Let Γ be a weakly well-typed global context, Δ be a weakly well-formed local context for Γ and t_1 a term weakly well-typed in Δ.*

We write $t_1 \to_{in} t_2$ if $t_1 \to_{\beta\Gamma} t_2$ and the reduction occurs inside a white subterm (no matter the position white or black).

The relation $\to_{out}$ is the complement of $\to_{in}$ with respect to $\to_{\beta\Gamma}$.

Definition 5.6.13 (The Relation $\to_{out}$)**.** *We write $\to_{out}$ for $\to_{\beta\Gamma} \setminus \to_{in}$.*

Lemma 5.6.14. $(\to_{out}) \subset (\to_{\beta} \cup \to_{Black})$.

Proof. White rewrite rules rewrite white terms (Lemma 5.6.10). By definition $\to_{out}$-steps cannot involve white subterms. □

Lemma 5.6.15. *Under the hypotheses of Theorem 5.6.5, the relation $\to_{out}$ is left-linear.*

Proof. Follows from Lemma 5.6.14 and Assumption **(A2)**. □

5.6.2 Proof of Product Compatibility

We now prove Theorem 5.6.5. We follow the proof of Theorem 5.2.4 and we first prove three lemmas: a *postponement* lemma, a *commutation* lemma and a *product types* lemma.

In the remaining of the section we work under the hypotheses of Theorem 5.6.5.

Postponement

Lemma 5.6.16. *Let t_1 be a weakly well-typed term.*

If $t_1 \to_{in} u_1 \to_{out} t_2$, then there exists u_2 such that $t_1 \to_{out} u_2 \to_{in}^* t_2$.

Proof. By definition, the $\to_{out}$-redex cannot be inside the $\to_{in}$-contractum.

Moreover, the $\to_{in}$-contractum does not overlap with the $\to_{out}$-redex. Indeed, by **(A4)**, black rewrite rules do not filter on white subterms and the $\to_{in}$-contractum is white.

Thus, there are only two possible situations:

- either the two reductions are parallel, and then they can occur in any order;
- or the $\to_{in}$-contractum is inside the $\to_{out}$-redex and, since $\to_{out}$ is left-linear (Lemma 5.6.15), there exists u_2 such that $t_1 \to_{out} u_2 \to_{in}^* t_2$.

□

Lemma 5.6.17 (Postponement)**.** *Let t_1 be a weakly well-typed term. If $t_1 \to_{\beta\Gamma}^* t_2$, then there exists u such that $t_1 \to_{out}^* u \to_{in}^* t_2$.*

Proof. We proceed by induction on the n-tuple $(m_1, \ldots, m_n)$ ordered lexicographically, where n is the number of $(\to_{out})$-steps and m_i is the number of $(\to_{in})$-steps on the left of the i-th $\to_{out}$.

- If $(m_1, \ldots, m_n) = (0, \ldots, 0)$, then the reduction has the form $t_1 \to_{out}^* u \to_{in}^* t_2$.
- Otherwise, using Lemma 5.6.16 makes the tuple decrease.

□

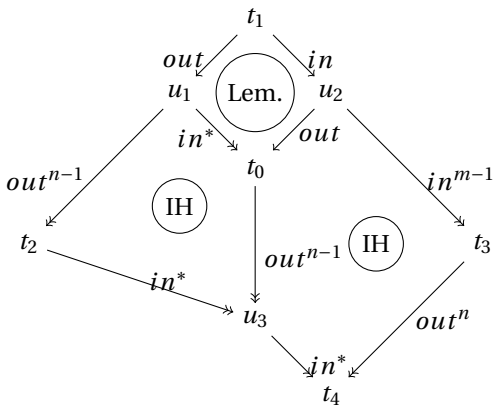


Figure 5.6: Proof of the Commutation Lemma

Commutation

Lemma 5.6.18. *Let t_1 be a black term.*

If $t_1 \to_{out} t_2$ and $t_1 \to_{in} t_3$, then there exists t_4 such that $t_3 \to_{out} t_4$ and $t_2 \to_{in}^* t_4$.

Proof. By definition, the $\to_{out}$-redex cannot be inside the $\to_{in}$-redex (with or without overlap).

Moreover, the $\to_{in}$-redexes do not overlap inside the $\to_{out}$-redex, since $\to_{out}$ does not filter on white subterms **(A4)**

Therefore, either the redexes are independent or the $\to_{in}$-redex is inside the $\to_{out}$-redex. Since $\to_{out}$ is left-linear (Lemma 5.6.15), in both cases, the reductions commute. □

Lemma 5.6.19 (Commutation)**.** *Let t_1 be a black term.*

If $t_1 \to_{out}^* t_2$ and $t_1 \to_{in}^* t_3$, then there exists t_4 such that $t_3 \to_{out}^* t_4$ and $t_2 \to_{in}^* t_4$.

Proof. We prove that, if $t_1 \to_{out}^n t_2$ and $t_1 \to_{in}^m t_3$, then there exists t_4 such that $t_3 \to_{out}^n t_4$ and $t_2 \to_{in}^* t_4$.

We proceed by induction on the pair (n, m) ordered lexicographically. We use Lemma 5.6.18 (Lem.) and the induction hypothesis (IH), as illustrated in Figure 5.6. □

Product Types

Beside the postponement lemma and the commutation lemma, we will need the following lemma, which says that internal reductions cannot introduce product types at the root of a term.

Lemma 5.6.20. *If $t_1 \to_{in}^* \Pi x : A_2.B_2$, then $t_1 = \Pi x : A_1.B_1$ with $A_1 \to_{in} A_2$ and $B_1 \to_{in} B_2$.*

Proof. By weak subject reduction, t_1 has the same type as the product type, that is to say **Type** or **Kind**. Therefore, t_1 is black. A $\to_{in}$-redex is either white or occurs inside a white term; therefore the reductions cannot occur at the root of t_1. Thus, t_1 is a product type. □

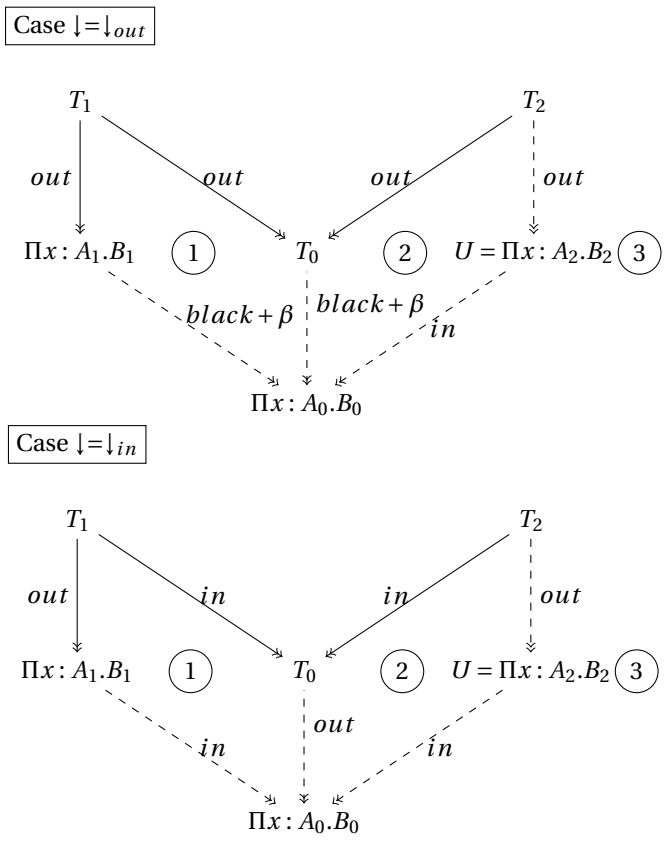


Figure 5.7: Proof of product compatibility

Proof of Theorem 5.6.5

We now have the tools to prove Theorem 5.6.5.

Proof of Theorem 5.6.5. Assume that we have $\Gamma;\Delta \vdash^w \Pi x : A_1.B_1 \equiv \Pi x : A_2.B_2$.

We have, for some n, $\Pi x : A_1.B_1 (\downarrow_{out} \cup \downarrow_{in})^n \Pi x : A_2.B_2$. Moreover, all the terms in the conversion are weakly well-typed.

We prove, by induction on n, that if $T_1 \rightarrow^*_{out} \Pi x : A_1.B_1$ and $T_1 (\downarrow_{out} \cup \downarrow_{in})^n T_2$, then there exist A_2 and B_2 such that $T_2 \rightarrow^*_{out} \Pi x : A_2.B_2$, $\Gamma;\Delta \vdash^w A_1 \equiv A_2$ and $\Gamma;\Delta(x : A_1) \vdash^w B_1 \equiv B_2$.

It boils down to proving, for $\downarrow=\downarrow_{out}$ and $\downarrow=\downarrow_{in}$, that, if $T_1 \downarrow T_2$ and $T_1 \rightarrow^*_{out} \Pi x : A_1.B_1$, then, for some A_2, B_2, $T_2 \rightarrow^*_{out} \Pi x : A_2.B_2$ with $\Gamma;\Delta \vdash^w A_1 \equiv A_2$ and $\Gamma;\Delta(x : A_1) \vdash^w B_1 \equiv B_2$.

- Case $\downarrow=\downarrow_{out}$ (see Figure 5.7). Let T_0 be the common reduct of T_1 and T_2: $T_1 \rightarrow^*_{out} T_0$ and $T_2 \rightarrow^*_{out} T_0$.

  1. By Lemma 5.6.14 and confluence of $\rightarrow_\beta \cup \rightarrow_{Black}$ **(A3)**, there exist A_0 and B_0 such that $\Pi x : A_1.B_1 (\rightarrow_\beta \cup \rightarrow_{Black})^* \Pi x : A_0.B_0$ and $T_0 (\rightarrow_\beta \cup \rightarrow_{Black})^* \Pi x : A_0.B_0$.

2. By the postponement lemma (Lemma 5.6.17), there exists U such that $T_2 \to^*_{out} U \to^*_{in} \Pi x : A_0.B_0$.
3. Finally, by the product types lemma (Lemma 5.6.20), $U = \Pi x : A_2.B_2$.

Moreover, by weak subject reduction (Theorem 5.4.25), we have $\Gamma;\Delta \vdash^w A_1 \equiv A_2$ and $\Gamma;\Delta(x : A_1) \vdash^w B_1 \equiv B_2$.

- Case $\downarrow=\downarrow_{in}$ (see Figure 5.7). Let T_0 be the common reduct of T_1 and T_2: $T_1 \to^*_{in} T_0$ and $T_2 \to^*_{in} T_0$.

1. By the commutation lemma (Lemma 5.6.19), there exist A_0 and B_0 such that $\Pi x : A_1.B_1 \to^*_{in} \Pi x : A_0.B_0$ and $T_0 \to^*_{out} \Pi x : A_0.B_0$.
2. By the postponement lemma (Lemma 5.6.17), there exists U such that $T_2 \to^*_{out} U \to^*_{in} \Pi x : A_0.B_0$.
3. Finally, by the product types lemma (Lemma 5.6.20), $U = \Pi x : A_2.B_2$.

Moreover, we have $\Gamma;\Delta \vdash^w A_1 \equiv A_2$ and $\Gamma;\Delta(x : A_1) \vdash^w B_1 \equiv B_2$.

□

5.6.3 Application

We now use Theorem 5.6.5 to prove that the global context from Section 5.3 satisfies product compatibility:

$\mathtt{U}_T$: **Type**.
$\dot{\epsilon}$: $\mathtt{U}_T \longrightarrow$ **Type**.
$\dot{\Pi}$: $\Pi a : \mathtt{U}_T.\Pi x : (\dot{\epsilon}\, a \longrightarrow \mathtt{U}_T).\mathtt{U}_T$.
$\dot{\epsilon}\ (\dot{\Pi}\ a\ b) \hookrightarrow \Pi x : \dot{\epsilon}\ a.\dot{\epsilon}\ (b\ x)$.

$\mathtt{Pair} : \mathtt{U}_T \longrightarrow \mathtt{U}_T \longrightarrow$ **Type**.
π_1 : $\Pi a : \mathtt{U}_T.\Pi b : \mathtt{U}_T.\mathtt{Pair}\ a\ b \longrightarrow \dot{\epsilon}\ a$.
π_2 : $\Pi a : \mathtt{U}_T.\Pi b : \mathtt{U}_T.\mathtt{Pair}\ a\ b \longrightarrow \dot{\epsilon}\ b$.
$\mathtt{mk\_pair}$: $\Pi.a : \mathtt{U}_T.\Pi.b : \mathtt{U}_T.\dot{\epsilon}\ a \longrightarrow \dot{\epsilon}\ b \longrightarrow \mathtt{Pair}\ a\ b$.
$\pi_1\ a_1\ b_1\ (\mathtt{mk\_pair}\ a_2\ b_2\ x\ y) \hookrightarrow x$.
$\pi_2\ a_2\ b_2\ (\mathtt{mk\_pair}\ a_2\ b_2\ x\ y) \hookrightarrow y$.

$\mathtt{mk\_pair}\ a\ b\ (\pi_1\ a\ b\ p)\ (\pi_2\ a\ b\ p) \hookrightarrow p$.

We take:

$Color(\mathtt{U}_T) = Color(\dot{\epsilon}) =$ **Black** and
$Color(\mathtt{Pair}) =$ **White**.

The (only) type-level rewrite rule is non-confusing since

Color$(\|\dot{\epsilon}\ (\dot{\Pi}\ a\ b)\|) =$ **Color**$(\|\dot{\epsilon}\|) = Color(\dot{\epsilon}) =$ **Black** and
Color$(\|\Pi x : \dot{\epsilon}\ a.\dot{\epsilon}\ (b\ x)\|) =$ **Color**$(\|\dot{\epsilon}\ (b\ x)\|) = Color(\dot{\epsilon}) =$ **Black**.

By Theorem 5.4.27, the rewrite rules are weakly well-typed.
The weak rewrite relation $\to^w_\Gamma$ is the relation generated by

$\|\dot{\epsilon}\ (\dot{\Pi}\ a\ b)\| \hookrightarrow \|\Pi x : \dot{\epsilon}\ a.\dot{\epsilon}\ (b\ x)\| =$ **Black** $\hookrightarrow$ **Black** $\longrightarrow$ **Black**.

Therefore, it is confluent and, by Lemma 5.4.17, weak product compatibility holds. It follows that Γ is weakly well-typed **(A1)**.

The only black rewrite rule is $(\dot{\epsilon}\ (\dot{\Pi}\ a\ b) \hookrightarrow x : \dot{\epsilon}\ a \longrightarrow \dot{\epsilon}\ (b\ x))$. It is left-linear **(A2)** and confluent together with β-reduction (Theorem 1.4.7) **(A3)**.

Finally, all the non-variable subterms of $\dot{\epsilon}\ (\dot{\Pi}\ a\ b)$ are at black positions **(A4)**.

Therefore, by Theorem 5.6.5, product compatibility holds for this global context in the Colored $\lambda\Pi$-Calculus Modulo defined in Section 5.5.

5.6.4 Back to the $\lambda\Pi$-Calculus Modulo

We do not know if the global context above verifies the product compatibility property for the unmodified $\lambda\Pi$-Calculus Modulo. However, the criterion is, in general, not applicable if we do not work with the weakly well-typed conversion. Here is a counter example:

$\mathtt{nat} : \mathbf{Type}$.
$A : \mathbf{Type}$.
$B : \mathbf{Type}$.
$a_1 : A$.
$a_2 : A$.
$T : A \longrightarrow \mathbf{Type}$.
$T\ a_1 \hookrightarrow A \longrightarrow A$.
$T\ a_2 \hookrightarrow B \longrightarrow B$.

$\mathtt{choose} : \mathtt{nat} \longrightarrow \mathtt{nat} \longrightarrow \mathtt{nat} \longrightarrow \mathtt{nat} \longrightarrow \mathtt{nat}$.
$\mathtt{choose}\ n\ n\ x\ y \hookrightarrow x$.
$\mathtt{choose}\ (\mathtt{S}\ n)\ n\ x\ y \hookrightarrow y$.

The only possible choice for a coloring is the following:
$Color(\mathtt{nat}) = \mathbf{White}$ and $Color(A) = Color(B) = Color(T) = \mathbf{Black}$.
For this coloring, the first two rules are black and the last two are white. One can check that the assumptions of Theorem 5.6.5 are verified. However, we have:

$T(\mathtt{choose}\ (\Omega\ S)\ (\Omega\ S)\ a_1\ a_2) \to_\Gamma T\ a_1 \to_\Gamma A \longrightarrow A$ and
$T(\mathtt{choose}\ (\Omega\ S)\ (\Omega\ S)\ a_1\ a_2) \to_\beta^* T\ (\mathtt{choose}\ (S\ (\Omega\ S))\ (\Omega\ S)\ a_1\ a_2) \to_{\beta\Gamma} T\ a_2 \to_{\beta\Gamma} B \longrightarrow B$.

But we do not have $\mathtt{nat} \equiv_{\beta\Gamma} A$; hence product compatibility does not hold. Of course, to be able to show that $A \longrightarrow A \equiv_{\beta\Gamma} B \longrightarrow B$, we need to go through the term $T(\mathtt{choose}\ (\Omega\ S)\ (\Omega\ S)\ a_1\ a_2)$ which is not weakly well-typed. In particular, the terms a_1 and a_2 have a black type but are used at black positions. Therefore, this conversion is not weakly well-typed.

5.7 Conclusion

We have studied the problem of proving product compatibility without using the confluence of the rewriting relations $\to_{\beta\Gamma}$ or $\to_{\beta\Gamma^b}$. Proving product compatibility without confluence allows us to consider global contexts with non-left-linear rewrite rules. Indeed, non-left-linear rewrite rules often generate non-confluent rewriting relations.

First, we have proven that product compatibility always holds for global contexts without Π-producing rewrite rules and, in particular, if the rewrite rules are at object level only.

Then, we have given a general criterion for product compatibility allowing mixing Π-producing rewrite rules and non-left-linear rewrite rules. To be able to prove this criterion, we modified the definition of the $\lambda\Pi$-Calculus Modulo to constrain conversion to go only through terms respecting a weak notion of typing. This new notion of typing, *weak typing*, has been defined and studied in detail. Its main feature is that it is an approximation of *usual* typing for which subject reduction is easy to prove.

Unfortunately, the criterion does not apply for the $\lambda\Pi$-Calculus Modulo. However, product compatibility for its variant can be used to prove other properties of the unmodified $\lambda\Pi$-Calculus Modulo. For instance, in Chapter 6, we will use Theorem 5.6.5 to prove the soundness of the type inference algorithm with respect to the $\lambda\Pi$-Calculus Modulo and the Colored $\lambda\Pi$-Calculus Modulo at the same time.

Confluence by Termination in the Colored $\lambda\Pi$-Calculus Modulo Beside being useful to prove the product compatibility property, confluence is also a key property to prove termination of the relation $\rightarrow_{\beta\Gamma}$ on well-typed terms in presence of type-level rewrite rules [Bla05a]. We conjecture that, for the Colored $\lambda\Pi$-Calculus Modulo, the assumption of confluence can be weakened to the confluence of $(\rightarrow_\beta \cup \rightarrow_{Black})$ when the assumptions of Theorem 5.6.5 are verified. This way, instead of using the confluence for all (untyped) terms to prove termination of well-typed terms, we use the confluence of $(\rightarrow_\beta \cup \rightarrow_{Black})$ on untyped terms to prove the termination of well-typed terms. Then, using Newman's lemma (Theorem 1.1.6), we can decide the confluence of $\rightarrow_{\beta\Gamma}$ for *well-typed* terms. If it holds, it means that the congruence is decidable for well-typed terms and, therefore, that type-checking is decidable.

Typed Conversion vs. Untyped Conversion When designing the Colored $\lambda\Pi$-Calculus Modulo, we have chosen to constrain the conversion to contain only weakly well-typed terms because weak subject reduction makes the set of weakly well-typed terms easy to manipulate. Another approach would be to constrain the conversion to contain only well-typed terms. This approach is the one used by Martin Löf's Type Theory [NPS90]. In this case, reduction is typed: rewriting and typing are mutually defined. The relation between systems with a typed reduction and systems with an untyped reduction is not easy to make. It has been studied by Adams [Ada06] and Siles and Herbelin [SH12]. They showed that, in the case of pure type systems with β-reduction, the two approaches (typed and untyped) are equivalent: the set of well-typed terms are the same. Their approach relies on a proof of confluence of the β-reduction based on parallel moves. We conjecture that their proof can be adapted for the $\lambda\Pi$-Calculus Modulo when the rewriting relation $\rightarrow_{\beta\Gamma}$ is parallel-closed [Hue80]. More generally, it would be interesting to develop a version of the $\lambda\Pi$-Calculus Modulo with a typed reduction and study the relation with the original $\lambda\Pi$-Calculus Modulo and with Martin Löf's Type Theory, in particular when the global context contains non-left-linear rewrite rules.

Chapter 6

Type Inference

Résumé Ce chapitre décrit les algorithmes de vérification de type pour les différents éléments du $\lambda\Pi$-Calcul Modulo : termes, contextes locaux et contextes globaux. On montre aussi que ces algorithmes sont corrects et complets en utilisant les résultats des chapitres précédents.

6.1 Introduction

In this chapter, we give algorithms to infer and check types for terms, check well-formedness of local contexts and check well-typedness of global contexts in the $\lambda\Pi$-Calculus Modulo. Using the theorems proved in the previous chapters, we study the conditions under which these algorithms are sound, complete and terminating.

We write the programs using pseudo code whose syntax is close to the *OCaml* programming language.

Since, we often need to reduce terms in the algorithms presented below, we assume given a function normalize that, given a global context Γ and a term t, computes a normal form for t with respect to the rewriting relation $\rightarrow_{\beta\Gamma}$.

```
val normalize : global_context → term → term
```

Of course, this function need not terminate if the term is not strongly normalizing. Therefore we will use it with global contexts Γ and terms t such that t is well-typed in Γ and $\rightarrow_{\beta\Gamma}$ is strongly normalizing on well-typed terms. However, in some cases we need to reduce potentially ill-typed terms. In these cases, to preserve termination, we will use a function bounded_normalize that computes a normal form but fails if no normal form is reached after a fixed number of steps. This function always terminates but may fail to compute a normal form for some entries.

```
val bounded_normalize : global_context → term → term
```

We also assume a function term_eq to compare terms up to α-equivalence (*i.e.*, up to renaming of bound variables).

```
val term_eq : term → term → boolean
```

6.2 Type Inference

We give below an algorithm for inferring a type for a given term.

```
1  let infer Γ Δ t =
2    match t with
3    | Kind → fail
4    | Type → Kind
5    | c    → Γ(c)
6    | x    → Δ(x)
7    | u v →
8      begin
9        let T_u = infer Γ Δ u in
10       let T_v = infer Γ Δ v in
11       match normalize Γ T_u with
12       | Π x:A.B →
13         if ( term_eq A (normalize Γ T_v) ) then B[x/v]
14         else fail
15       | _ → fail
16     end
17   | λ x:A.u →
18     begin
19       match infer Γ Δ A with
20       | Type → let B = infer Γ (Δ(x:A)) u in Π x:A.B
21       | _ → fail
22     end
23   | Π x:A.B →
24     begin
25       match infer Γ Δ A with
26       | Type →
27         begin
28           match infer Γ (Δ(x:A)) B with
29           | Kind → Kind
30           | Type → Type
31           | _ → fail
32         end
33       | _ → fail
34     end
```

This algorithm recursively inspects the shape of a term and applies the corresponding inference rules for the typing relation $\vdash$. The rule **(Conversion)** (Figure 2.4) is never used directly. However, in the case of an application, the inferred type of the function is reduced to ensure that it is a product type as in the premise of the **(Application)** rule.

Remark 6.2.1. *We do not need to fully normalize the term T_u on line 11. It suffices to reduce it until a product type is reached. However, in this case we need to normalize A before comparing it with the normal form of T_v.*

Theorem 6.2.2 (Soundness of infer)**.** *Let Γ be a well-typed global context and let Δ be a local context well-formed in Γ.*

If infer $\Gamma\ \Delta\ t = T$ *, then* $\Gamma; \Delta \vdash t : T$.

Proof. We proceed by induction on t.
We only detail the case where t is an application.

- Suppose that $t = u\ v$. We have infer $\Gamma\ \Delta\ u = T_u$, infer $\Gamma\ \Delta\ v = T_v$ and normalize $\Gamma\ T_u = \Pi\ x{:}A.B$.

  By induction hypothesis, we have $\Gamma;\Delta \vdash u : T_u$ and $\Gamma;\Delta \vdash v : T_v$.

  By subject reduction (Theorem 2.6.22) and conversion, we have $\Gamma;\Delta \vdash u : \Pi x : A.B$ and, by inversion, we get $\Gamma;\Delta \vdash A :$ **Type**.

  Since term_eq A (normalize $\Gamma\ T_v$), we have $A \equiv_{\beta\Gamma} T_v$.

  Therefore, by **(Conversion)**, we have $\Gamma;\Delta \vdash v : A$.

  Finally, using the **(Application)** rule, we get $\Gamma;\Delta \vdash u\ v : B[x/v]$.

□

Since we use the subject reduction property in the proof, we need Γ to be well-typed and Δ to be well-formed. To avoid the need for subject reduction, we can modify the algorithm to check that the result of the normalize function is well-typed.

```
let safe_infer Γ Δ t =
  match t with
  | (...)
  | u v →
    begin
      let T_u = safe_infer Γ Δ u in
      let T_v = safe_infer Γ Δ v in
      match normalize Γ T_u with
      | Π x:A.B →
        begin
          match safe_infer Γ Δ (Π x:A.B) with
          | Kind | Type →
            if term_eq A (normalize Γ T_v) then B[x/v]
            else fail
          | _ → fail
        end
      | _ → fail
    end
  | (...)
```

This allows us to drop the assumptions on Γ and Δ.

Theorem 6.2.3 (Soundness of safe_infer)**.** *If* safe_infer $\Gamma\ \Delta\ t = T$, *then* $\Gamma;\Delta \vdash t : T$.

Proof. We proceed by induction on t.
We only detail the case where t is an application.

- Suppose that $t = u\ v$. We have safe_infer $\Gamma\ \Delta\ u = T_u$, safe_infer $\Gamma\ \Delta\ v = T_v$, normalize $\Gamma\ T_u = \Pi\ x{:}A.B$, safe_infer $\Gamma\ \Delta\ (\Pi\ x{:}A.B) = s$.

  By induction hypothesis, we have $\Gamma;\Delta \vdash u : T_u$, $\Gamma;\Delta \vdash v : T_v$ and $\Gamma;\Delta \vdash \Pi x : A.B : s$.

  By inversion, we get $\Gamma;\Delta \vdash A :$ **Type**.

Since term_eq A (normalize Γ T_v), we have $A \equiv_{\beta\Gamma} T_v$.

Therefore, by **(Conversion)**, we have $\Gamma;\Delta \vdash u : \Pi x : A.B$ and $\Gamma;\Delta \vdash v : A$.

It follows, by **(Application)**, that $\Gamma;\Delta \vdash u\ v : B[x/v]$.

□

However, in practice we prefer to make sure that subject reduction holds and use the function infer instead of safe_infer. Indeed, the extra recursive call may be costly and it is, in most cases, useless.

Using the theorems of Chapter 5, we can also prove that the function infer is sound if the global context verifies the product compatibility property for the variant of the $\lambda\Pi$-Calculus Modulo introduced in Section 5.5.

Theorem 6.2.4. *Let Γ be a global context well-typed for $\vdash'$ (Definition 5.5.5) and let Δ be a local context well-formed in Γ for $\vdash'$.*

If infer Γ Δ $t = T$, *then* $\Gamma;\Delta \vdash t : T$.

Proof. Doing the same proof as in Theorem 6.2.2, we get that infer Γ Δ $t = T$ implies $\Gamma;\Delta \vdash' t : T$.

But, as remarked in Lemma 5.5.6, we have $(\vdash') \subset (\vdash)$. Therefore, infer Γ Δ $t = T$ implies $\Gamma;\Delta \vdash t : T$ as well. □

This means that, in the case where we cannot prove that a global context Γ verifies the product compatibility property (hence that it is well-typed) because it contains non-linear rewrite rules and Π-producing rewrite rules, we can still use Theorem 5.6.5 to prove that Γ is well-typed for $\vdash'$ and deduce the soundness of infer with respect to both $\vdash'$ and $\vdash$.

Regarding termination issues, note that recursive calls of infer are made on strict subterms. Therefore, the only possible source of non-termination is the use of the function normalize. However, if the relation $\rightarrow_{\beta\Gamma}$ is terminating on well-typed terms, then the function normalize always terminates.

Theorem 6.2.5 (Termination)**.** *Let Γ be a well-typed global context such that $\rightarrow_{\beta\Gamma}$ terminates on well-typed terms and let Δ be a local context well-formed in Γ.*

The call infer Γ Δ t *terminates on all entries t.*

Proof. Recursive calls are made on strict subterms. The function normalize is used on well-typed terms only. □

Without confluence, normal forms may not be unique, convertible terms may have different normal forms and terms convertible with a product type may have a normal form that is not a product type. On the other hand, without strong normalization, the function normalize may not terminate. For all these reasons, the function infer may fail to synthesize a type. However, if the relation $\rightarrow_{\beta\Gamma}$ is confluent and terminating, the algorithm is complete.

Theorem 6.2.6 (Completeness of infer)**.** *Let Γ be a well-typed global context such that the relation $\rightarrow_{\beta\Gamma}$ is confluent and terminating on well-typed terms and let Δ be a local context well-formed in Γ.*

If $\Gamma;\Delta \vdash t : T$, then there exists T_2 such that infer Γ Δ $t = T_2$.

Moreover, we have $\Gamma;\Delta \vdash t : T_2$ and $T_2 \equiv_{\beta\Gamma} T$.

Proof. First remark that, if $\Gamma;\Delta \vdash t : T$ and $\mathsf{infer}\ \ \Gamma\ \Delta\ t = T_2$, then, by soundness of infer (Theorem 6.2.2) and uniqueness of types (Theorem 2.6.25), we have $\Gamma;\Delta \vdash t : T_2$ and $T_2 \equiv_{\beta\Gamma} T$.

We proceed by induction on the typing derivation $\Gamma;\Delta \vdash t : T$.

- **(Sort)** Take $T_2 = \mathbf{Kind}$.
- **(Variable)** Take $T_2 = \Delta(x)$.
- **(Constant)** Take $T_2 = \Gamma(c)$
- **(Application)** Suppose that $t = u\ v$.

  By inversion, we have $\Gamma;\Delta \vdash u : \Pi x : V.U$, $\Gamma;\Delta \vdash v : V$ and $T \equiv_{\beta\Gamma} U[x/v]$.

  By induction hypothesis, there exist T_u and T_v such that $\mathsf{infer}\ \ \Gamma\ \Delta\ u = T_u$ and $\mathsf{infer}\ \ \Gamma\ \Delta\ v = T_v$.

  Moreover, since $T_u \equiv_{\beta\Gamma} \Pi x : U.V$ and $\rightarrow_{\beta\Gamma}$ is confluent and terminating, we have $\mathsf{normalize}\ \ \Gamma\ T_u = \Pi\ x{:}A.B$, for some A and B such that $T_v \rightarrow^*_{\beta\Gamma} A$.

  Therefore, we have $\mathsf{infer}\ \ \Gamma\ \Delta\ t = B[x/v]$.
- **(Abstraction)** By inversion and induction hypothesis (used twice).
- **(Product)** By inversion and induction hypothesis (used twice).
- **(Conversion)** By induction hypothesis.

□

6.3 Type Checking

By uniqueness of types, typechecking can be decomposed into inferring a type and then performing a convertibility test between the expected type and the inferred type.

```
let check Γ Δ t T =
  if term_eq T Kind then ( term_eq (infer Γ Δ t) Kind )
  else
    match infer Γ Δ T with
    | Type | Kind → ( term_eq (normalize Γ T) (normalize Γ (infer Γ Δ t)) )
    | _ → false
```

Theorem 6.3.1. *Let Γ be a well-typed global context and let Δ be a local context well-formed in Γ.*

- **(Soundness)** *If* $\mathsf{check}\ \ \Gamma\ \Delta\ t\ T = \mathbf{true}$, *then* $\Gamma;\Delta \vdash t : T$.
- **(Termination)** *If $\rightarrow_{\beta\Gamma}$ is terminating on well-typed terms, then check terminates on all entries t and T.*
- **(Completeness)** *Assuming that $\rightarrow_{\beta\Gamma}$ is confluent and terminating on well-typed terms, if* $\Gamma;\Delta \vdash t : T$, *then* $\mathsf{check}\ \ \Gamma\ \Delta\ t\ T = \mathbf{true}$.

Proof.

- **(Soundness)** Suppose that $T = \textbf{Kind}$. By Theorem 6.2.2 and the fact that sorts are only convertible to themselves (Lemma 2.3.5), we have $\Gamma; \Delta \vdash t : \textbf{Kind}$.

  Suppose that $T \neq \textbf{Kind}$. If we have check $\Gamma\ \Delta\ t\ T = \textbf{true}$, then infer $\Gamma\ \Delta\ T = s$ and infer $\Gamma\ \Delta\ t = T_2$ with $T \equiv_{\beta\Gamma} T_2$. Therefore, by Theorem 6.2.2, we have $\Gamma; \Delta \vdash t : T_2$ and $\Gamma; \Delta \vdash T : s$. By conversion, it follows that $\Gamma; \Delta \vdash t : T$.

- **(Termination)** By Theorem 6.2.5 and the assumption that $\rightarrow_{\beta\Gamma}$ is terminating on well-typed terms.

- **(Completeness)** Suppose that $T = \textbf{Kind}$. By Theorem 6.2.6 and Lemma 2.3.5, we have infer $\Gamma\ \Delta\ t = \textbf{Kind}$.

  Suppose that $T \neq \textbf{Kind}$. If we have $\Gamma; \Delta \vdash t : T$, then, by stratification (Lemma 2.6.10), $\Gamma; \Delta \vdash T : s$. By Theorem 6.2.6 and Lemma 2.3.5, infer $\Gamma\ \Delta\ T = s$ and infer $\Gamma\ \Delta\ t = T_2$ with $T \equiv_{\beta\Gamma} T_2$. Therefore, check $\Gamma\ \Delta\ t\ T = \textbf{true}$.

□

6.4 Well-Formedness Checking for Local Contexts

Checking that a local context is well-formed boils down to verifying that the types of its variables are well-typed.

```
let local_wf Γ Δ =
  match Δ with
  | ∅          → true
  | Δ0(x:T) →
    if (local_wf Γ Δ0) ∧ (x ∉ dom(Δ0)) then ( term_eq (infer Γ Δ0 T) Type )
    else false
```

Theorem 6.4.1. *Let Γ be a well-typed global context.*

- **(Soundness)** *If* local_wf $\Gamma\ \Delta = \textbf{true}$, *then* $\Gamma \vdash^{ctx} \Delta$.
- **(Termination)** *If the relation $\rightarrow_{\beta\Gamma}$ is terminating on well-typed terms, then* local_wf *terminates on all entries* Δ.
- **(Completeness)** *Assuming that the relation $\rightarrow_{\beta\Gamma}$ is confluent and terminating on well-typed terms, if $\Gamma \vdash^{ctx} \Delta$, then* local_wf $\Gamma\ \Delta = \textbf{true}$.

Proof. By induction on the local context Δ. The case $\Delta = \emptyset$ is trivial. Suppose $\Delta = \Delta_0(x : T)$.

- **(Soundness)** Suppose that local_wf $\Delta = \textbf{true}$. It means that local_wf $\Gamma\ \Delta_0 = \textbf{true}$, $x \notin \Delta_0$ and infer $\Gamma\ \Delta_0\ T = \textbf{Type}$. By induction hypothesis, we have $\Gamma \vdash^{ctx} \Delta_0$ and, by soundness of infer (Theorem 6.2.2), we have $\Gamma; \Delta_0 \vdash T : \textbf{Type}$. It follows that $\Gamma \vdash^{ctx} \Delta$.
- **(Termination)** By termination of infer (Theorem 6.2.5).
- **(Completeness)** Suppose that $\Gamma \vdash^{ctx} \Delta$. By inversion for $\vdash^{ctx}$ (Lemma 2.6.6), we have $\Gamma \vdash^{ctx} \Delta_0$, $x \notin \Delta_0$ and $\Gamma; \Delta_0 \vdash T : \textbf{Type}$. By induction hypothesis, we have local_wf $\Gamma\ \Delta_0 = \textbf{true}$. By completeness of infer (Theorem 6.2.6), we have infer $\Gamma\ \Delta_0\ T = \textbf{Type}$. It follows that local_wf $\Gamma\ \Delta_0 = \textbf{true}$.

□

6.5 Well-Typedness Checking for Rewrite Rules

In Chapter 3, we have given several methods to prove that a rewrite rule is well-typed. We now implement the check based on weakly well-formed rewrite rules (Theorem 3.6.6).

6.5.1 Solving Unification Constraints

This criterion is based on the computation of a permanent presolution (Definition 3.6.1). The function find_presolution takes as arguments a global context Γ, a set of variables $\mathcal{V}$ and a set of constraints $\mathscr{C}$ to solve. It computes a presolution for $\mathscr{C}$ whose domain is in $\mathcal{V}$.

```
let find_presolution Γ V C =
  match C with
  | ∅ → id
  | C0(t1,t2) →
    begin
      match bounded_normalize Γ t1, bounded_normalize Γ t2 with
      (* Atomic Terms *)
      | Kind, Kind
      | Type, Type
      | c1, c2 when ( c1 = c2 )
      | x1, x2 when ( x1 = x2 ∧ x1 ∉ V ) → find_presolution Γ V C0
      (* Solved Equations *)
      | x, t | t, x when ( x ∈ V ) →
        if x ∉ FV(t) then
          let C2 = map ( fun (a,b) → (a[x/t],b[x/t]) ) C0 in
            ( find_presolution Γ V C2) ⊎ { x → t }
        else fail (* occur check *)
      (* Rigid/Rigid Equation *)
      | c1 u1, c2 u2 when ( c1 = c2 ∧ |u1|=|u2| ∧ not (is_definable c1) ) →
        find_presolution Γ V (C0(u1,u2))
      | x1 u1, x2 u2 when ( x1 = x2 ∧ x1 ∉ V ∧ |u1|=|u2| ) →
        find_presolution Γ V (C0(u1,u2))
      | λ x:A1.u1, λ x:A2.u2 → find_presolution Γ V (C0(A1,A2)(u1,u2))
      | Π x:A1.B1, Π x:A2.B2 → find_presolution Γ V (C0(A1,A2)(B1,B2))
      (* Ignored Constraints *)
      | x u1, t | t, x u2 when ( x ∈ V )
      | c , t | t, c when (is_definable c)
      | c u, t | t, c u when (is_definable c) → find_presolution Γ V C0
      (* Failure *)
      | _, _ → fail
    end
```

To compute a permanent presolution for a set of equations, we adapt Herbrand's unification algorithm [Ter03]. We recursively unify syntactically the normalized equations, but we drop them when we do not know how to solve them. This is possible because we do not need to compute a solution of the unification problem but only a prefix to all solutions. Remark that we need to use bounded_normalize instead of normalize because we do not know if the constraints are well-typed.

Lemma 6.5.1. *Let Γ be a global context, $\mathcal{V}$ be a set of variables and $\mathscr{C}$ be a set of constraints.*

- *If* find_presolution $\Gamma\ \mathcal{V}\ \mathscr{C} = \sigma$, *then* $\sigma \in PreSol_\Gamma^{per}(\mathcal{V},\mathscr{C})$.
- *The function* find_presolution *terminates on all entries.*

Proof. By induction on the pair (n,E) ordered lexicographically where n is the number of variables in $\mathcal{V}$ occurring in $\mathscr{C}$ and E is the set $\{\ a \mid \exists b, (a,b) \in \mathscr{C} \vee (b,a) \in \mathscr{C}\ \}$ ordered by the multiset extension [DM79] of the subterm ordering.

Let Γ_2 a safe extension of Γ and σ such that $\sigma(A) \equiv_{\beta\Gamma_2} \sigma(B)$, for all $(A,B) \in \mathscr{C}$.

We only detail the case where $\mathscr{C} = \mathscr{C}_0(t_1,t_2)$ with bounded_normalize $\Gamma\ t_1 = c\ \vec{u}_1$, bounded_normalize $\Gamma\ t_2 = c\ \vec{u}_2$ and c is a static symbol.

- We have $\sigma(c\vec{u}_1) \equiv_{\beta\Gamma_2} \sigma(c\vec{u}_2)$. Moreover $\to_{\beta\Gamma_2}$ is confluent by hypothesis of safeness for Γ_2. Since c is static, by confluence of $\to_{\beta\Gamma_2}$, we have $\sigma(\vec{u}_1) \equiv_{\beta\Gamma_2} \sigma(\vec{u}_2)$. Therefore any solution of $\mathscr{C}_0(\vec{u}_1,\vec{u}_2)$ is a solution of $\mathscr{C}$ and any permanent presolution of $\mathscr{C}_0(\vec{u}_1,\vec{u}_2)$ is a permanent presolution of $\mathscr{C}$.

  It follows, by induction hypothesis, that σ is a permanent presolution of $\mathscr{C}$.

□

6.5.2 Checking Weak Well-Formedness of Rewrite Rules

We now (partially) implement the relations $\Gamma;\Delta_1,\Sigma,\mathscr{C}_1 \Vdash_i t \Rightarrow (\Delta_2,T,\mathscr{C}_2)$ and $\Gamma;\Delta_1;\Sigma,\mathscr{C}_1 \Vdash_c t \Leftarrow T \mid (\Delta_2,\mathscr{C}_2)$, necessary to type the left-hand side of weakly well-formed rewrite rules.

```
let infer_lhs Γ Δ Σ 𝒞 t =
  match t with
  | c   → (Δ,Γ(c),𝒞)
  | x   → (Δ,(ΔΣ)(x),𝒞)
  | u v  →
  begin
    let (Δ2,P,𝒞2) = infer_lhs Γ Δ Σ 𝒞 u in
    match bounded_normalize Γ P with
    | Π x:A.B → let (Δ3,𝒞3) = check_lhs Γ Δ2 Σ 𝒞2 v A in (Δ3,B[x/v],𝒞3)
    | _ → fail
  end
  | _ → fail

let check_lhs Γ Δ Σ 𝒞 t T =
  match t with
  | x  when ( x ∉ dom(ΔΣ) ) →
    if FV(T) ∩ dom(Σ) = ∅ then (Δ(x:T), 𝒞)
    else fail
  | u x when ( x ∈ dom(Σ) ) → check_lhs Γ Δ Σ 𝒞 u (Π y:Σ(x).T)
  | λ x:A.u    →
    begin
      match bounded_normalize Γ T with
      | Π x:A2.B →
        let (Δ2,𝒞2) = check_lhs Γ Δ (Σ(x:A2)) 𝒞 u B in (Δ2,𝒞2)
```

```
        | _ → fail
    end
  | _ → let (Δ2, T2, C2) = infer_lhs Γ Δ Σ C t in (Δ2, C2(T, T2))
```

We chose to implement only some inference rules of Figure 3.3 and Figure 3.4. In particular, we do not check type annotations on abstraction and do not infer the type of abstractions. Since abstractions always occur as arguments in left-hand sides of rewrite rules, their types is always known in advance, and therefore the information given by the type annotation is redundant.

Lemma 6.5.2 (Soundness of lhs_infer and lhs_check)**.**

- *If* infer_lhs $\Gamma\ \Delta_1\ \Sigma\ \mathscr{C}_1\ t = (\Delta_2, T, \mathscr{C}_2)$, *then* $\Gamma; \Delta_1, \Sigma, \mathscr{C}_1 \Vdash_i t \Rightarrow (\Delta_2, T, \mathscr{C}_2)$.
- *If* check_lhs $\Gamma\ \Delta_1\ \Sigma\ \mathscr{C}_1\ t\ T = (\Delta_2, \mathscr{C}_2)$, *then* $\Gamma; \Delta_1; \Sigma, \mathscr{C}_1 \Vdash_c t \Leftarrow T \mid (\Delta_2, \mathscr{C}_2)$.

Proof. By induction on t. □

Finally, we can implement the weak well-formedness check for rewrite rules.

```
let rewrite_wf Γ (f ū ↪ v) =
  let (Δ, C, T) = infer_lhs Γ ∅ ∅ ∅ (f ū) in
  let σ = find_presolution Γ FV(f ū) C in
  if ( is_definable f ∧ local_wf Γ σ(Δ) ) then
    check Γ σ(Δ) v σ(T)
  else false
```

Lemma 6.5.3. *Let* Γ *be a well-typed global context.*

- **(Soundness)** *If* rewrite_wf $\Gamma\ (f\ \vec{u} \hookrightarrow v) =$ **true**, *then* $(f\,\vec{u} \hookrightarrow v)$ *is weakly well-formed and* f *is a definable symbol.*
- **(Termination)** *If* $\to_{\beta\Gamma}$ *is terminating on well-typed terms, then* rewrite_wf *terminates on all entries* $(u \hookrightarrow v)$.

Proof.

- **(Soundness)** Follows from the soundness of infer_lhs and check_lhs (Lemma 6.5.2), soundness of find_presolution (Lemma 6.5.1), soundness of local_wf (Theorem 6.4.1) and soundness of check (Theorem 6.3.1).
- **(Termination)** The termination of rewrite_wf follows from the termination of infer_lhs and check_lhs (recursive calls are made on strict subterms), of find_presolution (Lemma 6.5.1), of local_wf (Theorem 6.4.1) and check (Theorem 6.3.1).

□

6.6 Checking Well-Typedness for Global Contexts

We have shown in Chapter 3, that β-well-formed global contexts (Definition 4.5.8) are well-typed. We now have most of the tools to check that a global context is β-well-formed.

The only missing element is confluence checking for $\rightarrow_{\beta\Gamma}$. We consider that we have a function is_confluent that, given a global context Γ, tries to decide if $\rightarrow_{\beta\Gamma^b}$ is confluent. This property is undecidable but we may assume that this function implements a simplification of Van Oostrom's development closed theorem (Theorem 4.6.6) such as the one in [AYT09].

```
type ext_bool = ext_true | ext_false | maybe
val is_confluent : global_context → ext_bool
```

The implementation of the β-well-formedness test is now straightforward.

```
let global_wf Γ =
  match Γ with
  | ∅ → true
  | Γ0(c:T) →
    if (global_wf Γ0) ∧ (c ∉ dom(Γ0)) then term_eq (infer Γ0 ∅ T) Type
    else false
  | Γ0(C:K) →
    if (global_wf Γ0) ∧ (C ∉ dom(Γ0)) then term_eq (infer Γ0 ∅ K) Kind
    else false
  | Γ0Ξ →
    if (global_wf Γ0) then
      (is_confluent Γ = ext_true) ∧ (∀ (u ↪ v) ∈ Ξ, rewrite_wf Γ0 (u ↪ v))
    else false
```

Theorem 6.6.1 (Soundness of global_wf)**.** *If* global_wf Γ = **true**, *then* Γ *is well-typed.*

Moreover, global_wf *terminates on all entries.*

Proof. We prove by induction on Γ that it is β-well-formed. Hence, by Theorem 4.5.10, it is well-typed.

- Suppose that $\Gamma = \emptyset$. Trivial.
- Suppose that $\Gamma = \Gamma_0(c : T)$. We have global_wf Γ_0 = **true**, infer $\Gamma_0\ \emptyset\ T$ = **Type** and $c \notin dom(\Gamma_0)$.

  By induction hypothesis, Γ_0 is β-well-formed (and well-typed). By soundness of infer (Theorem 6.2.2), we have $\Gamma; \emptyset \vdash T$: **Type**.

  It follows that Γ is β-well-formed.
- Suppose that $\Gamma = \Gamma_0(C : K)$. We have global_wf Γ_0 = **true**, infer $\Gamma_0\ \emptyset\ \mathsf{K}$ = **Kind** and $C \notin dom(\Gamma_0)$.

  By induction hypothesis, Γ_0 is β-well-formed (and well-typed). By soundness of infer (Theorem 6.2.2), we have $\Gamma; \emptyset \vdash K$: **Kind**.

  It follows that Γ is β-well-formed.
- Suppose that $\Gamma = \Gamma_0\Xi$. We have global_wf Γ_0 = **true**, for all $(u \hookrightarrow v) \in \Xi$, rewrite_wf $\Gamma_0\ (u \hookrightarrow v)$ and is_confluent Γ = **ext_true**.

  By induction hypothesis, Γ_0 is β-well-formed. By soundness of rewrite_wf (Lemma 6.5.3), the rewrite rules in Ξ are weakly well-formed for Γ_0 and are of

the form $(f\,\vec{u} \hookrightarrow v)$ where f is a definable symbol. Moreover, $\rightarrow_{\beta\Gamma^b}$ is confluent.

It follows that Γ is β-well-formed.

Since, recursive calls are made on strictly smaller global contexts, termination follows from the termination of infer (Theorem 6.2.5) and rewrite_wf (Lemma 6.5.3). □

6.7 Conclusion

We have given algorithms to infer and check types for terms, check well-formedness of local contexts and check well-typedness of global contexts in the $\lambda\Pi$-Calculus Modulo and we have proven that they are sound, complete and terminating. These algorithms have been implemented in DEDUKTI [BCHS], our type checker for the $\lambda\Pi$-Calculus Modulo.

Conclusion

(English version follows.)

Dans cette thèse, nous avons développé une nouvelle présentation formelle du $\lambda\Pi$-Calcul Modulo qui sert de fondation théorique au vérificateur de preuve DEDUKTI, que nous avons également implémenté.

Résumé des Contributions

- Dans le chapitre 2, nous avons présenté une nouvelle version du $\lambda\Pi$-Calcul Modulo, un système de types à base de types dépendants et de règles de réécriture, et nous en avons fait une étude détaillée. Nous avons introduit la notion de contexte global bien typé de manière à caractériser les contextes globaux pour lesquels le système de types vérifie certaines propriétés élémentaires telles que la préservation du type par réduction ou encore l'unicité des types. Un contexte global est bien typé si les déclarations de constantes sont bien typées, les règles de réécriture sont bien typées et la propriété de la compatibilité du produit est vérifiée. Nous avons aussi étudié le Calcul des Constructions Modulo, une extension du $\lambda\Pi$-Calcul Modulo avec du polymorphisme et des opérateurs de type.
- Dans le chapitre 3, nous avons étudié la propriété de bon typage des règles de réécriture. En partant de la preuve que les règles de réécriture fortement bien formées sont bien typées, on généralise le résultat en introduisant la notion de règle de réécriture faiblement bien formée. Contrairement aux règles fortement bien formées, les règles faiblement bien formées peuvent avoir un membre gauche non algébrique et mal typé. Ainsi on permet le filtrage sous les lieurs. On permet aussi de se débarrasser d'un certain type de non-linéarité à gauche dû aux contraintes du typage, et de préserver la confluence du système de réécriture. Enfin, nous avons donné une caractérisation exacte de la notion de bon typage pour les règles de réécriture vue comme un problème d'inclusion entre des ensembles de solutions de deux problèmes d'unification et nous en avons déduit l'indécidabilité du problème.
- Dans le chapitre 4, nous avons résolu un problème relatif à la présence de lieurs dans le membre gauche des règles de réécriture : la combinaison de telles règles avec la β-réduction génère un système de réécriture non confluent ; la confluence ne peut donc pas être utilisée pour prouver la propriété de compatibilité du produit ni pour décider le typage. Nous avons défini une notion de réécriture modulo β pour le $\lambda\Pi$-Calcul Modulo qui ne souffre pas du même

problème. Nous avons montré que la confluence de cette nouvelle notion de réécriture peut être utilisée pour prouver la propriété de compatibilité du produit et pour décider le typage. La réécriture modulo β est définie grâce à un encodage des termes du $\lambda\Pi$-Calcul Modulo dans un système de réécriture d'ordre supérieur. Ceci permet d'importer dans le $\lambda\Pi$-Calcul Modulo les résultats de confluence prouvés dans le contexte des systèmes de réécriture d'ordre supérieur. Nous avons aussi décrit comment implémenter efficacement la réécriture modulo β grâce à la compilation des règles de réécriture en arbres de décision.

- Dans le chapitre 5, nous avons étudié l'impact de l'ajout de règles de réécriture non linéaires à gauche dans le $\lambda\Pi$-Calcul Modulo. Les règles non linéaires à gauche génèrent la plupart du temps un système de réécriture non confluent lorsqu'on les associe à la β-réduction. Ceci rend la preuve de la propriété de la compatibilité du produit plus difficile. Nous montrons que cette propriété est vérifiée, indépendamment de la confluence, lorsqu'on ne considère que des régles de réécriture au niveau objet. Nous donnons aussi un critère pour prouver la propriété de la compatibilité du produit dans le cas où le contexte global contient des règles non linéaires à gauche et des règles produisant des types produits. Ce critère est applicable dans le $\lambda\Pi$-Calcul Modulo coloré dans lequel la relation de conversion est faiblement typée.

- Dans le chapitre 6, nous avons donné des algorithmes pour inférer et vérifier le type d'un terme, pour vérifier qu'un contexte local est bien formé, pour vérifier qu'une règle de réécriture est faiblement bien formée et pour vérifier qu'un contexte global est β-bien-formé. Nous prouvons aussi que ces algorithmes sont corrects et complets en utilisant les résultats des chapitres précédents.

Perspectives

Terminaison Parmi les conditions qui rendent la vérification de type décidable, nous n'avons pas étudié la terminaison du système de réécriture sur les termes bien typés. La terminaison est donc un prolongement logique à ce travail. La terminaison de calculs incorporant des types dépendants et des règles de réécriture a été étudiée par de nombreux auteurs (cf Section 2.9). Nous pensons que le travail de Blanqui [Bla04] sur la terminaison à base de types dans le Calcul des Constructions Algébriques peut être aisément adapté au cadre du $\lambda\Pi$-Calcul Modulo. Cependant, certaines modifications sont nécessaires pour prendre en compte notre notion de règle de réécriture bien typée ainsi que la réécriture modulo β. L'implémentation d'un critère de terminaison à base de types [Abe10] dans DEDUKTI serait aussi un défi intéressant. La terminaison de la réécriture modulo $\beta\eta$ a été étudiée dans [Bla15], mais dans un cadre simplement typé seulement.

Réécriture modulo une théorie équationnelle La commutativité est une propriété de certains opérateurs algébriques qui ne se comporte pas bien sous forme de règle de réécriture. En arithmétique, par exemple, il est commode de raisonner modulo la commutativité de l'addition et de la multiplication. Cependant, la règle de réécriture

$\texttt{plus}\ n\ m \hookrightarrow \texttt{plus}\ m\ n$.

génère un système de réécriture qui, de manière évidente, ne termine pas, ce qui nous empêche de l'utiliser dans DEDUKTI. Une approche possible pour résoudre ce problème est de considérer une notion de réécriture plus générale qui filtre modulo une théorie équationnelle donnée, par exemple modulo la commutativité de l'addition et de la multiplication. De la même manière que pour la réécriture modulo β, la confluence de la réécriture modulo théorie permettrait de prouver la compatibilité du produit et de décider le typage.

Conversion typée Quand nous avons défini le $\lambda\Pi$-Calcul Modulo, nous sommes parti d'un notion de réécriture non typée. L'idée était, d'une part, d'avoir une séparation nette entre les notions de réécriture et de typage et, d'autre part, d'avoir une définition du calcul aussi proche que possible de son implémentation. Dans le chapitre 5, nous avons vu que cette notion libérale de la réécriture était problématique lorsque l'on considère des règles non linéaires à gauche, car la confluence est immédiatement perdue. Ce problème pourrait être résolu en considérant des notions de réécriture et de conversion typées. Cependant, nous ne savons pas dans quelle mesure cette restriction modifie la théorie du $\lambda\Pi$-Calcul Modulo et si les résultats présentés dans cette thèse restent valides. Une comparaison précise entre les approches typée et non typée de la réécriture nous en apprendrait davantage.

Au-delà du Calcul des Constructions Modulo Comme nous l'avons montré, le $\lambda\Pi$-Calcul Modulo est un cadre logique très puissant lorsque l'on considère l'interprétions jugement/type. D'un autre côté, par la correspondance de Curry-Howard, le système de type correspond à la logique minimale du premier ordre modulo, un système logique proche de la Déduction Modulo [DHK03]. Si l'on ajoutait du polymorphisme, des opérateurs de type (cf Section 2.8) et des univers, on obtiendrait une logique d'ordre supérieur modulo très puissante. Nous pensons que ce système pourrait servir de fondation pour un nouvel assistant de preuve fondé sur la réécriture. Cette idée est très proche du projet de Chrząszcz and Walukiewicz-Chrząszcz d'ajouter de la réécriture à l'assistant de preuve Coq [CWC07].

Conclusion

In this thesis, we have developed a new formal presentation of the $\lambda\Pi$-Calculus Modulo to serve as a theoretical foundation its the proof checker, DEDUKTI, which we implemented.

Summary of Contributions

- In Chapter 2, we have given a new presentation of the $\lambda\Pi$-Calculus Modulo, a type system featuring dependent types and rewrite rules, together with a detailed theoretical study of the system. We have introduced the notion of well-typed global contexts to characterize global contexts for which basic properties such as subject reduction and uniqueness of types hold. A global context is well-typed if constant declarations are well-typed, rewrite rules are well-typed and product compatibility holds. We have also studied the Calculus of Constructions Modulo, an extension of the $\lambda\Pi$-Calculus Modulo with polymorphism and type operators.

- In Chapter 3, we have investigated the property of well-typedness of rewrite rules. Starting from the proof that strongly well-formed rewrite rules are well-typed, we have generalized the result through the notion of weakly well-formed rewrite rules. Unlike strongly well-formed rewrite rules, weakly well-formed rewrite rules can have a left-hand side that is neither algebraic nor well-typed. This allows matching under binders. It also permits getting rid of the non-left-linearities due to typing constraints, in order to preserve a confluent rewrite system. Finally, we have given an exact characterization of the well-typedness property for rewrite rules as a problem of inclusion between solutions of two unification problems and we have used this characterization to show the undecidability of well-typedness of rewrite rules.

- In Chapter 4, we have solved a problem arising from the presence of binders on the left-hand side of rewrite rules: confluence is lost for the combination of β-reduction with these rewrite rules; therefore, confluence cannot be used to prove product compatibility or decidability of type-checking. We have introduced a notion of rewriting modulo β for the $\lambda\Pi$-Calculus Modulo that does not suffer this problem. We have shown that the confluence of this new rewriting relation can be used to prove product compatibility and decidability of type-checking. Rewriting modulo β is defined through an encoding of the terms of the $\lambda\Pi$-Calculus Modulo into Higher-Order Rewrite Systems. As a result, it allows using confluence criteria designed for HRSs to prove the confluence of rewriting modulo β in the $\lambda\Pi$-Calculus Modulo. We have also

described a way to efficiently implement rewriting modulo β by compiling rewrite rules into decision trees.

- In Chapter 5, we have studied the impact of non-left-linear rewrite rules in the $\lambda\Pi$-Calculus Modulo. Non-left-linear rewrite rules almost always generate a non-confluent rewrite system when combined with β-reduction. This makes the proof of product compatibility more difficult. We have shown that product compatibility holds for object-level rewrite systems independently of confluence. We have also given a criterion for proving product compatibility for global contexts containing at the same time non-left-linear rewrite rules and (type-level) Π-producing rewrite rules. This criterion applies to the Colored $\lambda\Pi$-Calculus Modulo where the conversion rule is constrained to be weakly-well-typed.
- In Chapter 6, we have given algorithms to infer or check the type of a term, check that a local context is well-formed, check that a rewrite rule is weakly well-formed and check that a global context is β-well-formed. We have proved that these algorithms are sound and complete using the results of the previous chapters.

Perspectives

Termination Among the conditions making type-checking decidable, we have not studied the termination of rewrite systems on well-typed terms. This makes termination analysis an obvious continuation of this work. Termination in calculus mixing dependent types and rewrite rules has been studied by several authors (see Section 2.9). We believe that the work of Blanqui [Bla04] on the termination in the Calculus of Algebraic Constructions using type-based technics can be adapted to the $\lambda\Pi$-Calculus Modulo. However, Blanqui's criteria might need some modifications to cope with our notion of well-typed rewrite rules and our notion of rewriting modulo β. The implementation of a type-based termination [Abe10] criterion in DEDUKTI is also an interesting challenge.

Rewriting Modulo an Equational Theory Commutativity is a property of algebraic operators that does not behave well as a rewrite rule. In arithmetics for instance, it can be convenient to reason modulo the commutativity of the addition and the multiplication. However, the rewrite rule

$\texttt{plus}\ n\ m \hookrightarrow \texttt{plus}\ m\ n.$

generates a rewrite relation which is obviously non-terminating, preventing us to use it in DEDUKTI. One possible approach to solve this issue would be to consider a more general notion of rewriting where matching is done modulo some equational theory, for instance modulo the commutativity of addition and multiplication. As for rewriting modulo β, we expect that the confluence of rewriting modulo an equational theory ensures product compatibility and restores the decidability of type checking.

Typed Conversion When defining the $\lambda\Pi$-Calculus Modulo we chose to rely on an untyped notion of rewriting. The idea was to have a clean separation between

rewriting and typing and a system as close as possible to its implementation. In Chapter 5, we have seen that this liberal notion of rewriting is problematic when we consider non-left-linear rewrite rules because confluence is immediately lost. This issue could be solved by considering typed notions of rewriting and conversion. However, we do not know to which extent taking a typed notion of conversion changes the theory of the $\lambda\Pi$-Calculus Modulo and if the results presented in this thesis still hold in this setting. A precise comparison between both systems (with and without typed conversion) would be interesting.

Beyond the Calculus of Construction Modulo The $\lambda\Pi$-Calculus Modulo has shown to be a powerful logical framework, seeing it through the judgment-as-type interpretation. On the other hand, by the proposition-as-type interpretation, it corresponds to minimal predicate logic modulo, a proof system close to Deduction Modulo [DHK03]. By adding polymorphism, type operators (as in Section 2.8) and universes we would get a very powerful higher-order logic modulo. We believe that this logic can be used as a foundation for a new proof assistant based on rewriting. This idea is very close to the project of Chrząszcz and Walukiewicz-Chrząszcz to add rewriting to the Coq proof assistant [CWC07].

Bibliography

[AB14] A. Assaf and G. Burel. Translating HOL to Dedukti. Unpublished, 2014.

[Abe10] A. Abel. MiniAgda: Integrating Sized and Dependent Types. In *Partiality And Recursion in Interative Theorem (PAR 2010)*, 2010.

[Ada06] R. Adams. Pure Type Systems with Judgemental Equality. *Journal of Functional Programming*, 2006.

[ARCT11] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita Interactive Theorem Prover. In *Automated Deduction (CADE), 23rd International Conference, Proceedings*, 2011.

[Ass] A. Assaf. Krajono. `https://www.rocq.inria.fr/deducteam/Krajono/index.html`.

[Ass15] A. Assaf. A Calculus of Constructions with Explicit Subtyping. In *Types for Proofs and Programs (TYPES '14), 20th International Conference, Post-Proceedings*, 2015.

[AYT09] T. Aoto, J. Yoshida, and Y. Toyama. Proving Confluence of Term Rewriting Systems Automatically. In *Rewriting Techniques and Applications (RTA), 20th International Conference, Proceedings*, 2009.

[B+91] H. Barendregt et al. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1991.

[Bar90] F. Barbanera. Adding Algebraic Rewriting to the Calculus of Constructions: Strong Normalization Preserved. In *Conditional and Typed Rewriting Systems (CTRS), 2nd International Workshop, Proceedings*, 1990.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7 - Paris Diderot, 1999.

[BB12] M. Boespflug and G. Burel. CoqInE : Translating the calculus of inductive constructions into the $\lambda\Pi$-calculus modulo. In *Proof Exchange for Theorem Proving (PxTP), Second International Workshop*, 2012.

[BCH12] M. Boespflug, Q. Carbonneaux, and O. Hermant. The lambda-Pi-calculus Modulo as a Universal Proof Language. In *Proof Exchange for Theorem Proving (PxTP), Second International Workshop, Proceedings*, 2012.

[BCHS] M. Boespflug, Q. Carbonneaux, O. Hermant, and R. Saillard. Dedukti. `http://dedukti.gforge.inria.fr`.

[BDD07] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 14th International Conference, Proceedings*, 2007.

[BDN09] A. Bove, P. Dybjer, and U. Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics (TPHOLs), 22nd International Conference, Proceedings*, 2009.

[BF93a] F. Barbanera and M. Fernández. Combining First and Higher Order Rewrite Systems with Type Assignment Systems. In *Typed Lambda Calculi and Applications (TLCA), International Conference, Proceedings*, 1993.

[BF93b] F. Barbanera and M. Fernández. Modularity of Termination and Confluence in Combinations of Rewrite Systems with lambda_omega. In *Automata, Languages and Programming (ICALP), 20nd International Colloquium, Proceedings*, 1993.

[BFG94] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalization and Confluence in the Algebraic-lambda-Cube. In *Logic in Computer Science (LICS), Ninth Annual Symposium, Proceedings*, 1994.

[BG95] G. Barthe and H. Geuvers. Modular properties of algebraic type systems. In *Higher-Order Algebra, Logic, and Term Rewriting (HOA), Second International Workshop, Selected Papers*, 1995.

[BJO99] F. Blanqui, J. P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In *Rewriting Techniques and Applications (RTA), 10th International Conference, Proceedings*, 1999.

[Bla04] F. Blanqui. A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems. In *Rewriting Techniques and Applications (RTA), 15th International Conference, Proceedings*, 2004.

[Bla05a] F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 2005.

[Bla05b] F. Blanqui. Inductive types in the Calculus of Algebraic Constructions. *Fundamenta Informaticae*, 2005.

[Bla15] F. Blanqui. Termination of rewrite relations on lambda-terms based on Girard's notion of reducibility. *Theoretical Computer Science*, 2015. to appear.

[BMM03] E. Brady, C. McBride, and J. McKinna. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs (TYPES), International Workshop, Revised Selected Papers*, 2003.

[Boe11] M. Boespflug. *Conception d'un noyau de vérification de preuves pour le lambda-Pi-calcul modulo.* PhD thesis, École Polytechnique, 2011.

[BS91] U. Berger and H. Schwichtenberg. An Inverse of the Evaluation Functional for Typed lambda-Calculus. In *Logic in Computer Science (LICS), Sixth Annual IEEE Symposium, Proceedings*, 1991.

[Bur13] G. Burel. A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo. In *Proof Exchange for Theorem Proving (PxTP '13), Third International Workshop*, 2013.

[Cau] R. Cauderlier. Focalide. `https://www.rocq.inria.fr/deducteam/Focalide/index.html`.

[CD07] Denis Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *Typed Lambda Calculi and Applications (TLCA), 8th International Conference, Proceedings*, 2007.

[CD15] R. Cauderlier and C. Dubois. Objects and Subtyping in the $\lambda\Pi$-Calculus Modulo. Submitted, 2015.

[CDT] The Coq Development Team. The Coq Reference Manual. `http://coq.inria.fr`.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958.

[CGN01] H. Comon, G. Godoy, and R. Nieuwenhuis. The Confluence of Ground Term Rewrite Systems is Decidable in Polynomial Time. In *Foundations of Computer Science (FOCS), 42nd Annual Symposium*, 2001.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 1988.

[CHJ+12] M. Codescu, F. Horozal, A. Jakubauskas, T. Mossakowski, and F. Rabe. Compiling Logics. In *Recent Trends in Algebraic Development Techniques (WADT), 21st International Workshop, Revised Selected Papers*, 2012.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 1936.

[CWC07] J. Chrząszcz and D. Walukiewicz-Chrząszcz. Towards Rewriting in Coq. In *Rewriting, Computation and Proof*. Springer Berlin Heidelberg, 2007.

[DDG+13] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2013.

[DHK03] G. Dowek, T. Hardin, and C. Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning*, 2003.

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 1979.

[Dor11] A. Dorra. Equivalence de Curry-Howard entre le lambda-Pi calcul et la logique intuitionniste. Internship Report, 2011.

[Dou92] D. J. Dougherty. Adding Algebraic Rewriting to the Untyped Lambda Calculus. *Information and Computation*, 1992.

[Dow01] G. Dowek. Higher-Order Unification and Matching. In *Handbook of Automated Reasoning (in 2 volumes)*. MIT Press, 2001.

[Dow15] G. Dowek. Models and termination of proof-reduction in the lambda-Pi-calculus modulo theory. Draft, 2015.

[DW05] G. Dowek and B. Werner. Arithmetic as a Theory Modulo. In *Term Rewriting and Applications (RTA), 16th International Conference, Proceedings*, 2005.

[Gen35] G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 1935.

[Geu92] H. Geuvers. The Church-Rosser Property for beta-eta-reduction in Typed lambda-Calculi. In *Logic in Computer Science (LICS), Seventh Annual Symposium, Proceedings*, 1992.

[Har09] J. Harrison. HOL Light: An Overview. In *Theorem Proving in Higher Order Logics (TPHOLs), 22nd International Conference, Proceedings*, 2009.

[HHP93] R. Harper, F. Honsell, and G. D. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 1993.

[HPWD] T. Hardin, F. Pessaux, P. Weis, and D. Doligez. FoCaLiZe Reference Manual. `http://focalize.inria.fr`.

[Hue80] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 1980.

[Hur11] J. Hurd. The OpenTheory Standard Theory Library. In *NASA Formal Methods (NFM), Third International Symposium, Proceedings*, 2011.

[JO91] J. P. Jouannaud and M. Okada. A Computation Model for Executable Higher-Order Algebraic Specification Languages. In *Logic in Computer Science (LICS), Sixth Annual Symposium, Proceedings*, 1991.

[JR99] J. P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Logic in Computer Science (LICS), 14th Annual Symposium*, 1999.

[KB83] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In *Automation of Reasoning*. Springer Berlin Heidelberg, 1983.

[Klo80] J. W. Klop. *Combinatory reduction systems*. PhD thesis, Univ. Utrecht, 1980.

[Kor08] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Automated Reasoning (IJCAR), 4th International Joint Conference, Proceedings*, 2008.

[Kre] C. Kreitz. The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide. `http://www.nuprl.org/html/nuprl5docs.html`.

[Mar08] L. Maranget. Compiling Pattern Matching to Good Decision Trees. In *Proceedings of the ACM Workshop on ML*, 2008.

[Mat67] Y. Matijasevich. Simple Examples of Undecidable Associative Calculi. *Doklady Mathematics*, 1967.

[Mil91] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1991.

[Mül92] F. Müller. Confluence of the Lambda Calculus with Left-Linear Algebraic Rewriting. *Information Processing Letters*, 1992.

[New42] M. H. A. Newman. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics*, 1942.

[NGdV94] R. P. Nederpelt, H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath*. Elsevier Science, 1994.

[Nip91] T. Nipkow. Higher-Order Critical Pairs. In *Logic in Computer Science (LICS), Sixth Annual Symposium, Proceedings*, 1991.

[Nor07] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *ACM Transactions on Computational Logic*, 2008.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Martin-Löf's Type Theory*. Oxford University Press, 1990.

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[Oka89] M. Okada. Strong Normalizability for the Combined System of the Typed lambda Calculus and an Arbitrary Convergent Term Rewrite System. In *Symbolic and Algebraic Computation (ISSAC), International Symposium, Proceedings*, 1989.

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Automated Deduction (CADE)*. Springer, 1992.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, 1988.

[Pie08] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Principles of Programming Languages (POPL)*, 2008.

[Pie10] B. Pientka. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *Functional and Logic Programming*. Springer Berlin Heidelberg, 2010.

[PS99] F. Pfenning and C. Schürmann. System Description: Twelf - a Meta-logical Framework for Deductive Systems. In *Automated Deduction (CADE)*. Springer, 1999.

[Ros73] B. K. Rosen. Tree-Manipulating Systems and Church-Rosser Theorems. *Journal of the ACM*, 1973.

[SH12] V. Siles and H. Herbelin. Pure Type System conversion is always typable. *Journal of Functional Programming*, 2012.

[Tan88] V. Tannen. Combining Algebra and Higher-Order Types. In *Logic in Computer Science (LICS), Third Annual Symposium, Proceedings*, 1988.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[TG89] V. Tannen and J. H. Gallier. Polymorphic Rewriting Conserves Algebraic Strong Normalization and Confluence. In *Automata, Languages and Programming (ICALP), 16th International Colloquium, Proceedings*, 1989.

[Toy87] Y. Toyama. On the Church-Rosser Property for the Direct Sum of Term Rewriting Systems. *Journal of the ACM*, 1987.

[vO95] V. van Oostrom. Development Closed Critical Pairs. In *Higher-Order Algebra, Logic, and Term Rewriting (HOA), Second International Workshop, Selected Papers*, 1995.

[Wal03] D. Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. *Journal of Functional Programming*, 2003.

Index

Vérification de typage pour le $\lambda\Pi$-Calcul Modulo : théorie et pratique

Résumé : la vérification automatique de preuves consiste à faire vérifier par un ordinateur la validité de démonstrations d'énoncés mathématiques. Cette vérification étant purement calculatoire, elle offre un haut degré de confiance. Elle est donc particulièrement utile pour vérifier qu'un logiciel critique, c'est-à-dire dont le bon fonctionnement a un impact important sur la sécurité ou la vie des personnes, des entreprises ou des biens, correspond exactement à sa spécification. Dedukti est l'un de ces vérificateurs de preuves. Il implémente un système de types, le $\lambda\Pi$-Calcul Modulo, qui est une extension du λ-calcul avec types dépendants avec des règles de réécriture du premier ordre. Suivant la correspondance de Curry-Howard, Dedukti implémente à la fois un puissant langage de programmation et un système logique très expressif. Par ailleurs ce langage est particulièrement bien adapté à l'encodage d'autres systèmes logiques. On peut, par exemple, importer dans Dedukti des théorèmes prouvés en utilisant d'autres outils tels que *Coq*, *HOL* ou encore *Zenon*, ouvrant ainsi la voie à l'interopérabilité entre tous ces systèmes.

Le $\lambda\Pi$-Calcul Modulo est un langage très expressif. En contrepartie, certaines propriétés fondamentales du système, telles que l'unicité des types ou la stabilité du typage par réduction, ne sont pas garanties dans le cas général et dépendent des règles de réécriture considérées. Or ces propriétés sont nécessaires pour garantir la cohérence des systèmes de preuve utilisés, mais aussi pour prouver la correction et la complétude des algorithmes de vérification de types implémentés par Dedukti. Malheureusement, ces propriétés sont indécidables. Dans cette thèse, nous avons donc cherché à concevoir des critères garantissant la stabilité du typage par réduction et l'unicité des types qui soient décidables, de manière à pouvoir être implémentés par Dedukti.

Pour cela, nous donnons une nouvelle définition du $\lambda\Pi$-Calcul Modulo qui rend compte de l'aspect itératif de l'ajout des règles de réécriture dans le système en les explicitant dans le contexte. Une étude détaillée de ce nouveau calcul permet de comprendre qu'on peut ramener le problème de la stabilité du typage par réduction et de l'unicité des types à deux propriétés plus simples qui sont la compatibilité du produit et le bon typage des règles de réécriture. Nous étudions donc ces deux propriétés séparément et en donnons des conditions suffisantes effectives.

Ces idées ont été implémentées dans Dedukti, permettant d'augmenter grandement sa fiabilité.

Mots clés : théorie de la preuve, théorie des types, méthodes formelles

Typechecking in the $\lambda\Pi$-Calculus Modulo: Theory and Practice

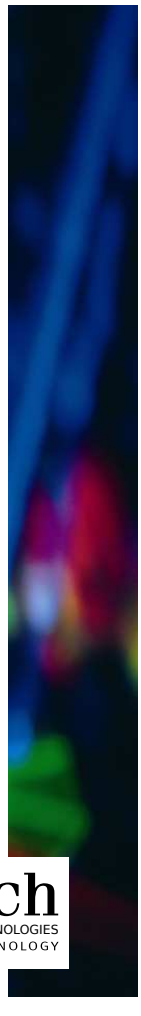
Abstract: Automatic proof checking is about using a computer to check the validity of proofs of mathematical statements. Since this verification is purely computational, it offers a high degree of confidence. Therefore, it is particularly useful for checking that a critical software, i.e., a software that when malfunctioning may result in death or serious injury to people, loss or severe damage to equipment or environmental harm, corresponds to its specification. Dedukti is such a proof-checker. It implements a type system, the $\lambda\Pi$-Calculus Modulo, that is an extension of the dependently typed λ-calculus with first-order rewrite rules. Through the Curry-Howard correspondence, Dedukti implements both a powerful programming language and an expressive logical system. Furthermore, this language is particularly well suited for encoding other proof systems. For instance, we can import in Dedukti theorems proved using other tools such as *Coq*, *HOL* or *Zenon*, initiating a form of interoperability between these systems.

The $\lambda\Pi$-Calculus Modulo is a very expressive language. On the other hand, some fundamental properties such as subject reduction (that is, stability of typing by reduction) and uniqueness of types are not guaranteed in general and depend on the rewrite rules considered. Yet, these properties are necessary for guarantying the coherence of the proof system, but also for proving the soundness and completeness of the type-checking algorithms implemented in Dedukti. Unfortunately, these properties are undecidable. In this thesis we design new criteria for subject reduction and uniqueness of types that are decidable, in order to be implemented in Dedukti.

For this purpose, we give a new definition of the $\lambda\Pi$-Calculus Modulo that takes into account the iterative aspect of the addition of rewrite rules in the typing context. A detailed study of this new system shows that the problems of subject reduction and uniqueness of types can be reduced to two simpler properties that are called product compatibility and well-typedness of rewrite rules. Hence, we study these two properties separately and we give effective sufficient conditions for them to hold.

These ideas have been implemented in Dedukti increasing its reliability.

Keywords: proof theory, type theory, formal methods



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY