Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Terminaison à base de tailles: Sémantique et généralisations

THÈSE

présentée et soutenue publiquement le 20 Avril 2011

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Cody Roux

Composition du jury

Rapporteurs :	Jürgen Giesl Gilles Barthe
Examinateurs :	Claude Kirchner (directeur) Frédéric Blanqui (co-encadrant) Gilles Dowek Andreas Abel Adam Cichon

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Le travail de thèse est un chemin long et parfois difficile, et mon chemin n'aurait pas pu se faire sans l'aide précieuse des personnes qui m'ont entouré. D'abord celle de Frédéric Blanqui, dont l'impressionnante habileté technique et l'amour de la rigueur ont grandement contribué à mes réalisations techniques. Frédéric s'est révélé une source quasi-inépuisable d'observations techniques et de contre-exemples subtils, et a su donner certaines idées importantes du coeur de ce manuscrit. Vient ensuite celle de Claude Kirchner, dont le soutien moral s'est révélé précieux, ainsi que les conseils scientifiques et "humains". Claude a su supporter mes explications parfois confuses sur mon travail avec une patience infinie. Gilles Dowek a très généreusement accepté d'assister dans mon encadrement malgré son emploi du temps très chargé, et ses conseils en Réécriture, en Logique, en Sciences et en Philosophie qui me serviront bien au-dela de mon travail de thèse.

C'est Andreas Abel qui a le plus contribué, je pense, au peu que je comprend en théorie des types, et je lui en suis très reconnaissant. Il a eu la gentillesse de m'accueillir à deux reprises à Munich pour des séjours qui ont défini une partie de la direction de cette thèse. Colin Riba n'arrive pas loin derrière, et j'admirerai toujours ses points de vue très orginaux sur des sujets classiques.

Je remercie Jurgen Giesl et Gilles Barthe d'avoir pris le temps et l'énergie d'être rapporteurs de mon manuscrit, et d'avoir donné leur point de vue subtil à mes modestes travaux. Je remercie Adam Cichon d'avoir accepté de présider mon Jury.

Je remercie Makoto Hamana d'avoir pris le temps de donner des explications sur son travail et pour les discussions importantes qui s'en suivirent.

Une thèse se produit dans un environnement particulier, et je remercie l'équipe Pareo (anciennement Protheo) et en particulier Pierre-Etienne Moreau, de m'en avoir fourni un particulièrement chaleureux. Je remercie mes co-bureaux, Paul, Clément et Claudia, de m'avoir permis de travailler dans un cadre joyeux et stimulant. Je remercie les étudiants de l'université de Tsinghua de leur accueil, particulièrement Hehua et Dr. Lee. De même je remercie les thésards de l'équipe Marelle de Sophia-Antipolis, et en particulier Ioana pour son support.

Je remercie les membres de l'ARC Corias de m'avoir permis de me rendre à Paris et pour de nombreuses discussions très stimulantes, et Germain pour ses pouvoirs d'organisation. De même je remercie les membres du projet Mocca pour le travail intéressant qui s'est déroulé pendant ce projet.

Je remercie l'equipe Proval de m'avoir accueilli avant d'avoir soutenu ma thèse et particulièrement Guillaume Melquiond et Xavier Urbain pour m'avoir conseillé pour ma présentation.

Je remercie mes parents, qui ont su me donner compréhension et soutien tout au long de ce travail.

Enfin je remercie Camille, pour être ce qu'elle est, et je lui dédicace ce travail.

"Why is it hard to imagine eternity?" demanded Natasha. "After today comes tomorrow, and then the next day, and so on for ever; and there was yesterday, and the day before..."

— *War and Peace*, L. TOLSTOY, *1869*

Sommaire

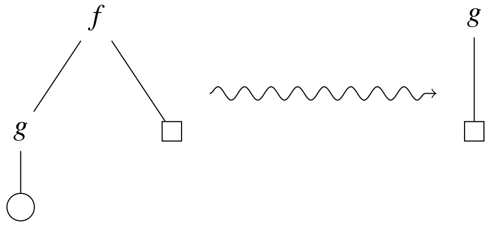
Partie II A Type-Based Dependency Analysis

Introduction

Il n'est pas exagéré de prétendre que la compréhension du comportement réel du code source des programmes qui tournent sur les machines numériques est un défi majeur du XXIè siècle. Les promesses remplies de la puissance de calcul apportée par la micro-électronique sont contrecarrées par l'extrême complexité du raisonnement formel sur les programmes. Le raisonnement informel sur les programmes structurés ne permet pas d'éviter en général les comportements pathologiques. Reste alors le raisonnement formel, basé sur une formulation mathématiquement précise du comportement logiciel. Cependant, même l'approche formelle souffre de limitations: il est difficile de la mettre en pratique sur des programmes réalistes, c'est *l'explosion combinatoire* des états possibles des programmes, qui est une conséquence de l'*indécidabilité* de la preuve de nombreuses propriétés concernant l'exécution d'un algorithme. La terminaison est un sous-problème du problème plus général de la correction d'un programme, mais il est significatif, car la terminaison est en général *nécessaire* à la correction du programme (en général nous désirons que le programme *retourne une valeur*, avant de *terminer*), et elle nécessite une analyse particulière.

La terminaison est le type quintessentiel du problème indécidable. En effet, c'est le premier problème, avec celui de décider si un énoncé de l'arithmétique est prouvable, pour lequel l'indécidabilité fut prouvée par Alan Turing [Tur36] et Alonzo Church [Chu36]. En outre, la terminaison peut être vue comme un problème "très indécidable": la preuve d'indécidabilité, en plus d'être simple, est confortée par de nombreux problèmes d'énoncé simple dont on ne sait pas, aujourd'hui, s'ils terminent ou non. Par exemple, il est facile de construire un programme dont une preuve de terminaison réfuterait la conjecture de Goldbach: il suffit d'énumérer les entiers pairs, et d'arrêter si l'entier considéré n'est pas somme de deux nombres premiers inférieurs. Cela suggère que la terminaison est essentiellement un problème *sémantique*, c'est-à-dire qui dépend du *sens* du programme, ou encore de son comportement *voulu*.

Cette limitation n'a pas empêché le développement d'analyses de terminaison, particulièrement après l'avènement de l'âge informatique, où se pose en particulier le problème bien réel de la correction des centaines de milliers de lignes de code utilisées dans l'industrie. En pratique, de nombreux critères de terminaison, nécessairement incomplets, existent pour divers langages. Nous nous intéressons à un modèle de calcul particulier, la *réécriture*. La réécriture est un paradigme de calcul calculatoirement complet (voir par exemple Post [Pos47], Dershowitz et Jouannaud [DJ90], et Davis [Dav85]), qui modélise une grande classe de calculs. En particulier, la réécriture précise l'aspect calculatoire du raisonnement équationnel, et en conséquence est très utile pour exprimer *déclarativement* un calcul: il suffit de spécifier les *équations* que satisfont l'objet à calculer. La réécriture du *premier ordre* se concentre sur des arbres annotés par des symboles de fonction, avec à leur feuille des *variables* qui sont destinées à être *instanciées*. Par exemple la règle $f(g(x), y) \to g(y)$ représente la transformation



Où ○ et □ sont des arbres arbitraires. On pourra se référer à Terese [BKdV03] ou Baader et Nipkow [BN98] pour une introduction complète.

Le *λ-calcul* ajoute de la puissance à ce formalisme en réifiant le concept de substitution: on ajoute les constructions d'*abstraction*, représentée par $\lambda x.t$ et l'*application*, représentée par la juxtaposition. Le terme $\lambda x.t\ u$ représente alors le terme t dans lequel toutes les occurrences (libres) de la variable x sont remplacés par le terme u. La simple présence de ces constructions suffit à donner un calcul complet (voir Church [Chu41]), mais la combinaison de ce calcul et de la réécriture donne un système suffisamment expressif pour servir de base à un langage de programmation. Les langages ML [MTHM97, CM98] et Haskell [Fax02] ont à leur cœur un λ-calcul avec de la réécriture avec la restriction que le filtrage se fait sur des termes non-définis, et que les règles doivent être orthogonales.

Il existe de nombreux critères de terminaison pour la réécriture. Pour la réécriture du premier ordre, il est commun de distinguer deux types de critères: les critères syntaxiques et les critères sémantiques. Les critères syntaxiques sont avant tout destinés à être décidables et à faire l'objet d'un *algorithme*, et les objets mathématiques considérés sont l'ensemble des termes et le système de réécriture. Un exemple important de ce genre de critère est donné par les *ordres de simplification*, et en particulier le *recursive path ordering* (RPO) [Der82], qui est utilisé dans la pratique. D'un autre côté, les critères sémantiques cherchent en général à être *complets*, c'est-à-dire que tout système de réécriture normalisant passe le critère. La structure mathématique considérée est une abstraction du système de réécriture, sa *sémantique*. L'exemple prototypique est le critère de *Manna-Ness* [MN70], qui considère un ensemble X muni d'un ordre bien fondé $>_X$, avec une fonction d'interprétation $(\!|\_|\!)$ des termes vers X qui vérifie $t \to^+ u \Rightarrow (\!|t|\!) >_X (\!|u|\!)$. Il est facile de montrer qu'un tel critère est complet, en prenant le modèle "syntaxique", dans lequel l'interprétation d'un terme est le terme lui-même.

En ce qui concerne le λ-calcul, le calcul pur n'est pas normalisant, et il convient donc de ne considérer qu'une classe de termes normalisants. Une classe très importante de tels termes sont les *termes simplement typés*: on construit une fonction de typage, qui permet d'associer des types de la forme $B, C, \ldots$ ou $a \to b$ a certains λ-termes, ou $B, C, \ldots$ sont des *constantes de type*. On peut ensuite montrer (voir par exemple [GLT89]) que chaque terme qui admet un type simple est fortement normalisant. Bien qu'apparemment syntaxique de nature, le typage a une contrepartie sémantique: il est tentant d'interpréter chaque constante de type comme un ensemble, et $a \to b$ comme un ensemble de procédures qui, à un élément de l'interprétation de a, associe un élément de l'interprétation de b. La preuve de terminaison de Tait [Tai67], utilise cette intuition, en interprétant chaque type en un ensemble de termes normalisants. Ce point de vue est conforté par les développements en logique mathématique visant à établir une correspondance entre théorie des types et théorie des systèmes logiques: c'est la *correspondance de Curry-Howard-DeBruijn* (on pourra se référer à De Bruijn [dB95] pour une présentation générale). Dans le cadre de cette correspondance, un λ-terme typé est la preuve d'une proposition que représente son type. L'existence d'une forme normale correspond alors à l'existence d'une preuve sans coupure. Cette notion étant fortement liée à la consistance du système, il faut en général construire un *modèle* pour la montrer. C'est donc bien un point de vue sémantique.

Évidemment, ce point de vue est un peu caricaturé, et il n'existe pas de manière objective de distinguer un critère syntaxique d'un critère sémantique. Néanmoins, une approche puissante pour donner un critère décidable est de prendre un critère sémantique et d'en donner une représentation syntaxique, qui se prête à la manipulation par les programmes. Donnons trois exemples.

Le premier concerne l'utilisation de *types annotés* pour déduire certaines contraintes concernant la taille des formes normales des arguments de fonctions définies. Introduits par Hughes *et al* [HPS96], et indépendemment par Giménez [Gim96], ils ont étés étudiés notamment par Abel [Abe06], Frade [Fra03], Barthe *et al* [BFG⁺04] et Blanqui [Bla04] pour différents systèmes de types, allant des types simples au Calcul des Constructions de Coquand et Huet [CH88]. L'avantage de la méthode par typage est sa puissance vis-à-vis des précédents critères syntaxiques, par exemple le *schéma général* de Blanqui, Jouannaud et Okada [BJO02, Bla05b, BJO99] ou la *condition de garde* décrite par Giménez et Amadio *et al* [Gim95, ACG98], se basant en partie sur Coquand [Coq93]. La terminaison à base de types permet de montrer la normalisation en présence de certains systèmes qui ne sont pas simplement normalisants, et permet surtout de s'affranchir d'une certaine sensibilité à la forme syntaxique des définitions. Une extension de ce critère pour les types simples et les types de données du premier ordre est donnée par Blanqui et Riba [BR06], et une version similaire de Xi [Xi01], qui permet d'ajouter des prédicats de l'*arithmétique de Presburger* [Pre29] pour décrire les annotations de tailles. Ceci augmente grandement la puissance du critère, car on dispose alors de toute l'arithmétique de Presburger pour exprimer la sémantique d'un symbole de fonction définie.

Le second critère prend un modèle équationnel d'un système de réécriture du premier ordre et utilise ce modèle pour construire un système *annoté*, dont la terminaison est équivalente à celle du système originel. Cette méthode porte le nom de méthode des *annotations sémantiques*, et fut introduite par Zantema [Zan95]. Une extension naturelle du critère consiste à remplacer les modèles par des prémodèles, et le critère devient alors complet. Cette méthode de preuve de normalisation forte a récemment été étendue aux systèmes de réécriture d'ordre supérieurs par Hamana [Ham07], en utilisant des prémodèles issus de la vision catégorique d'algèbres proposée par Fiore *et al* [FPT99], qui utilise des constructions sur des préfaisceaux.

Le dernier critère est celui des paires de dépendances, décrit en premier par Arts et Giesl [AG00]. Il est très important pour la réécriture au premier ordre, car il permet d'être utilisé en conjonction avec d'autres critères: c'est une *approche* [GAO02, GTSK05]. Le but est de montrer qu'il n'y a pas de *chaînes de dépendances* infinies, ou les chaînes de dépendances dénotent une séquence d'appels de fonctions. La technique est donc plutôt syntaxique, et elle est utilisée en pratique par les logiciels d'analyse de terminaison, mais fait en général usage d'un *graphe de dépendances*, qui est une abstraction des successions possibles de paires de dépendances dans une chaîne.

Cette thèse tente de donner une vision générale de ces trois critères sémantiques pour une forme de réécriture d'ordre supérieure, la réécriture *algébrique à gauche.* Dans un premier temps, on montre qu'une certaine forme de critère de terminaison à base de tailles est subsumée par le critère des annotations sémantiques. On décrit un critère d'annotations sémantiques inspiré de celui de Hamana. Nous montrons qu'il est possible, étant donné un modèle du système de réécriture augmenté de la β-réduction, de décrire un système de réécriture sur des termes qui portent des annotations de la sémantique. Ensuite nous montrons qu'en rajoutant des *règles structurelles*, le système annoté simule le système original, avec β-réduction. Malheureusement, le système annoté n'est jamais normalisant, même si le système original l'est! Nous contournons le problème en montrant que si le système annoté vérifie une propriété plus faible, la *normalisation relative* par rapport aux règles structurelles, alors le système original est fortement normalisant.

Ensuite nous construisons un modèle qui capture l'intuition sémantique sous-jacente du système de réécriture. En effet, ce modèle permet d'interpréter les types de données inductives par des ensembles cumulatifs, dont les éléments ont une notion naturelle de *rang ordinal*. Les types fonctionnels sont interprétés par des "vraies" fonctions. Pour limiter la taille de ces espaces, nous utilisons la notion de *réalisabilité*.

Pour montrer la terminaison relative du système annoté, on utilise un critère simple de précédence, adaptée du *schéma général* [Bla03]. Ce critère est suffisamment général pour prouver la terminaison de différents systèmes annotés, et il contient toute l'information combinatoire nécessaire à la preuve de normalisation forte. Nous affirmons qu'il peut donc être utilisé comme une "boite noire" qui permet de montrer la terminaison d'un système pour lequel nous avons une sémantique suffisamment précise du système avec règles algébriques. C'est donc en un sens un résultat de modularité.

Cette approche par les prémodèles ne permet cependant pas de capturer un certain aspect de la terminaison des systèmes avec β-réduction: la *non-localité*. En effet, les fonctions définies à l'intérieur d'un terme ne disposent pas localement de toute l'information sur ses arguments: certaines variables peuvent être *instanciées* par une β-réduction.

Nous montrons qu'une variation des types annotés utilisés pour la terminaison à base de taille, les *types raffinés*, peut être utilisée pour capturer cette information non-locale. Ceci nous permet de construire un analogue du graphe de dépendances approximé dans les types. Le langage étudié est un langage d'ordre supérieur avec β-réduction et réécriture. La sémantique opérationnelle porte sur des termes dans lesquels les annotations de types ont étés effacées, ce qui permet d'avoir toute l'information nécessaire au typage dans les termes, tout en évitant les réductions "administratives".

Nous montrons qu'il existe une analyse syntaxique du graphe de dépendance approximé qui est tout à fait analogue à une analyse du premier ordre sur le graphe de dépendance approximé, la méthodes des *projections simples*, et qui permet de donner une condition suffisante à la terminaison des termes bien typés. Ceci permet d'opérer un rapprochement entre les méthodes issues de la théorie de la réécriture et celles utilisées en théorie des types. Nous comparons la puissance de cette analyse avec d'autres approches pour les paires de dépendance à l'ordre supérieur.

Ce document est disposé comme suit:

- Nous donnons d'abord un chapitre qui introduit les notations et concepts de base de la réécriture d'ordre supérieure considérée dans cette thèse. Nous redémontrons le théorème bien connu de la normalisation forte du λ-calcul simplement typé, pour donner une introduction aux concepts de calculabilité utilisés par la suite.

- Nous abordons ensuite une première partie consacrée à l'application des annotations sémantiques pour la preuve d'un critère de terminaison à base de types annotés.

  – Le premier chapitre est consacré à la description du critère de terminaison à base de tailles dont la preuve de correction est l'objet des chapitres suivants, avec des exemples de systèmes qui passent (ou non) le critère.

  - Dans le second chapitre nous introduisons le concept d'algèbre d'ordre supérieure, ainsi que celle d'objet monoïdal dans la catégorie des $\bullet$-monoïdes qui nous permet de donner la définition de prémodèle d'un système de réécriture. Les notions de théorie des catégories nécessaires seront rappelées.
  - Dans le troisième chapitre, nous utilisons la sémantique définie dans le chapitre précédent pour décrire la méthode des annotations sémantiques et donner le théorème fondamental: le système annoté termine relativement à certaines règles structurelles si et seulement si le système original termine.
  - Ensuite nous décrivons comment construire un modèle dans lequel les types de base sont interprétés par une description ensembliste des types de données qu'ils représentent, et les fonctions sont représentés par des fonctions *réalisées* par un certain terme, dans le cas où le système de réécriture satisfait les conditions du théorème de terminaison donné au chapitre 1. Nous suggérons que ce modèle correspond à l'intuition de la sémantique du système de type de ce premier chapitre.
  - Dans le cinquième chapitre nous montrons qu'il existe un critère de terminaison relative général qui permet de montrer la terminaison (relativement aux règles structurelles) de systèmes annotés. Ce critère se base sur une précédence sur les symboles de fonction définis, qui doit être *compatible* avec les règles structurelles. Nous montrons qu'un système qui passe le critère du chapitre 2, annoté avec la sémantique du chapitre 5 passe le critère. Pour montrer la compatibilité avec les règles structurelles nous avons besoin d'un lemme combinatoire.

    Nous donnons ensuite une application supplémentaire de cette approche pour montrer un théorème de modularité de Breazu-Tannen, Gallier & Okada [Oka89, GBT89].

- La seconde partie rend compte d'une approche par les types, similaire à la terminaison à base de tailles, mais qui permet une approximation plus précise de la succession d'appels de fonctions.

  - Le premier chapitre de cette partie décrit la théorie classique des paires de dépendance pour la réécriture du premier ordre, pour fixer les notations et les concepts, et contraster avec la situation à l'ordre supérieur.
  - Le second chapitre décrit un système de types dépendants, que nous appelons *types raffinés*, qui permet, étant donné un système de réécriture bien typé, de donner un ensemble de paires de dépendances et un graphe de dépendances approximé. Ensuite nous décrivons un critère syntaxique sur le graphe, qui correspond au critère de *simple projection* dans le cadre standard des paires de dépendances, et énonçons le théorème principal de cette partie: si le système de réécriture passe le critère, tout terme bien typé est fortement normalisant sous ce système et la β-réduction.
  - Le troisième chapitre détaille la preuve du théorème principal. Nous appliquons les techniques de calculabilité utilisées dans les sections précédentes. Un soin particulier doit être pris pour traiter le non-déterminisme inhérent de la réécriture. Celui-ci nous pousse à considérer des suites d'*ensembles de formes normales* de termes, auxquelles nous appliquons le fameux *lemme de König.*

- Nous décrivons les approches à la terminaison les plus liées à notre approche, et nous expliquons comment notre approche se situe par rapport à ces efforts. Nous donnons ensuite des perspectives concernant l'unification et l'extension des critères proposés.

Nos contributions sont les suivantes:

- Nous montrons que la méthode des annotations sémantiques de Hamana [Ham07] peut être modifiée afin que la β-réduction soit préservée en tant que règle de calcul dans le système annoté. Pour cela, il est nécessaire d'introduire des règles structurelles, et de considérer la terminaison relativement à ces règles et non plus la terminaison tout court. Nous montrons la correction de cette approche.

- Nous donnons un modèle de réalisabilité pour les systèmes de réécriture qui satisfont le critère de taille, qui permet d'interpréter les termes de manière intuitive. Un terme en forme normale est dénoté par un tuple qui le représente en tant qu'arbre. Les abstractions sont interprétées par des fonctions dans l'ensemble des fonctions réalisées, et nous montrons que les éléments de cette sémantique ont un ordinal naturellement associé, le *rang*, qui nous sert à définir les interprétations de fonctions définies par induction bien fondée, ainsi que montrer la terminaison.

- Nous montrons que la méthode d'annotations sémantiques peut s'appliquer aux systèmes qui passent le critère de taille. Comme les règles structurelles permettent alternativement un affaiblissement et une instantiation du contexte dans lequel sont interprétés les termes, il est nécessaire, pour montrer la bonne fondaison d'un ordre sur ces interprétations, d'utiliser un lemme combinatoire, du a Doornbos et von Karger pour montrer la stabilité de cet ordre vis-à-vis de ces modifications du contexte.

- Nous donnons une nouvelle preuve de modularité de la terminaison entre la réécriture du premier ordre et le λ-calcul simplement typé.

- Nous décrivons un système de types pour un langage d'ordre supérieur basé sur la réécriture, qui permet de décrire un graphe de dépendances des appels de fonctions. L'examen des cycles dans ce graphe permet d'identifier les sources de non-terminaison potentielles. Nous donnons un critère syntaxique sur ce graphe, basé sur l'ordre sous terme, qui permet de garantir la terminaison des termes bien typés. Le langage comporte des annotations de types qui facilitent l'inférence, mais la sémantique opérationnelle est donnée sur les termes avec les annotations effacées. Nous prouvons que ce critère est correct, c'est-à-dire que pour tout système qui passe ce critère, et tout terme bien typé, l'effacement de ce terme est fortement normalisant sous le système de réécriture et β-réduction.

Définitions

Commençons par dire un mot sur la méta-théorie. Nous adoptons la convention, habituelle dans les développements non-formels, de ne pas nous étendre sur celle-ci; en particulier, nous ne détaillerons pas les axiomes précis utilisés. Nous supposerons simplement que nous pouvons sans peine construire des définitions et preuves inductives, des espaces de fonctions, etc.

Ceci dit, nous supposerons que nous sommes placés dans un cadre méta-théorique proche de la théorie de Zermelo-Fraenkel avec axiome du Choix [FBHL73] (on pourra également se référer a Jech [Jec06]), ce qui permet d'utiliser le traitement habituel des ordinaux, par exemple [CL03]. Nous utilisons en particulier l'axiome du choix à certains endroits, en évitant la question du système d'axiomes minimal nécessaire pour formaliser le résultat.

Cette thèse traite de terminaison de systèmes de réécriture d'ordre supérieurs. Nous travaillerons dans des systèmes qui sont des extensions du λ-calcul avec une signature de fonctions. Les définitions et lemmes peuvent être trouvés dans Terese [BKdV03], mais nous rappelons les bases ici.

1 Relations et termes

La réécriture est un modèle de calcul basé sur la notion de *réduction*. Mathématiquement, nous modéliserons la réduction par une relation.

Definition 1 Soit E un ensemble quelconque, et $R \subseteq E \times E$ une relation sur E. Nous noterons xRy si $(x, y) \in R$. On dit que R est un *préordre* si R est réflexive et transitive. On dit que R est un *ordre* si R est un préordre et antisymétrique.

La *clôture transitive* R^+ de R est la plus petite relation telle que:

- $xRy \Rightarrow xR^+y$
- $xR^+y \wedge yR^+z \Rightarrow xR^+z$

La *clôture reflexive transitive* R^* de R est la plus petite relation telle que:

- $xR^+y \Rightarrow xR^*y$
- xR^*x

La *clôture symétrique réflexive transitive* $\equiv_R$ de R est la plus petite relation telle que:

- $xR^*y \Rightarrow x \equiv_R y$
- $x \equiv_R y \Rightarrow y \equiv_R x$

La *composée* $R \circ S$ de deux relations R et S est la relation définie par:

$$xR \circ S y \Leftrightarrow \exists z, xRz \wedge zS y$$

Étant donné un ensemble X, et une relation R, le *quotient* $X/\equiv_R$ de X par $\equiv_R$ est défini par:

$$X/\equiv_R := \{\overline{x} \mid x \in X\}$$

avec $\overline{x}$ la *classe d'équivalence* de x définie par

$$\overline{x} = \{y \mid x \equiv_R y\}$$

Nous ne travaillons pas sur des ensembles E arbitraires. Nous voulons considérer des ensembles d'éléments qui modélisent correctement un calcul a effectuer.

Definition 2 Donnons un ensemble Σ que l'on appelle *Signature*, et un ensemble $\mathcal{X}$ infini appelé *ensemble de variables*. L'ensemble des λ-termes sur la signature Σ est l'ensemble défini par la forme de Backus-Naur (BNF) suivante:

$$t, u \in \mathcal{T}rm^{\Sigma} := x \mid f \mid t\ u \mid \lambda x.t$$

avec $x \in \mathcal{X}$ et $f \in \Sigma$.

Nous adopterons les conventions de notation suivantes: l'application est associative à droite, donc $(t\ u)\ v$ s'écrit $t\ u\ v$ et nous adopterons la notation vectorielle pour une série d'applications, $t\ u_1 \ldots u_n$ se note $t\ \vec{u}$. On notera $\mathcal{T}rm$ au lieu de $\mathcal{T}rm^{\Sigma}$ si il n'y a pas d'ambiguïté.

Définissons maintenant le *renommage des variables*:

Definition 3 Soit t un terme et x, y deux variables. Le *renommage de x par y dans t*, noté $t\{y/x\}$ est défini par induction sur t:

- $x\{y/x\} = y$
- $z\{y/x\} = z$ si $z \in \mathcal{X}$ et $z \neq x$
- $f\{y/x\} = f$ pour $f \in \Sigma$
- $t\ u\{y/x\} = t\{y/x\}\ u\{y/x\}$
- $(\lambda z.t)\{y/x\} = \lambda z.(t\{y/x\})$ pour tout $z \in \mathcal{X}$

La présence de termes comme $\lambda x.x$ nous oblige à définir le concept de *variable liée*:

Definition 4 On définit l'ensemble des variables *libres* et *liées* d'un terme $t \in \mathcal{T}rm$ dénotés respectivement $\mathcal{FV}(t)$ et $\mathcal{BV}(t)$ par induction sur t:

- $\forall x \in \mathcal{X}, \mathcal{FV}(x) = \{x\}$
- $\forall f \in \Sigma, \mathcal{FV}(f) = \mathcal{BV}(f) = \emptyset$
- $\mathcal{FV}(t\ u) = \mathcal{FV}(t) \cup \mathcal{FV}(u)$, $\mathcal{BV}(t\ u) = \mathcal{BV}(t) \cup \mathcal{BV}(u)$
- $\mathcal{FV}(\lambda x.t) = \mathcal{FV}(t) \backslash \{x\}$, $\mathcal{BV}(\lambda x.t) = \mathcal{BV}(t) \cup \{x\}$

Nous voulons considérer les termes *modulo* renommage des variables liées: nous voulons moralement avoir:

$$\forall t \in \mathcal{T}rm, \forall y \notin \mathcal{FV}(t), \lambda x.t = \lambda y.(t\{y/x\})$$

Il est possible de garantir cette égalité en travaillant sur un certain quotient de l'ensemble $\mathcal{T}rm$. Cependant travailler explicitement avec cet ensemble quotient est difficile en pratique. Nous adopterons plutôt la *convention de Barendregt* [Bar84]: étant donné un terme de la forme $\lambda x.t$ et un ensemble fini $\mathcal{E}$ de variables, on peut toujours supposer $x \notin \mathcal{E}$, quitte à remplacer $\lambda x.t$ par $\lambda y.t\{y/x\}$. En particulier, nous passerons sous silence certaines difficultés techniques issues de cette approche, et qui rendent difficile la formalisation rigoureuse des développements.

Notons au passage certaines approches alternatives:

- Les variables de De Bruijn [dB91]: il n'y a pas de nom de variable associée au λ, et les variables dans les termes sont représentes par des nombres qui dénotent le nombre de λ à remonter pour arriver à celui qui lie la variable en question; les variables libres sont traites comme des variables liées par des λ "fictifs".
- L'approche "localement sans nom" [MM04]. Cette technique tente de combiner les avantages des termes nommés et des variables à la De Bruijn en donnant des noms aux variables libres et en adoptant l'approche précisée ci-dessus pour les variables liées. Cette technique à été utilisée avec succès pour formaliser certains développements non-triviaux de systèmes avec lieurs.
- L'approche "syntaxe abstraite d'ordre supérieure" [MN87, Hof99]. L'idée ici est d'interpréter une abstraction non pas comme une construction syntaxique, mais comme une vraie fonction de la méta-théorie. Cette approche a été exploitée dans les implémentations et dans les formalisations sur machine, mais est assez peu utilisée pour les preuves sur papier. En effet, il faut restreindre l'espace des fonctions considérées pour ne pas avoir "trop" de termes, c'est à dire des termes pathologiques qui ne sont pas issus de la définition inductive.

Notons que la pratique informelle de la convention de Barendregt peut être mise sur une fondation solide en utilisant la *logique nominale* de Pitts et Gabbay [Pit03, Gab07]. Dans cette approche formelle, les termes sont considérés modulo *permutation* de leur variables (libres et liées) et les propriétés prouvées par induction sur la structures doivent être invariantes par cette action, enfin ils considèrent un opérateur de *fraîcheur* qui affirme qu'une certaine variable ne se trouve *pas* dans un certain ensemble fini.

Les preuves données ici peuvent être formalisées de manière complètement rigoureuse avec l'une des techniques ci-dessus.

Definition 5 Une *substitution* est une application partielle $\mathcal{X} \to \mathcal{T}rm$, dont le domaine est fini. On notera Θ l'ensemble des substitutions. La substitution qui envoie x_1 sur u_1, x_2 sur u_2,...,x_n sur u_n est notée $\{x_1, \ldots, x_n \mapsto u_1, \ldots, u_n\}$.

Soit $\theta \in \Theta$ et $t \in \mathcal{T}rm$. La *substitution θ appliquée à t*, noté $t\theta$, est le terme défini inductivement par:

- $x\theta = \theta(x)$ si x est dans le domaine de θ
- $x\theta = x$ sinon

- $f\theta = f$
- $t_1\ t_2\theta = t_1\theta\ t_2\theta$
- $\lambda y.t\theta = \lambda y.(t\theta)$ si y n'est pas dans le domaine de θ (et n'est pas défini sinon).

On note θ^x_t pour la substitution égale à θ sur le domaine de θ et égale à t sur x, qui est supposé en dehors du domaine de θ.

Remarquons que en général,

$$t\{x_1, \ldots, x_n \mapsto u_1, \ldots, u_n\} \neq t\{x_1 \mapsto u_1\} \ldots \{x_n \mapsto u_n\}$$

En effet, x_j peut apparaître libre dans u_i pour $j < i$. On a alors, par exemple $t\{x_1 \mapsto u_1\}\{x_2 \mapsto u_2\} = t\{x_1, x_2 \mapsto u_1\{x_2 \mapsto u_2\}, u_2\}$

Definition 6 Un ensemble de *filtres* (ou *patterns*) est un ensemble $\mathcal{P}$ équipé d'une opération partielle:

$$\mathsf{match}\colon \mathcal{P} \times \mathcal{T}rm \to \Theta$$

si $\mathsf{match}(p, t)$ est défini, on dit que t est une *instance* de p ou que p *filtre* t avec la substitution $\mathsf{match}(p, t)$.

Cette définition de filtres est très générale. On utilise surtout l'instance suivante:

Definition 7 On prend comme ensemble de filtres l'ensemble $\mathcal{A}lg \subseteq \mathcal{T}rm$ des termes *algébriques*, définis mutuellement avec les *patterns algébriques*:

$$a \in \mathcal{A}lg := f\ p_1 \ldots p_n$$

$$p_1 \ldots p_n \in \mathcal{P}alg := x \mid a$$

avec $x \in \mathcal{X}$ et $f \in \mathcal{F}$.

Pour la fonction match on prend:

$$\begin{array}{lcl} \mathsf{match}(x, t) & = & \{x \mapsto t\} \\ \mathsf{match}(f, f) & = & \{\} \\ \mathsf{match}(t_1\ t_2, u_1\ u_2) & = & \mathsf{match}(t_1, u_1) \uplus \mathsf{match}(t_2, u_2) \end{array}$$

Où $\{\}$ est la substitution à domaine vide, $\uplus$ est l'opération suivante: si θ et θ' sont deux substitutions, $\theta \uplus \theta'$ est définie si pour tout x dans l'intersection des domaines de θ et θ', $\theta(x) = \theta'(x)$. La substitution $\theta \uplus \theta'$ est alors la substitution de domaine $\mathsf{dom}(\theta) \cup \mathsf{dom}(\theta')$ et égale à θ sur $\mathsf{dom}(\theta)$ et à θ' sur $\mathsf{dom}(\theta')\backslash\mathsf{dom}(\theta)$.

$\mathsf{match}(a, b)$ n'est pas défini dans tous les autres cas.

Sauf précision contraire, on prendra $\mathcal{P} = \mathcal{T}rm$ et match comme défini ci-dessus.

2 Réécriture

Nous nous restreignons dans cette thèse à ne considérer que des patterns algébriques pour toutes les règles à l'exception de la règle β. Il est possible de considérer des définitions plus générales de patterns, ce qui permet de donner plus de règles de réécriture. Cependant, la théorie de la réécriture avec patterns algébriques évite certains problèmes épineux, dont le *matching d'ordre supérieur*. De plus, cette restriction est néanmoins très expressive, en particulier elle permet de représenter les programmes de langages fonctionnels purs sous forme de système de réécriture, sous réserve d'ajouter une règle spéciale que l'on appellera β, et sur laquelle nous allons revenir dans ce paragraphe.

Definition 8 Étant donné un ensemble de filtres $\mathcal{P}$, une *règle de réécriture* sur $\mathcal{T}rm$ est un couple $(l, r) \in \mathcal{P} \times \mathcal{T}rm$ que l'on notera $l \to r$. Dans ce cas, l est appelé le *membre gauche* de la règle, et r le *membre droit*.

Donnons maintenant la définition d'une règle bien particulière, et qui justifiera le nom de réécriture d'ordre supérieur.

Definition 9 Prenons comme ensemble de patterns l'ensemble des termes, et comme fonction match l'égalité:

- $\mathsf{match}(t, u) = \{\}$ si $t = u$
- $\mathsf{match}(t, u)$ n'est pas défini sinon.

On définit la *β-réduction* comme l'ensemble des règles de réécritures suivant:

$$\{(\lambda x.t)u \to t\{x \mapsto u\} \mid t, u \in \mathcal{T}rm\}$$

On parlera souvent de l'ensemble des règles définissant la β-réduction comme d'une seule règle. La β-réduction donne un moyen de "réifier" la notion de substitution, qui est fondamentale en informatique.

Une règle de réécriture mène à une notion de *réduction* définie comme suit:

Definition 10 Soit $\rho = (l, r)$ une règle de réécriture, et t un terme. On dit que t se *réécrit en tête* en u sous ρ, noté $t \to^h_\rho u$ si:

- $\mathsf{match}(l, t) = \theta$
- $u = r\theta$

En particulier, $\mathsf{match}(l, t)$ doit être défini. Le terme u est alors appelé le *réduit en tête* de t par ρ. Si il existe u tel que $t \to^h_\rho u$ alors t est appelé *rédex* pour ρ.

Comme match est une fonction partielle, un terme t ne peut avoir au plus qu'un seul réduit, ce qui induit une fonction partielle: $\to_\rho : \mathcal{T}rm \to \mathcal{T}rm$, qui a un terme associe son réduit s'il en a un et n'est pas définie sinon.

La notion de réécriture n'est intéressante que si il est possible d'appliquer une règle à l'intérieur d'un terme. Pour exprimer cette notion de "réécriture profonde", il est utile d'introduire la notion suivante.

Definition 11 L'ensemble des *contextes de termes*, noté $\mathcal{T}_{\text{ctxt}}$ est défini par:

$$C[] \in \mathcal{T}_{\text{ctxt}} := [] \mid t\ C[] \mid C[]\ u \mid \lambda x.C[]$$

si $t, u \in \mathcal{T}rm$ et $x \in \mathcal{X}$.

Si $t \in \mathcal{T}rm$ et $C[] \in \mathcal{T}_{\text{ctxt}}$ alors $C[t]$ est le terme défini par induction sur $C[]$ par:

- si $C[] = []$ alors t
- si $C[] = C'[]\ u$ alors $C'[t]\ u$
- si $C[] = u\ C'[]$ alors $u\ C'[t]$
- si $C[] = \lambda x.C'[]$ alors $\lambda x.C'[t]$ si x n'apparaît pas dans t.

Étant donné une règle de réécriture ρ, des termes t, u et un contexte $C[]$, on dit que t se *réécrit* en u sous ρ dans le contexte $C[]$ (ou à la *position* $C[]$), noté $t \rightarrow_{\rho}^{C[]} u$, si $t = C[t']$, et il existe un terme u' tel que $t' \rightarrow_{\rho}^{h} u'$ et $u = C[u']$.

On dit que t se réécrit en u sous ρ si il existe un contexte $C[]$ tel que $t \rightarrow_{\rho}^{C[]} u$.

Ici nous perdons déjà le déterminisme que nous avions dans la définition de la réécriture de tête: un terme t peut se réécrire en de nombreux termes différents selon la position à laquelle s'opère la réécriture. Notons également que $\rightarrow_{\rho}^{h}=\rightarrow_{\rho}^{[]}$ pour toute règle ρ.

Definition 12 Un ensemble $\mathcal{R}$ de règles de réécriture est appelé un *système de réécriture*. On dit que t se réécrit en u sous $\mathcal{R}$, et on note $t \rightarrow_{\mathcal{R}} u$, si il existe $\rho \in \mathcal{R}$ tel que $t \rightarrow_{\rho} u$.

On dit que t se réécrit en plusieurs pas en u sous $\mathcal{R}$ si $t \rightarrow_{\mathcal{R}}^{*} u$.

On dit que t se réécrit en au moins un pas en u sous $\mathcal{R}$ si $t \rightarrow_{\mathcal{R}} \rightarrow_{\mathcal{R}}^{+} u$.

Par abus nous dirons parfois "se réécrit en un pas" pour éviter les ambiguïtés, et nous oublierons les références a $\mathcal{R}$ si il n'y a pas de confusion possible quand au système de réécriture considéré.

Ici enfin le non déterminisme est triple: un terme t peut se réécrire non seulement à plusieurs positions différentes mais également sous plusieurs règles différentes, et en un nombre de pas différents. Notons qu'un système de réécriture ne doit pas nécessairement avoir tous ses membres gauches qui proviennent du même ensemble de patterns. En particulier nous mélangerons souvent réécriture avec des règles dont les membres gauches sont des patterns algébriques et la règle β.

Introduisons finalement les concepts qui seront l'objet principal de nos préoccupations dans cette thèse:

Definition 13 Soit E un ensemble et R une relation sur E. Un élément $e \in E$ est en *forme normale* pour R si il n'existe aucun e' tel que eRe'.

On dit que $n \in E$ est une *forme normale* de e si eRn et n est en forme normale pour R.

Un élément $e \in E$ est *fortement normalisant* pour R si il n'existe pas de suite infinie $(e_n)_{n \in \mathbb{N}}$ telle que $e = e_0$ et $e_i R e_{i+1}$ pour tout $i \in \mathbb{N}$. On notera alors $e \in \mathcal{SN}_R$ ou $e \in \mathcal{SN}$ si il n'y a pas d'ambiguïté.

On dit que R est *fortement normalisant* (ou *bien fondé*), si pour tout $e \in E$, $e \in \mathcal{SN}_R$.

Une relation induit un principe d'induction sur les éléments fortement normalisants.

Lemma 14 (induction bien fondée)
Soit E un ensemble et R une relation sur E. Soit P un sous-ensemble de E qui vérifie:

$$\forall x, (\forall y, xRy \Rightarrow y \in P) \Rightarrow x \in P$$

Alors $\mathcal{SN}_R \subseteq P$.

Proof. Soit $x \in \mathcal{SN}_R$. Montrons $x \in P$ par contradiction. Supposons que $x \notin P$. Alors par hypothèse il existe y_1 tel que xRy_1 et $y_1 \notin P$. Par l'axiome du choix relationnel, on peut construire une suite infinie $y_1, y_2, \ldots$ telle que pour tout i, y_iRy_{i+1} et $y_i \notin P$. Mais ceci contredit l'hypothèse $x \in \mathcal{SN}_R$. ■

Notons au passage que la preuve ci-dessus utilise une version de l'axiome du choix de manière essentielle. Une autre définition possible de la normalisation forte d'une relation est de prendre le lemme 14 comme postulat pour $\mathcal{SN}$. Cette définition est souvent adoptée dans les approches constructives à la formalisation de résultats de normalisation forte.

On s'intéresse a la normalisation forte pour les termes $t \in \mathcal{T}rm$ sous la relation $\rightarrow_{\mathcal{R}}$ pour un système de réécriture donné.

Notons que la terminaison de la réécriture est indécidable en général [Pos47]. Il n'est donc possible de trouver des procédures de décision *partielles*. En particulier, si une méthode est *correcte*, c'est à dire qu'elle affirme la terminaison de systèmes qui sont effectivement fortement normalisants, alors elle ne peut être *complète*, c'est à dire qu'il y a des systèmes de réécriture fortement normalisants pour lesquelles la procédure ne pourra pas affirmer qu'ils terminent. D'autre part, tout critère à la fois correct et complet ne peut pas être décidable.

La situation est même pire que cela: en absence d'autres règles, il est indécidable de déterminer si un terme t est fortement normalisant sous β! C'est d'ailleurs un des premiers systèmes pour lequel la normalisation a été montré indécidable [Chu41]. Il existe des termes qui ne normalisent pas pour β: Si on prend le terme $\delta = \lambda x.x\ x$, alors

$$\delta\delta = (\lambda x.x\ x)\lambda x.x\ x \rightarrow_\beta (\lambda x.x\ x)\lambda x.x\ x = \delta\delta$$

3 Le typage et le théorème fondamental

Pour "brider" la puissance de la β-réduction, et pour pouvoir exclure les termes comme $\delta\delta$, on introduit une restriction sur l'ensemble des termes considérés: c'est le *typage* [Chu40]. L'intuition est la suivante: à chaque terme $t \in \mathcal{T}rm$ on essaye d'associer un *type* qui dénote une abstraction de son comportement calculatoire. En particulier, si pour chaque u qui a un comportement dénoté par le type T, $t\ u$ a le comportement dénoté par le type U, alors on associe au terme t le type $T \rightarrow U$. Cette intuition sera fondamentale pour comprendre la sémantique des règles de typage. Pour que cette définition soit bien fondée il faut un ensemble de *types de base.*

Definition 15 On définit donc l'ensemble des *types simples*

$$T, U \in \mathcal{T} := B \mid T \rightarrow U$$

avec $B \in \mathcal{B}$ un ensemble de *types de base.*

Nous écrirons $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ au lieu de $T_1 \rightarrow \big(T_2 \rightarrow (\ldots \rightarrow T_n)\ldots\big)$.

Pour que l'inférence des types associés aux termes soit plus facile, nous allons considérer un ensemble modifié de termes, qui portent des annotations de types. On appelle souvent cet ensemble de termes avec annotations *termes à la Church.*

$$\frac{}{\Gamma, x\colon T, \Delta \vdash x\colon T}\ \textbf{ax}$$

$$\frac{}{\Gamma \vdash f\colon \tau_f}\ \textbf{symb}$$

$$\frac{\Gamma \vdash t\colon T \to U \qquad \Gamma \vdash u\colon T}{\Gamma \vdash t\ u\colon U}\ \textbf{app}$$

$$\frac{\Gamma, x\colon T \vdash t\colon U}{\Gamma \vdash \lambda x\colon T.t\colon T \to U}\ \textbf{abs}$$

Figure 1: Règles de typage pour les types simples

Definition 16 L'ensemble $\mathcal{T}rm_{\mathcal{T}}$ des *termes annotes* pour une signature $\mathcal{F}$ est défini par:

$$t, u \in \mathcal{T}rm_{\mathcal{T}} := x \mid f \mid t\ y \mid \lambda x\colon T.t$$

avec $x \in \mathcal{X}$, $f \in \mathcal{F}$ et $T \in \mathcal{T}$.

L'ensemble des *contextes de typage* sur T est défini par:

$$\Gamma \in Ctxt := \bullet \mid \Gamma, x\colon T$$

avec $x \in \mathcal{X}$, $T \in \mathcal{T}$. Nous écrirons $x_1\colon T_1, \dots x_n\colon T_n$ au lieu de $\bullet, x_1\colon T_1, \dots x_n\colon T_n$. On suppose également que les x_i sont *distincts*. Si x apparaît dans un contexte Γ on dit que x est dans le *domaine* de Γ. Plus généralement on adoptera la nomenclature des substitutions, ces dernières étant un analogue sémantique des contextes de typage.

Étant donné une signature $\mathcal{F}$, un *assignement de types* τ est une fonction $\mathcal{F} \to \mathcal{T}$. Si $f \in \mathcal{F}$, on écrit τ_f au lieu de $\tau(f)$. Étant donné $f \in \mathcal{F}$ et un assignement de types τ, on dit que f est *d'arité* n si $\tau_f = T_1 \to \dots T_n \to B$ avec B un type de base.

Lorsqu'il n'y a pas de confusion, nous dirons simplement terme pour terme annoté et contexte pour contexte de typage. Il est clair que nous pouvons étendre toutes les définitions ci-dessus (substitution, filtrage, réécriture), à cette nouvelle classe de termes.

Étant donné un assignement de types τ, il est maintenant possible d'associer un type à certains termes dans $\mathcal{T}rm_{\mathcal{T}}$, suivant les règles décrites dans la figure 1.

Definition 17 Soit Γ un contexte, T un type et t un terme. Un *séquent* est un triplet (Γ, t, T) que l'on écrit $\Gamma \vdash t\colon T$. On dit que *t a le type T dans le contexte* Γ, si $\Gamma \vdash t\colon T$ est dérivable grâce aux règles décrites dans la figure 1. On écrira $\nabla \rhd \Gamma \vdash t\colon T$ si ∇ est *l'arbre de dérivation* du séquent $\Gamma \vdash t\colon T$.

On dira parfois juste $\Gamma \vdash t\colon T$ au lieu de $\Gamma \vdash t\colon T$ est dérivable.

Remarquons que si, au lieu du terme δ décrit ci-dessus, nous prenons le terme $\delta_T = \lambda x\colon T.x\ x$, alors il n'existe aucun T, aucun U et aucun Γ tel que $\Gamma \vdash \delta_T\colon U$. En effet un examen rapide des règles montre que ceci est équivalent à l'assertion: $\Gamma, x\colon T \vdash x\colon T \to U$ et $\Gamma, x\colon T \vdash x\colon T$. Ceci n'est possible que si $T \to U = T$, ce qui est impossible pour notre algèbre de types.

Nous pouvons donc être rassuré que notre premier exemple de terme non fortement normalisant pour la β-réduction ne peut pas être bien typé dans notre système de types. Nous allons aller plus loin et montrer que *tout* terme bien typé dans ce système est fortement normalisant pour β. La normalisation faible a été prouvé par Turing (voir Gandi [CHS80]), mais nous allons utiliser l'idée de preuve de Tait [Tai67] qui prouve encore une fois la normalisation faible, mais qui peut être adaptée à la normalisation forte. C'est cette méthode de preuve qui va servir pour tous les résultats de normalisation forte dans cette thèse.

Theorem 18 Soit Γ, t, T tels que $\Gamma \vdash t : T$. On a $t \in \mathcal{SN}_\beta$.

Proof. Nous allons utiliser la technique dite de *Tait-Girard* [Gir72] appelée aussi celle des *candidats de réductibilité* (*computability candidates* en anglais). Pour prouver la normalisation forte d'un terme $t : T$, nous allons montrer que t appartient a un certain ensemble $[\![T]\!]$ qui ne contient que des termes fortement normalisants. Il faut donc moralement trouver une sémantique des types $[\![\_]\!]$, et montrer son *adéquation*, c'est a dire sa correction vis-à-vis des règles de typage.

Definition 19 A chaque type T on associe un ensemble $[\![T]\!] \subseteq \mathcal{T}rm_{\mathcal{T}}$ par induction sur le type:

- pour un type $B \in \mathcal{B}$, $[\![B]\!] = \mathcal{SN}_\beta$
- pour un type $T \to U$, $[\![T \to U]\!] = [\![T]\!] \rightarrowtriangle [\![U]\!] = \{t \in \mathcal{SN} \mid \forall u \in [\![T]\!], t\ u \in [\![U]\!]\}$

Il s'agit de montrer maintenant que pour chaque type T, $[\![T]\!]$ satisfait certaines conditions de stabilité vis-à-vis de la réduction. Ces conditions utilisent la notion d'élément neutre:

Definition 20 Un terme $t \in \mathcal{T}rm_{\mathcal{T}}$ est dit *neutre* si il est de la forme:

- $x \in \mathcal{X}$
- $t\ u$

Si t n'est pas neutre on dit que c'est une *valeur*. Si v est une valeur et $t \to_\beta v$, on dit que v est une valeur de t.

Lemma 21 Pour tout $T \in \mathcal{T}$, $[\![T]\!]$ satisfait les *conditions de Girard*:

- normalisation forte: $[\![T]\!] \subseteq \mathcal{SN}$.
- stabilité par réduction: si $t \in [\![T]\!]$ et $t \to_\beta u$ alors $u \in [\![T]\!]$.
- condition de faisceau (appelée ainsi dans [Gal90]): si t est neutre, et $\forall u, t \to_\beta u \Rightarrow u \in [\![T]\!]$ alors $t \in [\![T]\!]$.

Proof. Nous procédons par induction sur le type T:

- cas $B \in \mathcal{B}$.
  - Le premier point est évident.
  - La stabilité par réduction est aussi claire: si $t \in \mathcal{SN}$ et $t \to_\beta u$ alors $u \in \mathcal{SN}$.
  - La condition de faisceau: si tout u tel que $t \to_\beta u$ est dans $\mathcal{SN}$ alors pour toute réduction infinie $t \to_\beta t_1 \to_\beta \ldots$, $t_1 \in \mathcal{SN}$, or $t_1 \to_\beta t_2 \to_\beta \ldots$ est une réduction infinie de t_1, contradiction. On peut donc conclure $t \in \mathcal{SN}$.
- cas $T \to U$.

- Le premier point est également clair.
- Stabilité par réduction: Supposons $t \in [\![T]\!] \multimap [\![U]\!]$. Soit $u \in [\![T]\!]$ arbitrairement choisi. On a $t\ u \in [\![U]\!]$ par définition. Par hypothèse d'induction, pour tous les réduits t' de t, $t'\ u \in [\![U]\!]$, car $t'\ u$ est un réduit de $t\ u$. En conséquence, comme u a été choisi arbitraire, t' est dans $[\![T]\!] \multimap [\![U]\!]$.
- Condition de faisceau: prenons $t \in \mathcal{T}rm_{\mathcal{T}}$ neutre et un $u \in [\![T]\!]$ arbitrairement choisi, et supposons que tout réduit t' de t est dans $[\![T]\!] \multimap [\![U]\!]$. Observons que $t\ u$ est neutre. Par hypothèse d'induction, si tout les réduits de $t\ u$ sont dans $[\![U]\!]$ alors $t\ u$ l'est également. Pour prouver cela, nous allons utiliser le fait que u est fortement normalisant par hypothèse d'induction. Nous procédons maintenant par induction bien fondée sur u ordonné par la relation $\rightarrow_{\beta}^{*}$. Les réduits en un pas de $t\ u$ sont de la forme $t'\ u$ avec t' un réduit de t ou $t\ u'$ avec u' un réduit de u. En effet, une réduction en tête ne peut pas se produire, car il faudrait que t soit de la forme $\lambda x.v$, ce qui n'est pas possible car t est neutre.
  - ∗ Si le réduit est de la forme $t'\ u$ on peut conclure par hypothèse sur t (et u).
  - ∗ Si le réduit est de la forme $t\ u'$ alors nous appliquons l'hypothèse d'induction (sur u) pour conclure que tous les réduits de $t\ u'$ sont dans $[\![U]\!]$ et donc (par l'induction sur le type) $t\ u'$ l'est également.

  Tous les réduits de $t\ u$ sont donc dans $[\![U]\!]$ et on peut appliquer l'hypothèse d'induction sur le type pour conclure que $t\ u$ l'est également, puis conclure par généricité de u.

■

Pour pouvoir prouver que les termes bien typés sont dans l'interprétation de leur type, nous allons procéder par induction. Le lemme suivant nous permet de traiter le cas de la règle **symb**.

Lemma 22 (Correction de l'interprétation pour les symboles)
Pour tout $f \in \mathcal{F}$, $f \in [\![\tau_f]\!]$

Proof. Soit $f \in \mathcal{F}$ et supposons $\tau_f = T_1 \rightarrow \dots T_n \rightarrow B$ avec B un type de base. Soit $t_1 \in [\![T_1]\!], \dots, t_n \in [\![T_n]\!]$ arbitraires. Il suffit de montrer que $f\ \vec{t}$ est dans $[\![B]\!] = \mathcal{SN}$. Il suffit d'examiner les réduits de $f\ \vec{t}$: ils sont nécessairement de la forme $f\ t'_1 \dots t'_n$ avec $t_i \rightarrow_{\beta}^{*} t'_i$. En effet, $f\ \vec{t}$ ne peut jamais être un rédex pour β. Chaque t_i étant fortement normalisant par le lemme 21, on peut conclure qu'il ne peut pas y avoir de réductions infinies de $f\ \vec{t}$. ■

Voici un dernier lemme utile pour la normalisation forte.

Lemma 23 Pour tout $x \in \mathcal{X}$ et tout T, $x \in [\![T]\!]$.

Proof. Il suffit d'utiliser la condition de faisceau du lemme 21, en observant que x est en forme normale, tout affirmation sur ses réduits étant donc trivialement vraie. ■

En général, pour prouver des propriétés sur la sémantique opérationnelle du λ-calcul, il faut avoir une notion de substitution "en attente", qui lie chaque variable libre à un terme, c'est la notion de *clôture* (on pourra consulter [Ste78]), qui est également utilisée pour implémenter la β-réduction. Pour montrer la normalisation forte, nous devons définir la notion de *substitution calculable.*

Definition 24 Soit Γ un contexte et θ une substitution. On dit que θ *satisfait* Γ, et on note $\theta \models \Gamma$, si les domaines de θ et Γ coïncident, et $\theta(x) \in [\![\Gamma(x)]\!]$ pour tout x dans le domaine de Γ, θ. Dans ce cas on dira que la substitution θ est *calculable.*

Si t est un terme, θ une substitution et T un type, on notera $\theta \models t : T$ pour $t\theta \in [\![T]\!]$.

La notation $\theta \models t\!:\!T$ rappelle clairement celle de la relation de typage. Le théorème d'adéquation met en relation ces deux notions.

Theorem 25 (Adéquation) Soit t un terme, Γ un contexte et T un type.

$$\Gamma \vdash t\!:\!T \quad \Rightarrow \quad \forall \theta \models \Gamma, \theta \models t\!:\!T$$

On appelle aussi ce théorème: théorème de *correction.*

Proof. Soit Γ, t, T comme dans le théorème. Nous allons procéder par induction sur le jugement $\Gamma \vdash t\!:\!T$. Soit θ une substitution telle que $\theta \models \Gamma$.

- Cas **ax**: $\theta(x) \in [\![T]\!]$ par hypothèse sur θ.
- Cas **symb**: Application directe du lemme 22.
- Cas **app**: Par induction, on a $\theta \models t\!:\!T \to U$ et $\theta \models u\!:\!T$. On a donc par définition de $[\![T]\!] \twoheadrightarrow [\![U]\!]$: $t\theta\ u\theta = (t\ u)\theta \in [\![U]\!]$.
- Cas **abs**: Par hypothèse d'induction, pour tout θ' tel que $\theta' \models \Gamma, x\!:\!T$, on a $\theta' \models t\!:\!U$. Soit $u \in [\![T]\!]$ arbitraire. Montrons que $(\lambda x.t)\theta\ u \in [\![U]\!]$. Nous allons utiliser la condition de faisceau sur $[\![U]\!]$ en exploitant le fait que $(\lambda x.t)\theta\ u$ est neutre. Il faut donc montrer que tous les réduits de $(\lambda x.t)\theta\ u$ sont dans $[\![U]\!]$. Comme on peut supposer que x est en dehors du domaine de θ, $(\lambda x.t)\theta = \lambda x.t\theta$. Les réduits sont de trois types:

1. $(\lambda x.t')\ u$ avec t' un réduit de $t\theta$.
2. $(\lambda x.t\theta)\ u'$ avec u' un réduit de u.
3. $t\theta\{x \mapsto u\}$.

Pour les deux premiers cas, nous allons procéder par induction bien fondée sur $t\theta$ et u respectivement. u est bien fondé par la condition de normalisation forte (lemme 21). En posant $\theta' = \theta^x_x$, on a par le lemme 23, $\theta' \models \Gamma, x\!:\!T$ et donc $t\theta' \in [\![U]\!]$, ce qui permet de déduire que $\lambda x.t\theta' = \lambda x.t\theta$ est fortement normalisant. On montre donc dans les trois cas que le réduit est dans $[\![U]\!]$:

1. Par induction bien fondée sur $\lambda x.t\theta$, tous les réduits de $(\lambda x.t')\ u$ sont dans $[\![U]\!]$, donc par la condition de faisceau et le fait que $(\lambda x.t')\ u$ soit neutre, ce terme est dans $[\![U]\!]$.
2. De même que précédemment.
3. On pose $\theta' = \theta^x_u$. Par hypothèse, $u \in [\![T]\!]$ et $\theta \models \Gamma$, donc $\theta' \models \Gamma, x\!:\!T$. De l'hypothèse d'induction (globale), on peut déduire que $t\theta' = t\theta\{x \mapsto u\}$ est dans $[\![U]\!]$.

Ceci conclut la preuve.

■

Nous pouvons enfin terminer la preuve de normalisation forte: pour tout séquent $\Gamma \vdash t\!:\!T$, on peut donner la substitution θ de même domaine que Γ et qui envoie toute variable sur elle même. Par le lemme 23, $\theta \models \Gamma$, et par le théorème d'adéquation (théorème 25), $t = t\theta \in [\![T]\!]$. Or le lemme 21 affirme $[\![T]\!] \subseteq \mathcal{SN}$, ce qui nous permet de conclure. ■

Nous avons donc ce que nous voulions, une manière de contrôler la non-normalisation de la β-réduction. Nous voulons maintenant combiner celle-ci avec des règles de réécriture algébriques. En effet, bien que la β-réduction sur les termes non typés soit

Turing-complète, il est bien souvent préférable, dans la pratique de la programmation par exemple, d'avoir un système typé, donc avec une β-réduction normalisante, couplé avec des fonctions "nommées", qui peuvent être représentés par des systèmes de réécriture algébriques. En effet, cela permet d'utiliser le système de type pour garantir certaines propriétés de correction du programme, ainsi que d'éviter les lourdeurs (et lenteurs) des encodages de structures de données dans le λ-calcul sans symboles de fonctions. Notons que la normalisation forte d'un système de réécriture algébrique (sans β), même dans un cadre typé, est indécidable.

Definition 26 Une règle $l \rightarrow r$ est dite *bien typée* si pour tout contexte Γ et tout type T:

$$\Gamma \vdash l : T \Rightarrow \Gamma \vdash r : T$$

Un système de réécriture $\mathcal{R}$ est dit bien typé si toutes ses règles le sont.

Property 27 Pour un filtre algébrique l, si $\Gamma \vdash l : T$ alors T est unique, et si Γ ne contient que les variables (nécessairement libres) de l, alors Γ est unique également.

Proof. Procédons par induction sur l. Comme c'est un terme algébrique, il est de la forme $f\ p_1 \dots p_n$ avec les p_i des patterns algébriques. On procède par induction sur n. Si $n = 0$, alors $l = f$ et $T = \tau_f$. Sinon, comme $\Gamma \vdash f\vec{p} : T$, on a $\Gamma \vdash f\ p_1 \dots p_{n-1} : T_1 \to T$ et $\Gamma \vdash p_n : T_1$. On peut alors raisonner par cas sur p_n:

- cas variable: $\Gamma \vdash x : T_1$. Le type T_1 est déterminé de manière unique par τ_f. De plus la seule manière de dériver ce séquent est par la règle **ax**, donc x est dans le domaine de Γ et $\Gamma(x) = T_1$.
- cas terme algébrique: par hypothèse d'induction, T_1 est déterminé de manière unique par p_n, et Γ est déterminé de manière unique sur les variables de p_n.

■

Une proposition plus générale apparaît dans Barbanera *et al* [BFG97].

Il est donc possible de prouver qu'une règle de réécriture algébrique $l \rightarrow r$ est bien typée en exhibant simplement un type T et un contexte Γ qui ne contient que les variables de l, tel que $\Gamma \vdash l : T$ et $\Gamma \vdash r : T$.

Cependant, il ne suffit pas de considérer un système de réécriture $\mathcal{R}$ fortement normalisant dans un cadre typé pour pouvoir conclure à la normalisation forte du système $\mathcal{R} \cup \beta$. On dit que la normalisation forte est *non modulaire*. Illustrons notre propos par un exemple tiré de Blanqui [Bla05b].

Example 1 Soit $\mathcal{B}$ l'ensemble $\{B\}$ des types de bases (réduit à un seul élément) et $\mathcal{F}$ la signature $\{f, c\}$, avec l'assignement de types $\tau_c = (B \to B) \to B$ et $\tau_f = B \to B \to B$. On définit le système de réécriture (algébrique) $\mathcal{R}$ suivant:

$$\mathcal{R} = \{f\ (c\ x) \rightarrow x\}$$

Le système $\mathcal{R}$ est bien typé et fortement normalisant. En effet, à chaque terme $t \in \mathcal{T}rm_{\mathcal{T}}$ nous pouvons associer le nombre n_t de symboles c contenus dans t. Ensuite il est facile de vérifier que pour chaque u tel que $t \rightarrow_{\mathcal{R}} u$, $n_t > n_u$. Il ne peut donc y avoir de suite infinie de pas de réécriture.

Cependant $\mathcal{R}\cup\beta$ n'est pas fortement normalisant sur les termes bien typés avec la β-réduction: si

$$\delta = \lambda x\colon B.(f\ x)\ \Big(c\ (f\ x)\Big)$$

alors

$$\delta\ (c\ \delta) \rightarrow_\beta \Big(f\ (c\ \delta)\Big)\ \Big(c\ (f\ (c\ \delta))\Big) \rightarrow^*_\mathcal{R} \delta\ (c\ \delta)$$

Le terme $\delta\ (c\ \delta)$ ne normalise donc pas sous $\mathcal{R} \cup \beta$.

Remarquons que le terme δ ci dessus présente une forte similarité avec notre terme $\lambda x.x\ x$ présenté ci-dessus comme contre-exemple de la normalisation des termes non-typés. Le phénomène principal qui permet l'écriture de tels termes est l'absence de *positivité*: le terme c est de type $(B \rightarrow B) \rightarrow B$, et il y a donc une occurrence *négative* du type B dans les arguments de la fonction c. Une condition de positivité est donc nécessaire pour pouvoir garantir la normalisation des termes typés pour des systèmes de réécriture intéressants, on pourra consulter Mendler [Men87] pour une discussion détaillée.

Dans les chapitres qui suivent, nous allons donner des critères pour la combinaison de systèmes de réécriture algébriques et la β-réduction.

Part I

Semantics for size-based termination

1

A Type-Based Termination Criterion

We describe a type system more complex than the one given in definition 17. The idea is straightforward: type systems allow us to compute an abstraction of the *value* of a program, where value may refer to the interpretation of the program in some given semantics, or just the set of normal forms of its representation as a (strongly normalizing) term. Computing the value of a program is undecidable, as is determining if a program terminates, but it is furthermore very easy to prove undecidability of one given the undecidability of the other, as we may show in the following informal reasoning. Suppose we have given a procedure t that takes as input p and determines if p terminates on the empty input. We can build a procedure that determines the value of p in the following manner: apply t to p. If t returns `non-terminating`, then return `no value`. Otherwise we compute the normal form of p, which exists by definition and return that. In the other direction, let v be a procedure that computes the value of a given program, or returns `no value` if there is none. To build t we can just apply v to p and if v returns `no value`, then we return `non-terminating`, and return `terminating` otherwise.

More pragmatically, it is often possible to use information on the value of a terms to show termination. Let us give an informative example.

Example 2 (Structural recursion)
Let $\mathcal{R}$ be the following rewrite system over $\mathcal{T}rm$: we give the signature $\Sigma = \{0, S, f\}$ and the rules $\mathcal{R} = \{f\ (S\ x) \to S\ (f\ x)\}$. Then $\mathcal{R}$ (without β-reduction) is strongly normalizing. Note first that simple counting of the number of symbols is insufficient to prove termination (contrary to example 1) because the rule preserves the number of symbols.

Proof.
Consider the *size* function which maps a term t to the number of $s$es at the *head* of t:

- $\text{size}(S\ t) = \text{size}(t) + 1$
- $\text{size}(0) = 0$
- $\text{size}(\lambda x.t) = 0$
- $\text{size}(t\ u) = 0$ if $t \neq S$.

It is easy to show that terms in $\mathcal{SN}$ only have a finite number of reducts (as the rewrite system is finite). We can therefore define the following interpretation function $(\!|\_|\!) : \mathcal{SN} \to \mathbb{N}$

$$(\!|t|\!) = \max_{t \to^*_{\mathcal{R}} u} \mathrm{size}(u)$$

We can then use the $(\!|\_|\!)$ function to prove strong normalization of $\mathcal{R}$. Let t be a term. We show by structural induction that $t \in \mathcal{SN}$:

- Variable case: trivial.
- Case $\lambda x.t$: this term is strongly normalizing if and only if t is strongly normalizing, which is true by induction hypothesis.
- Case 0, S: trivially true.
- Case $t\ u$: Notice first that t and u are in $\mathcal{SN}$ by induction hypothesis. We proceed by case analysis on t. If $t \neq f$, then the reducts of $t\ u$ are of the form $t'\ u'$ with t' and u' reducts of t and u respectively. Indeed, if $t \neq f$, then a simple examination of the (unique) rule suffices to see that t may never reduce to f, and then we may never apply this rule to $t'\ u'$. We can therefore conclude by the induction hypothesis.

  Otherwise, if $t = f$, then we look at possible reducts of $f\ u$. We proceed by lexicographic induction on the pair $((\!|u|\!), u)$ ordered by $>_{\mathbb{N}} \times \to^+_{\mathcal{R}}$, u is strongly normalizing by induction hypothesis, so $(\!|u|\!)$ is defined, and $\to^+_{\mathcal{R}}$ is well-founded on reducts of u.

  - $f\ u'$ with $u \to^+ u'$. We have $((\!|u|\!), u) > ((\!|u'|\!), u')$ and can conclude by induction hypothesis.
  - $S\ (f\ v)$ with $u = S\ v$. It suffices to show that $f\ v$ is in $\mathcal{SN}$. By definition $(\!|u|\!) >_{\mathbb{N}} (\!|v|\!)$, and we can apply the induction hypothesis.

  All reducts of $f\ u$ are therefore in $\mathcal{SN}$, and so is $f\ u$. ■

We can extract two major elements from the normalization proof above:

1. The use of a *semantics* to express *decrease*.
2. The notion of *recursive call*: to prove normalization of $f\ (S\ v)$ it *suffices* to prove normalization of $f\ v$, which is a subterm of a reduct of $f\ (S\ v)$.

There are similarities between the above proof and the proof of theorem 18. We shall use a type system to build a syntactic approximation of (a variant of) the $(\!|\_|\!)$ function defined above, but which allows treatment of β-reduction. Typing will be used both to compute the approximation and to guarantee that b-reduction is well-behaved. Such size-based type systems were introduced independently by Hughes *et al* [HPS96] and Gimenez [Gim96], and subsequently underwent many developments [Abe06, BR06, Bla05a, CK01, BFG$^+$04, BGP06, BGP09].

In what follows we describe an adaptation to simple types of the the type system given in Blanqui [Bla04], which is described for the Calculus of Algebraic Constructions [Bla01], an extension of the Calculus of Constructions of Coquand and Huet [CH88]. The system for simple types is described in Blanqui & Roux [BR09], and is quite similar to that exposed in Barthe *et al* [BFG$^+$04], where the main difference is that their formalism involves inductive datatypes with a fixpoint operator and an explicit matching construct rather than rewrite rules. The system described here presents the advantage of allowing (certain) matches on defined symbols. However there are systems which can be shown to terminate in the Barthe *et al* framework but not with the criterion we give here (see example 9).

1.1 The type system

In everything that follows, we will assume that signatures are countable. This is not a very strong restriction in practice, as actual (physical) programs may only contain a finite number of symbols.

Definition 28 (Size types)
We define the *size-algebra* $\mathcal{A}$ with the following BNF:

$$a, b \in \mathcal{A} := \alpha \mid 0 \mid \mathrm{s}(a) \mid \max(a, b) \mid \infty$$

with $\alpha \in \mathcal{V}$ an infinite set of *size variables*. We can then define the *simple sized-types*:

$$T, U \in \mathcal{T}_{\mathcal{A}} := B^{a} \mid T \to U$$

With $B \in \mathcal{B}$ the set of *base types*. A type of the form B^a is called an *atomic type*.

The set of *terms* is then defined by:

$$t, u, t_1, \ldots, t_n \in \mathcal{T}rm_{\mathcal{A}} := x \mid f \mid \lambda x{:}T \mid t\ u$$

With $f \in \Sigma$ a *signature* and $T \in \mathcal{T}_{\mathcal{A}}$. We write $\mathcal{T}rm$ if there is no ambiguity.

We then extend the definitions for rewrite systems and reduction of terms in $\mathcal{T}rm_{\mathcal{A}}$ by straightforward analogy with that on non annotated terms as described in chapter (definitions 9, 12).

A *type assignment* τ is a function from Σ to $\mathcal{T}_{\mathcal{A}}$.

We can easily define an erasure of a sized-type to a simple type.

Definition 29 For each size-type T, we define T, the type T^{∞} is obtained from T by replacing each size annotation by ∞.

We define the *erasure* as follows:

- $|B^a| = B$
- $|T \to U| = |T| \to |U|$

We separate Σ into two subsets: the subset $\mathcal{C}$ of *constructors* and the set $\mathcal{D}$ of *eliminators*. Only constructors may appear deeply in a left-hand-side of a rule, and only eliminators can appear at the head of a left-hand-side. However these subsets need not be disjoint! In particular a constructor may be defined by rewrite rules. The distinction is useful to enforce more restrictive typing rules on constructors.

We wish to give a size semantics to terms using $\mathcal{A}$. The intuition is the following: if a term t is of size n, then the application of a constructor c to t is of size $n + 1$. If c takes two arguments t and u of sizes n and m, then the size of $c\ t\ u$ is taken to be $\max(n, m) + 1$. We must then give a type system capable of assigning an approximation of this size, and then check that recursive calls are made on terms of smaller size, as in example 2. A first difficulty arises from the fact that certain terms can not be given a finite size. Consider the example of *Brouwer ordinals*:

Example 3 (Brouwer Ordinals)
Take the following base types:

$$\mathcal{B} = \{Nat, Ord\}$$

And the following constructors:

$$\mathcal{C} = \{0, S, 0_O, S_O, Lim\}$$

With the following type assignment:

$$\begin{array}{lcl} \tau_0 & = & Nat \\ \tau_S & = & Nat \to Nat \\ \tau_{0_O} & = & Ord \\ \tau_{S_o} & = & Ord \to Ord \\ \tau_{Lim} & = & (Nat \to Ord) \to Ord \end{array}$$

And as eliminators:

$$\mathcal{D} = \{rec_T \mid T \in \mathcal{T}\}$$

With types $\tau_{rec_T} = T \to (Nat \to T \to T) \to Nat \to T$. We give the rewrite rules:

$$\mathcal{R} = \{rec_T\ t\ u\ 0 \to t, rec_T\ t\ u\ (S\ \ v) \to u\ v\ (rec_T\ t\ u\ v) \mid T \in \mathcal{T}\}$$

The *Brouwer ordinals* are the set of terms of type Ord in the empty context.

The Brouwer ordinal $Lim\ (rec_{Ord}\ 0_O\ S_O)$ can not be attributed a finite size. Indeed, if we give size n to terms of the shape

$$S_o^n\ 0_O = S_O(\dots(S_O\ 0_O)\dots)$$

Where S_O appears n times, then we cannot bound the size of $rec_{Ord}\ 0_O\ S_O\ t$ for any t, as it is equal to n for $t = S^n\ 0$.

But if we consider $Limf$ as a tree, it is reasonable to consider $f\ t$ as a subtree of it. Therefore, $Lim\ (rec_O\ 0_O\ S_O)$ has as subtrees trees of size n for any $n \in \mathbb{N}$. To talk about the size of this term, it is therefore necessary to be able to consider terms of *ordinal size*. The term above can then be given the size $\omega + 1$, where ω the first infinite ordinal.

However we do not want to explicitly speak of ordinals in our type system. Indeed in general it is not necessary to know the exact size of a term to be able to compare the size of arguments in a recursive call. It is only necessary to be able to express the *difference in sizes* between the two terms. Our type annotations are capable of expressing such differences if they are strictly smaller than ω.

The elements $0, \mathsf{s}$ and max have a natural interpretation in ordinals, and we use the ∞ symbol to denote either an unknown ordinal, or one that can not be expressed with s, max and 0 alone.

One can only give a size denotation to a term built with constructors unless their type respects certain *positivity conditions.* We draw our definitions from Mendler [Men87].

Definition 30 (Positive and negative positions)
Take $T \in \mathcal{T}$ or $\mathcal{T}_{\mathcal{A}}$ and let A be an atomic type. A appears *positively* in T if:

- $T = A$
- $T = T_1 \to T_2$ and A appears *positively* in T_2 or *negatively* in T_1 or both.

And A appears *negatively* in T if $T = T_1 \to T_2$ and A appears positively in T_1 or negatively in T_2 or both.

Finally A appears *strictly positively* in T if:

- $T = A$
- $T = T_1 \to T_2$ and A appears strictly positively in T_2.

It is obvious that if an atomic type appears strictly positively in T then it appears positively in T.

A datatype represents a structure in informatics. We use our base type to represent *algebraic datatypes* [MTH91]. To define the semantics of such structures (and give a correct definition of a well-formed datatype), we suppose we are given a *declaration preorder* $>_{\mathcal{B}}$ over $\mathcal{B}$. We suppose that this preorder is well-founded on its strict part, and we write $B \simeq_{\mathcal{B}} C$, if $B \geq_{\mathcal{B}} C$ and $C \geq_{\mathcal{B}} B$.

Definition 31 (Well-formed constructors) Given a signature $\Sigma = \mathcal{C} \cup \mathcal{D}$ and a type assignment τ, a symbol $c \in \mathcal{C}$ is a *constructor of* $B \in \mathcal{B}$ if

$$\tau_c = T_1 \to \ldots \to T_n \to B^a$$

with $T_1, \ldots, T_n \in \mathcal{T}_{\mathcal{A}}$ and $a \in \mathcal{A}$

A constructor c of B is *strictly positive* if its type τ_c is of the form $T_1 \to \ldots \to T_n \to B^a$ and if C^b appears in T_i then $C \leq_{\mathcal{B}} B$, and furthermore if $C \simeq_{\mathcal{B}} B$, then C^b appears strictly positively in T_i. The indexes i in which some $C \simeq_{\mathcal{B}}$ appears are called *inductive indexes* and the arguments at those indexes, *inductive arguments.*

A constructor c of $\mathcal{B}$ is *well-formed* if its type is of the form $\tau_c = T_1 \to \ldots \to T_n \to B^a$, if it is strictly positive, and

- If C^b appears in T_i with $C \simeq_{\mathcal{B}} B$, then $b = \alpha_i$ is a variable in $\mathcal{V}$.
- If C^b appears in T_i with $C \not\simeq_{\mathcal{B}} B$, then $b = \infty$.
- All size variables appearing in the T_i are distinct.
- $a = \mathrm{s}(\max(\alpha_1, \ldots, \alpha_k))$ with $\alpha_1, \ldots, \alpha_k$ being the size variables in τ_c and

  $$\max(a_1, \ldots, a_k) = \max(a_1, \max(a_2, \ldots, \max(\alpha_{k-1}, \alpha_k) \ldots))$$

  if $k \geq 2$, otherwise

  $$\max(a) = a$$

  if $k = 1$ and

  $$\max() = 0$$

  if $k = 0$.

Functions are defined by constructor elimination. A function is *recursive* if its value in some arguments depends on its value in other arguments. Clearly, a function f defined by rewriting may be recursive if one of the rules defining f, that is of the form $f\vec{l} \to r$, contains the symbol f in the right hand side.

However a function may be recursive without fulfilling this condition. For instance, taking $\{f\ x \to g\ x, g\ y \to f\ y\}$, the value of $f\ x$ depends on its own value (again in x). Recursive function definitions are the fundamental source of non-termination in typed rewrite systems, so we need a way to deal with this system. In fact we need a slightly more general treatment of recursion, which expresses the fact that f and g are *mutually defined.* We proceed as for mutually defined datatypes.

Definition 32 (Call preorder) A *call preorder* over a signature Σ is a preorder $\geq_{\mathcal{D}}$ over $\mathcal{D}$ which is well-founded in its strict part. We write $f \simeq_{\mathcal{D}} g$ if $f \geq_{\mathcal{D}} g$ and $f \leq_{\mathcal{D}} g$.

A left-algebraic rewrite system $\mathcal{R}$ over Σ is said to *respect* $>_{\mathcal{D}}$ if in every rule $f\vec{l} \to r \in \mathcal{R}$ and every symbol $g \in \Sigma$ that appears in r or $\vec{l}$, $f \geq_{\mathcal{D}} g$.

We are only going to consider *orthogonal* systems here, as they will facilitate the model construction of chapter 4. Orthogonal systems are systems for which rewrite rules may not *interact*, that is at each position in a term only one head rewrite-rule may apply.

Definition 33 (Orthogonality) Let l, l' be two algebraic patterns with distinct variables. We say that l and l' are *overlapping* if there exists a subterm m of l such that m and l' are *unifyable*, or symmetrically. If in this case $m = l$ then we say that it is a *head overlap*.

An algebraic pattern is *linear* if every variable occurs only once.

A left-algebraic rewrite rule $l \to r$ is *left-linear* if l is linear, and given two rules $\rho_1 = l_1 \to r_1$ and $\rho_2 = l_2 \to r_2$, we say ρ_1 and ρ_2 *overlap* if:

- $\rho_1 \neq \rho_2$ and l_1 and l_2 overlap.
- $\rho_1 = \rho_2$ and l_1 and l_2 have a *non-head overlap*.

We say that a left-algebraic rewrite system $\mathcal{R}$ is *orthogonal* if every rewrite rule is left-linear and for every pair ρ_1, ρ_2 of rules do *not* overlap.

It is a fact that orthogonal rewrite systems are easier to study, and in particular the are *confluent*, even in the presence of β-reduction: if $\mathcal{R}$ is an orthogonal rewrite system, then for every term t and terms u, v such that $t \to^*_{\mathcal{R}\cup\beta} u, t \to^*_{\mathcal{R}\cup\beta} v$, then there exists a term w such that $u \to^*_{\mathcal{R}\cup\beta} w$ and $v \to^*_{\mathcal{R}\cup\beta} w$. See van Oostrom [vO94] for an in depth discussion. Note that a term may overlap itself as we may have for instance $f\ (f\ x)$ which unifies with $f\ y$ (as we require distinct variable names).

We can now give the typing rules that will allow us to give size-labelled types to terms in $\mathcal{T}_{\mathcal{A}}$.

Definition 34 (Size-type system)
A term t in $\mathcal{T}_{\mathcal{A}}$ is *well-typed* of type T under the context Γ in the size-type system if

$$\Gamma \vdash_{size} t : T$$

is derivable using the rules of figure 1.1. We adopt the nomenclature and notations of definition 17 for simple types.

The type system involves an order on the size labels, which will also be used in checking decrease of recursive calls.

Definition 35 The *size order* $\leq \subseteq \mathcal{A} \times \mathcal{A}$ is defined by the rules of figure 1.2. We write $a \simeq b$ if $a \geq b$ and $a \leq b$. We define the *strict* size order by taking

$$\mathrm{s}(a) > b \Leftrightarrow a \geq b \wedge a, b \not\simeq \infty$$

$$\frac{}{\Gamma, x\colon T, \Delta \vdash_{size} x\colon T}\ \textbf{ax}$$

$$\frac{\phi \text{ subst}}{\Gamma \vdash_{size} f\colon \tau_f \phi}\ \textbf{symb}$$

$$\frac{\Gamma \vdash_{size} t\colon T \to U \qquad \Gamma \vdash_{size} u\colon T}{\Gamma \vdash_{size} t\ u\colon U}\ \textbf{app}$$

$$\frac{\Gamma, x\colon T^{\infty} \vdash_{size} t\colon U}{\Gamma \vdash_{size} \lambda x\colon |T|.t\colon T \to U}\ \textbf{abs}$$

$$\frac{\Gamma \vdash_{size} t\colon T \qquad T \leq U}{\Gamma \vdash_{size} t\colon U}\ \textbf{sub}$$

Figure 1.1: Typing rules for labelled types

$$\frac{}{0 \leq a} \qquad \frac{}{\alpha \leq \alpha}$$

$$\frac{a \leq b}{a \leq \mathrm{s}(b)} \qquad \frac{a \leq b}{\mathrm{s}(a) \leq \mathrm{s}(b)}$$

$$\frac{a \leq b}{a \leq \max(b, c)} \qquad \frac{a \leq c}{a \leq \max(b, c)}$$

$$\frac{a \leq c \qquad b \leq c}{\max(a, b) \leq c} \qquad \frac{}{a \leq \infty}$$

Figure 1.2: Order on the size algebra

We have defined a set of types labelled with a certain size algebra. Notice however that the domain of abstractions (the type T in the expression $\lambda x\colon T.t$) is restricted: every size variable in T is required to be ∞. This is a restriction on the set of typeable terms, and also on the set of rewrite systems that pass the termination criterion, as for instance example 9. The reasons for this restriction are twofold.

- It allows preservation of the *subject reduction* property:

  $$\forall \Gamma, t, T,\ \Gamma \vdash_{size} t\colon T, t \to_{\mathcal{R}\cup\beta} u \Rightarrow \Gamma \vdash_{size} u\colon T$$

  We give a counter example to this property in the absence of the restriction in example 4.

- The formalism that we describe in chapters 3 and 5 that will allow us to prove correctness of the criterion is incapable of treating systems in which abstracted variables can contain size information.

The type system is quite similar to that for simple types (see chapter *definitions*) with the following differences:

$$\frac{a \leq b}{B^a \leq B^b}$$

$$\frac{T_2 \leq T_1 \qquad U_1 \leq U_2}{T_1 \to U_1 \leq T_2 \to U_2}$$

Figure 1.3: Subtyping rules

- The **symb** rule. The type of function symbols is implicitly universally quantified in its free size variables. We therefore authorize arbitrary instantiations of these by a substitution ϕ, as a function symbol is introduced.
- The **abs** rule. As previously mentioned, we only authorize abstractions over variables of type T^∞ for a given T. This prevents size information to be contained in abstracted variables.
- The **sub** rule. The intuitive semantics of type B^a is the set of B elements with size *at most a*. Subtyping allows one to express this intuition. There are many natural rules that can only be typed using the subtyping rule. For instance the system given in example 6 can only be typed if we can infer $Nat^0 \leq Nat^\infty$. This is a common occurrence in functions defined by matching on arguments, where we need the output type of the function to be an upper bound of the types of the right hand sides.

Example 4 (Counter-example to subject reduction)
Let us give a counterexample to subject reduction if we allow arbitrary size annotations in abstractions. This example is taken from Barthe *et al* [BGP09]. Suppose that the **abs** rule is as follows:

$$\frac{\Gamma, x\colon T \vdash_{size} t\colon U}{\Gamma \vdash_{size} \lambda x\colon T.t\colon T \to U}\ \textbf{abs}$$

Take the type $Nat \in \mathcal{B}$, with constructors 0 and S in Σ with respective types Nat^0 and $Nat^\alpha \to Nat^{s(\alpha)}$. Consider the defined function $f \in \Sigma$ of type $Nat^\alpha \to Nat^\alpha$ defined by the following rules:

$$\begin{array}{lll} f\ 0 & \to & 0 \\ f\ (S\ \ x) & \to & (\lambda z\colon Nat^\beta.z)x \end{array}$$

This system is well-typed in the context $\Gamma = x\colon Nat^\beta$. The judgement

$$y\colon Nat^\gamma \vdash_{size} f\ (S\ \ y)\colon Nat^\gamma$$

is derivable, but $f\ (S\ \ y)$ reduces to $(\lambda\ z\colon Nat^\beta.z)y$, and it is not possible to type this term, as $Nat^\gamma \nleq Nat^\beta$.

1.2 Decrease of recursive calls and the main theorem

We need a way to observe decrease in recursive calls. This decrease is measured on the size of the arguments applied to defined functions. We can only measure size of elements of base type, and for each defined function symbol we identify a certain number of arguments for which we wish to observe decrease: the *recursive arguments.*

Definition 36 (Elimination form)
Let $\Sigma = \mathcal{C} \cup \mathcal{D}$ be a signature and $f \in \mathcal{D}$ be a defined function with type τ_f. We say that τ_f is in *elimination form* if there is some natural number k such that

$$\tau_f = B_1^{\alpha_1} \to \ldots \to B_k^{\alpha_k} \to T_1 \to \ldots T_n \to B_f^{a_f}$$

with $\mathcal{FV}(a_f) \subseteq \{\alpha_1, \ldots, \alpha_k\}$, and α_i not appearing in T_j for any i, j and $\alpha_i \neq \alpha_j$ if $i \neq j$. We say that $B_1, \ldots, B_k$ are the *recursive types* of f, and k is the *number of recursive arguments of* f.

We suppose from now on that for every signature $\Sigma = \mathcal{D} \cup \mathcal{C}$, $f \in \mathcal{D}$ and every typing function τ that τ_f is in elimination form, with n_f the number of recursive arguments.

To compare the size of the arguments, we use a valuation function that allows us to combine the sizes of the different recursive arguments into one global decrease, for instance by applying a lexicographic or multi-set transformation.

Definition 37 (Function status)
If Σ is a signature, τ a typing function and f is a defined function with k recursive arguments, a *status* for f is a function $stat^f$ that takes a well-founded set $(E, >)$ and returns a well-founded order $>_f$ over E^k that *respects* $>$:

- if $\vec{e}, \vec{e}', \vec{e}'' \in E^k$, and for every $1 \leq i \leq k$, $e_i \geq e'_i$ then
$$\vec{e}' >_f \vec{e}'' \Rightarrow \vec{e} >_f \vec{e}''$$
- if $E, >_E$ and $F, >_F$ are well-founded sets and $s : E \to F$ verifies
$$e >_E e' \Rightarrow s(e) >_F s(e')$$
then for $\vec{e}, \vec{e}' \in E^k$
$$\vec{e} >_f \vec{e}' \Rightarrow s(\vec{e}) >_f s(\vec{e}')$$

It is trivial to check that the lexicographic order verifies this property.

All we need to do now is define a typing judgement that, given a pair $(f, \vec{a})$ of a defined function and size annotations $(a_1, \ldots, a_k)$, expressed the following fact: *all functions $g \simeq_{\mathcal{D}} f$ are called on arguments that are strictly smaller than $a_1, \ldots, a_k$.*

Definition 38 (Recursive judgement)
Let Σ be a signature with τ a typing function. We suppose given a status $stat^f$ for each $f \in \mathcal{D}$. Furthermore suppose that if $f \simeq_{\mathcal{D}} g$, then the number of recursive arguments for f and g are equal and $stat^f = stat^g$.

Suppose the number of recursive arguments of f is k, and take $a_1, \ldots, a_k$ in $\mathcal{A}$. The *recursive judgment* $\vdash^f_{\vec{a}}$ is defined by the rules of figure 1.1, with the exception of the **symb** rule which is replaced with the rules:

$$\frac{\vec{a} >_f \vec{b} \qquad g \simeq_{\mathcal{D}} f}{\Gamma \vdash^f_{\vec{a}} g : \tau_g \phi}\ \textbf{symb-call}$$

With ϕ a substitution, $\tau_g = B_1^{\beta_1} \to \ldots B_k^{\beta_k} \to U_1 \to \ldots U_m \to B_g^{b_g}$ and $\vec{b} = \vec{\beta}\phi$

And the rule

$$\frac{g <_{\mathcal{D}} f}{\Gamma \vdash^{f}_{\vec{a}} g : \tau_g \phi}\ \textbf{symb-call'}$$

Where ϕ is an arbitrary substitution.

This presentation of a recursive jugment is quite close to that described by Xi [Xi01]. We use this recursive call judgement to give a type-based termination criterion for left-algebraic rules with β-reduction. To do this, we need to correctly approximate the size of the arguments in the left-hand-side of rewrite rules. Typing is insufficient without further modifications:

Example 5 Take *Nat* as (the only) atomic type, and split $\Sigma = \{0, S, f\}$ into $\mathcal{C} = \{0, S\}$ and $\mathcal{D} = \{f\}$. The labelled types are as follows: $\tau_0 = Nat^0, \tau_S = Nat^\alpha \rightarrow Nat^{\mathrm{s}(\alpha)}, \tau_f = Nat^\gamma \rightarrow Nat^\infty$. We define $\mathcal{R}$ to be the rewrite system with the unique rule

$$f\ (S\ \ x) \rightarrow f\ (S\ \ 0)$$

Let $\Gamma = x\colon Nat^{\mathrm{s}(\alpha)}$. We can derive

$$\Gamma \vdash_{size} S\ \ x\colon Nat^{\mathrm{s}(\mathrm{s}(\alpha))}$$

Furthermore

$$\Gamma \vdash_{size} S\ \ 0\colon Nat^{\mathrm{s}(0)}$$

But we have $\mathrm{s}(\mathrm{s}(\alpha)) > \mathrm{s}(0)$. The naive criterion (that does not have any restriction on the typing of left-hand sides) is therefore satisfied, as there is a strict decrease in the type of the argument of the recursive call. However this system is not strongly normalizing on well-typed terms, as we can derive

$$\vdash_{size} f\ (S\ \ 0)\colon Nat^\infty$$

And this term admits the infinite reduction

$$f\ (S\ \ 0) \rightarrow f\ (S\ \ 0) \rightarrow \ldots$$

The problem can be described in this manner: there is no instance a of the size variable α such that the term $S\ 0$ has size $\mathrm{s}(\mathrm{s}(a))$ (as a tree). The context Γ is therefore not suitable for typing the left-hand side of our rule. This property, namely having equality between the size of an instance of a left-hand side and some instance of the size-annotation in its type, shall be stated and proven explicitly in chapter 4 (lemma 102).

As a consequence, we must give a more restrictive version of typing for left hand sides.

Definition 39 (Minimal typing)

Let $f\ p_1 \ldots p_n$ be an algebraic pattern. We define *minimal typing* as the judgement defined by the rules of figure 1.2, where a type A is *minimal* in the context Γ if it is of the form

$$T_1 \rightarrow \ldots T_n \rightarrow B^\alpha$$

with:

- α a variable that does *not* appear in Γ.
- for each i, all the size annotations that appear in T_i are equal to ∞.

$$\frac{A \text{ minimal in } \Gamma, \Delta}{\Gamma, x : A, \Delta \vdash_{min} x : A}$$

$$\frac{\Gamma \vdash_{min} t : T \to U \qquad \Gamma \vdash_{min} u : T'}{\Gamma \vdash_{min} t\ u : U\phi}\ T\phi = T'$$

$$\frac{}{\Gamma \vdash_{min} c : \tau_c}\ c \in C$$

Figure 1.4: Rules for minimal typing

This constraint on typing is related to the *pattern condition* in Blanqui & Roux [BR09] and *accessibility* in [BR06], which are more semantic in nature, and are guaranteed by our minimal typing rules.

Notice that $\vdash_{min} \subseteq \vdash_{size}$. We have omited the abstraction rule, as it is not necessary to type algebraic patterns and the subtyping rule, as it leads to problems similar to the ones described in example 5.

Definition 40 (Size-criterion)
Let $\Sigma = C \cup \mathcal{D}$ be a signature, and τ be a typing function. A rule $l \to r$ is *well-formed* if $l = f\ p_1 \ldots p_n$ is an algebraic pattern with $f \in \mathcal{D}$, every function symbol in p_i is in C and n is equal to the number of recursive arguments of f.

Let $\mathcal{R}$ be a left-algebraic rewrite system over Σ, $>_{\mathcal{B}}$ a declaration preorder $>_{\mathcal{D}}$ a call preorder, and we suppose given a status $>_f$ for each $f \in \mathcal{D}$. The system $\mathcal{R}$ *passes the size-criterion* if

- $>_{\mathcal{D}}, >_{\mathcal{B}}$ are well-founded.
- $\mathcal{R}$ respects $>_{\mathcal{D}}$.
- Every $c \in C$ is a well formed constructor for some $B \in \mathcal{B}$.
- Every $f \in \mathcal{D}$ is in elimination form.
- For every rule $f\ l_1 \ldots l_n \to r$, n is the number of recursive arguments of f.
- $\mathcal{R}$ is orthogonal.
- Each rule $f\vec{l} \to r \in \mathcal{R}$ satisfies the *decrease condition*: There is a context Γ and a type $T \in \mathcal{T}_{\mathcal{A}}$ such that
  - $\Gamma \vdash_{min} l_i : B_i^{a_i}$
  - $\Gamma \vdash_{min} f\ \vec{l} : T$
  - $\Gamma \vdash^{f}_{\vec{a}} r : T$

The correctness theorem for size-based termination can be expressed as follows:

Theorem 41 (Main theorem)
Suppose the conditions of definition 40 are satisfied. Then if $\mathcal{R}$ passes the size criterion we have for each $\Gamma, T \in \mathcal{T}$ and each term t:

$$\Gamma \vdash t : T \Rightarrow t \in \mathcal{SN}_{\mathcal{R}\cup\beta}$$

Notice that to the size-types are only needed to to establish that $\mathcal{R}$ passes the criterion, and that strong normalization is established for every well-typed term for simple (unlabeled) types.

The proof of this theorem is the object of the semantic analysis of the next sections.

Let us give an application of this termination technique.

Example 6 We take this example from Blanqui [Bla04]. Consider the signature $\Sigma = \{S, 0\} \cup \{minus, div\}$ with base type Nat and types of constants given by $0 \colon Nat^{\mathrm{s}(0)}, S : Nat^{\alpha} \to Nat^{\mathrm{s}(\alpha)}, minus \colon Nat^{\alpha} \to Nat^{\beta} \to Nat^{\alpha}, div \colon Nat^{\alpha} \to Nat^{\beta} \to Nat^{\infty}$. We set $div >_{\mathcal{D}} minus$ and give the rules:

$$\begin{array}{lllr}
minus\ 0 & n & \to & 0 \\
minus\ n & 0 & \to & n \\
minus\ (S\ \ n) & (S\ \ m) & \to & minus\ n\ m \\
div\ 0 & m & \to & 0 \\
div\ (S\ \ n) & (S\ \ m) & \to & S\ \ (div\ (minus\ n\ m)\ (S\ \ m))
\end{array}$$

Each rule can be typed in the context $n \colon Nat^{\gamma}, m \colon Nat^{\delta}$.

It is easy to verify that this rewrite system satisfies the termination criterion. We detail rule $div\ (S\ \ n)\ (S\ \ m) \to div\ (minus\ n\ m)\ (S\ \ m)$. We have

$$n \colon Nat^{\gamma}, m \colon Nat^{\delta} \vdash_{min} S\ \ n \colon Nat^{\mathrm{s}(\gamma)}$$

and

$$n \colon Nat^{\gamma}, m \colon Nat^{\delta} \vdash_{min} S\ \ m \colon Nat^{\mathrm{s}(\delta)}$$

We take for $stat_{div}$ the function that takes a well-founded order $>_E$ over E and returns the following well-founded order $>_{E \times E}$ over $E \times E$:

$$(e_1, e_2) >_{E \times E} (e'_1, e'_2) \Leftrightarrow e_1 >_E e'_1$$

We therefore only compare the size of the first argument of div in recursive calls. It is easy to check that

$$n \colon Nat^{\gamma}, m \colon Nat^{\delta} \vdash^{div}_{(\mathrm{s}(\gamma), \mathrm{s}(\delta))} div\ (minus\ n\ m)\ (S\ \ m) \colon Nat^{\infty}$$

Indeed

$$n \colon Nat^{\gamma}, m \colon Nat^{\delta} \vdash_{size} minus \colon Nat^{\gamma} \to Nat^{\delta} \to Nat^{\gamma}$$

And we can apply the **symb-call'** with the substitution that sends α to γ and β to δ. Then by applying the rule **app** we get

$$n \colon Nat^{\gamma}, m \colon Nat^{\delta} \vdash^{div}_{(\mathrm{s}(\gamma), \mathrm{s}(\delta))} minus\ n\ m \colon Nat^{\gamma}$$

But $\mathrm{s}(\gamma) > \gamma$, and so $(\mathrm{s}(\gamma), \mathrm{s}(\delta)) >_{div} (\gamma, \mathrm{s}(\delta))$. We can therefore apply **symb-call** with ϕ such that $\phi(\alpha) = \gamma$ and $\phi(\beta) = \mathrm{s}(\delta)$. Finally applying **app** again gives

$$n \colon Nat^{\gamma}, m \colon Nat^{\delta} \vdash^{div}_{(\mathrm{s}(\gamma), \mathrm{s}(\delta))} S\ \ (div\ (minus\ n\ m)\ (S\ \ m)) \colon Nat^{\infty}$$

The above system is not simply terminating. Our criterion gives semantic information, contained in the fact that *minus* returns a term smaller or equal in size to its first argument, encoded in the type $Nat^{\alpha} \to Nat^{\beta} \to Nat^{\alpha}$.

Here is an example with matching on defined symbols and higher-order constructors. Note that higher-order constructors can not be themselves defined, as they can not be both well-formed and in elimination form (see definitions 31 and 36).

Example 7 We take as base types Nat and $Prop$, and give the signature $\Sigma = \{S, 0, \neg, \wedge, \vee, \forall, \exists, \top, \bot\}$ with $C = \Sigma$ and $\mathcal{D} = \{\neg\}$ and we adopt an infix notation for $\wedge$ and $\vee$. We give the types $0 : Nat^{\mathrm{s}(0)}$, $S : Nat^{\alpha} \to Nat^{\mathrm{s}(\alpha)}$, $\top, \bot : Prop^{\mathrm{s}(0)}$, $\wedge, \vee : Prop^{\alpha} \to Prop^{\beta} \to Prop^{\mathrm{s}(\max(\alpha,\beta))}$, $\neg : Prop^{\alpha} \to Prop^{\mathrm{s}(\alpha)}$ and $\exists, \forall : Nat^{\infty} \to Prop^{\mathrm{s}(\alpha)}$. We consider the following rewrite rules:

$$\begin{array}{lcl}
\neg(x \wedge y) & \to & (\neg x) \vee (\neg y) \\
\neg(x \vee y) & \to & (\neg x) \wedge (\neg y) \\
\neg(\forall f) & \to & \exists(\lambda n : N. \neg f\ n) \\
\neg(\exists f) & \to & \forall(\lambda n : N. \neg f\ n) \\
\neg(\top) & \to & \bot \\
\neg(\bot) & \to & \top
\end{array}$$

This system represents the negation elimination, and it can be shown to be terminating using the size-types criterion. Set $\Gamma := x : Prop^{\alpha}, y : Prop^{\beta}$, and $\Gamma' = f : Nat^{\infty} \to Prop^{\alpha}$. We have

$$\Gamma \vdash_{min} \neg(x \wedge y), \neg(x \vee y) : Prop^{\mathrm{s}(\mathrm{s}(\max(\alpha,\beta)))}$$

$$\Gamma' \vdash_{min} \neg(\forall f), \neg(\exists f) : Prop^{\mathrm{s}(\mathrm{s}(\alpha))}$$

and

$$[] \vdash_{min} \neg(\top), \neg(\bot) : Prop^{\mathrm{s}(\mathrm{s}(0))}$$

Now we need to show that for every right hand side of the rewrite rules, each recursive call to $\neg$ is made on a smaller argument. For instance for the first rule we have

$$\Gamma \vdash_{min} x \wedge y : Prop^{\mathrm{s}(\max(\alpha,\beta))}$$

and

$$\Gamma \vdash^{\neg}_{\mathrm{s}(\max(\alpha,\beta))} (\neg x) \vee (\neg y) : Prop^{\mathrm{s}(\mathrm{s}(\max(\alpha,\beta)))}$$

And the other rules may be treated in the same manner.

Note that we may not add certain natural rules like $\neg(\neg(x)) \to x$ as this would break orthogonality.

In fact it is not entirely trivial to give an interesting example of an orthogonal system with defined constructors. A non-orthogonal system can sometimes be adapted to an orthogonal system though, by adding a non-defined constructor that *freezes* reduction.

Example 8 Take the set of base types $\mathcal{B} = \{Nat, List\}$ and the signature $C = \{S, 0, nil, cons, freeze, map\}$ and $\mathcal{D} = \{fold, map\}$ with the types

$$\begin{array}{lll} S & : & Nat^{\alpha} \to Nat^{\mathrm{s}(\alpha)} \\ 0 & : & Nat^{\mathrm{s}(0)} \\ nil & : & List^{\mathrm{s}(0)} \\ cons & : & Nat^{\infty} \to List^{\alpha} \to List^{\mathrm{s}(\alpha)} \\ freeze & : & List^{\alpha} \to List^{\mathrm{s}(\alpha)} \\ map & : & List^{\alpha} \to (Nat^{\infty} \to Nat^{\infty}) \to List^{\mathrm{s}(\alpha)} \\ fold & : & List^{\alpha} \to Nat^{\infty} \to (Nat^{\infty} \to Nat^{\infty} \to Nat^{\infty}) \to Nat^{\infty} \end{array}$$

And the rules

$$\begin{array}{lll} map\ nil\ f & \to & nil \\ map\ (cons\ x\ y)\ f & \to & cons\ (f\ x)\ (map\ y\ f) \\ fold\ nil\ a\ f & \to & a \\ fold\ (cons\ x\ y)\ a\ f & \to & f\ x\ (fold\ y\ a\ f) \\ fold\ (map\ (freeze\ x)\ f)\ a\ g & \to & fold\ x\ a\ (\lambda n{:}Nat.g\ (f\ n)) \end{array}$$

We may see this as a kind of optimization procedure, where the optimization of a *fold* over a *map* is only allowed to fire if the list to which is applied the map is *frozen* by the *freeze* constructor. The system is orthogonal, and it is easy to verify that it passes the termination criterion, using the precedence $fold >_{\mathcal{D}} map$.

Finally let us give a simple example of a system for which termination may not be shown using our criterion.

Example 9 (Non-local size information)

We take $\mathcal{B} = \{Nat\}$ and $\Sigma = \mathcal{D} \cup C$ with $C = \{S, 0\}$ and $\mathcal{D} = \{f\}$. We give the types $S : Nat^{\alpha} \to Nat^{\alpha}, 0 : Nat^{\mathrm{s}(0)}, f : Nat^{\alpha} \to Nat^{\infty}$ and as a rewrite system the unique rule

$$\mathcal{R} = \{f\ (S\ \ x) \to (\lambda z{:}Nat.f\ z)\ x$$

In the context $x : Nat^{\beta}$, we can derive

$$x : Nat^{\beta} \vdash_{min} f\ (S\ \ x) : Nat^{\infty}$$

and

$$x : Nat^{\beta} \vdash_{min} S\ \ x : Nat^{\mathrm{s}(\beta)}$$

However we can *not derive*

$$\vdash^{f}_{\mathrm{s}(\alpha)} (\lambda z{:}Nat^{\infty}.f\ z)\ x$$

Indeed in the context $x : Nat^{\alpha}, z : Nat^{\infty}$, $f\phi : Nat^{\infty} \to Nat^{\infty}$ implies $\phi(\alpha) = \infty$, and in particular we can not prove $\mathrm{s}(\beta) \not> \phi(\alpha)$.

It is not difficult to show that this rewrite system can not be shown to terminate (on well-typed terms) with β-reduction, for *any* possible annotation for f. The size information necessary to show termination is "hidden" by the β-expansion in the right-hand-side. However a similar system *can* be treated in the formalism presented by Barthe *et al* [BFG⁺04], as they have a more liberal rule for abstraction.

2

Higher Order Algebras

We define Σ_λ-algebras, which allow us to give a semantics to higher-order rewrite systems. We closely follow the developments of Fiore, Plotkin and Turi [FPT99] for the definitions of Σ_λ-algebras and $\bullet$-monoids, as well as that of Hamana [Ham07, Ham98] for premodels of rewrite systems, simply flushing out some of the categorical content. A good overview of this approach to higher order algebras can be found in Zsido [Zsi10]. The basic idea is to generalize the categorical description of first-order algebraic structures (see for example Vene [Ven00] for an overview), which can be made using endofunctors F on the category **Set**, and giving the notion of F-algebra. To be able to consider bound variables, the concept of variables must be reified, and we must now consider functors over the category $\mathbf{Set}^{\mathcal{G}}$ (category of presheafs), with $\mathcal{G}$ the category of contexts (of variables) and renamings. One can define the concept of Σ_l-algebra, and prove that it is possible to interpret terms t in $\mathcal{T}rm$ over a signature Σ into such an algebra, using an interpretation function $(\!|t|\!)$. It is then possible to describe a monoidal product $\bullet$ in $\mathbf{Set}^{\mathcal{G}}$ which captures the intuition of *instantiation*. We then consider Σ_λ-algebras which are also $\bullet$-monoids for this monoidal product. A final requirement is the compatibility of the two structures (called *strength*). When considering functors over the category **Pre** of preordered sets instead of just sets, we can describe *premodels* for a rewrite system $\mathcal{R} \cup \beta$, which are simply required to verify $(\!|t|\!) \geq (\!|u|\!)$ if $t \to^*_{\mathcal{R}\cup\beta} u$.

2.1 Categorical Basics

We will briefly give the basics of category theory necessary for the comprehension of this section, while referring to Mac Lane [Lan71] for details. As mentioned previously, we will not worry much about foundational concerns.

Definition 42 A *category* C is given by a class of *objects* Obj_C and for each pair of objects A, B a set of *morphisms* $C(A, B)$, which is endowed with the following structure:

- For each object $A \in Obj_C$, there is a morphism id_A
- For each $A, B, C \in Obj_C$, there is an application $\circ : C(A, B) \to C(B, C) \to C(A, C)$
- The following equalities hold:

1. $(f \circ g) \circ h = f \circ (g \circ h)$

2. $id_A \circ f = f$
3. $f \circ id_B = f$

for each objects A, B, C and each $f \in \mathcal{C}(A, B), g \in \mathcal{C}(B, C), h \in \mathcal{C}(C, D)$.

We will write $f\colon A \to B$ for $f \in \mathcal{C}(A, B)$. From now on if we write $f \circ g$ we suppose that there are A, B and C such that $f \in \mathcal{C}(A, B)$ and $g \in \mathcal{C}(B, C)$

Given a pair of categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* F associates to each $A \in Obj_{\mathcal{C}}$ an object $FA \in Obj_{\mathcal{D}}$ and to each $f \in \mathcal{C}(A, B)$ a morphism $Ff \in \mathcal{D}(FA, FB)$, while respecting the equalities:

- $Fid_A = id_{FA}$
- $F(f \circ g) = (Ff) \circ (Fg)$

A functor $F\colon \mathcal{C} \to \mathcal{C}$ is called an *endofunctor*.

Given two functors F and G between $\mathcal{C}$ and $\mathcal{D}$, a *natural transformation* $\eta\colon F \to G$ is an operation that associates to each A of $\mathcal{C}$ a map $\eta_A\colon FA \to GA$ such that for all $f\colon A \to B$

$$Gf \circ \eta_A = \eta_B \circ Ff$$

In this case we say that *the following diagram commutes*:

$$\begin{tikzcd}
FA \arrow[r, "\eta_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\eta_B"'] & GB
\end{tikzcd}$$

Given a category $\mathcal{C}$ and two objects A, B we say that A and B are *in isomorphism* or *isomorphic* if there exist f and g such that $g \circ f = id_A$ and $f \circ g = id_B$.

For a given category $\mathcal{C}$ a *final object* $1_{\mathcal{C}}$ is an object of $\mathcal{C}$ such that for any object $A \in Obj_{\mathcal{C}}$, there exists a unique morphism $!_A\colon A \to 1_{\mathcal{C}}$. Final objects are unique up to unique isomorphism.

We may define a dual notion of *initial object* $0_{\mathcal{C}}$ such that for every object A there exists a unique morphism $\phi_A\colon 0_{\mathcal{C}} \to A$.

For any given category $\mathcal{C}$ an object C is a *product* of A and B if:

1. there exist $\pi_1\colon C \to A$ and $\pi_2\colon C \to B$.
2. the *universal property* holds: for each D and $f\colon D \to A$, $g\colon D \to B$ there exists a unique morphism $\langle f, g\rangle\colon D \to C$ such that the following diagram commutes:

$$\begin{tikzcd}
 & D \arrow[d, "\langle f,g\rangle"] \arrow[ddl, "f"'] \arrow[ddr, "g"] & \\
 & C \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
A & & B
\end{tikzcd}$$

Such an object is unique up to unique isomorphism and we sometimes write $A \times B$ for such an object. Note that given $f : A \to A'$ and $g : B \to B'$, if $A \times B$ and $A' \times B'$ exist, it is possible to build an application $f \times g : A \times B \to A' \times B'$ defined by $f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle$.

In the same way we define a dual notion of *coproduct*: for A and B, a coproduct $A + B$ is an object along with morphisms $\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$, such that, for any C and $f : A \to C$, $g : B \to C$ there is a unique $f \sqcup g : A + B \to C$ such that

$$\begin{tikzcd}
A \arrow[dr, "\iota_1"] \arrow[ddr, "f"'] & & B \arrow[dl, "\iota_2"'] \arrow[ddl, "g"] \\
& A + B \arrow[d, "f \sqcup g"] & \\
& C &
\end{tikzcd}$$

commutes, we similarly define $f + g$.

We say that a category C is *cartesian* if there exists an initial object 1_C and for each pair of objects A and B there is a product object $A \times B$.

For a given cartesian category C and an object A of C, we have the *right product functor* $\_ \times A$, defined by its action on objects B by $B \times A$ and on applications $f : B \to B'$ as $f \times id_A$. It is easy to verify that the functor properties hold. One can similarly define the left product functor $A \times \_$.

The motivational example of a category is the category **Set**, defined as the category with objects equal to the class of all sets, and as morphisms between two sets A and B as the set of all applications between A and B (written B^A). It is easy to verify that this is a category. Furthermore **Set** is cartesian, its final object given by a set with one element and its products given by the usual cartesian product of two sets.

Given two categories C and $\mathcal{D}$, the *functor category* $\mathcal{D}^C$ is the category which has as objects the functors between C and $\mathcal{D}$ and as morphisms the natural transformations between functors.

Given a set E, we may construct the *trivial category* in which the objects are the elements $e \in E$ and the set $E(e, e')$ is defined by a singleton set if $e = e'$ and the empty set otherwise, taking the identity morphism to be the unique element. All the above requirements are trivially satisfied.

We describe how to construct a notion of *higher order algebra* as a semantics for the rewrite systems described in the previous section. To understand the categorical notion of algebra, we shall start with an example of a first order algebra: the algebra of natural numbers.

Example 10 (The algebra of natural numbers)

Consider the final object $1 = \{\emptyset\}$ of the category **Set**. We define N to be the functor from **Set** to **Set** defined on objects by $NA = 1 + A$, and where the action on morphisms is defined by $Nf = 1 + f$. A N-algebra is then an object A along with a morphism $\mathsf{nat}_A : NA \to A$. We first observe that the class of all N-algebras is a category with the following notion of morphism: a morphism between N-algebras A and B is a morphism $\phi : A \to B$ in the category **Set** such that the following diagram commutes:

$$\begin{tikzcd}
NA \arrow[r, "N\phi"] \arrow[d, "\mathrm{nat}_A"'] & NB \arrow[d, "\mathrm{nat}_B"] \\
A \arrow[r, "\phi"'] & B
\end{tikzcd}$$

Now take $\mathbb{N}$ to be the set of natural numbers. There are two morphisms $z\colon 1 \to \mathbb{N}$ and $s\colon \mathbb{N} \to \mathbb{N}$, defined by $z(\emptyset) = 0$ and $s(n) = n + 1$, which endow $\mathbb{N}$ with a structure of N-algebra. Furthermore observe that $\mathbb{N}$ endowed with these two maps is *initial* in the category of N-algebras. Let us prove this: let A be a N-algebra, with the structure given by the morphism nat_A. It is easy to show that nat_A can be decomposed as $1_a \sqcup f$ where $1_a\colon 1 \to A$ is a constant function $1_a(\emptyset) = a \in A$, and $f\colon A \to A$. We may then define the following function $\phi_A\colon \mathbb{N} \to A$ by induction on $\mathbb{N}$:

$$\begin{aligned}
\phi_A(0) &\mapsto a \\
\phi_A(n) &\mapsto f(\phi_A(n-1))
\end{aligned}$$

We first verify that this is indeed an N-algebra morphism, that is show that $\phi \circ \mathrm{nat}_{\mathbb{N}} = \mathrm{nat}_A \circ N\phi$. This amounts to checking:

$$\phi \circ \iota_1(\emptyset) = \phi(0) = 1_a \circ\ id_1 \circ \iota_1(\emptyset) = a$$

and

$$\phi \circ \iota_2(n-1) = \phi(n) = f \circ \phi \circ \iota_2(n-1) = f(\phi(n-1))$$

Both of which are true by definition of ϕ.

We must also show uniqueness: let ψ be another morphism from the algebra $\mathbb{N}$ to A. We prove by induction on n that $\forall n, \phi(n) = \psi(n)$:

- $n = 0$. We have by definition of a morphism $\psi \circ \mathrm{nat}_{\mathbb{N}} = \mathrm{nat}_A \circ N\psi$ but

  $$\psi \circ \mathrm{nat}_{\mathbb{N}} \circ \iota_1(\emptyset) = \psi \circ 1_0(\emptyset) = \psi(0)$$

  On one hand and

  $$\mathrm{nat}_A \circ N\psi \circ \iota_1(\emptyset) = \mathrm{nat}_A \circ id_1 \circ \iota_1(\emptyset) = 1_a(\emptyset) = a$$

  on the other, from which we conclude $\psi(0) = a$.

- By induction $\phi(n-1) = \psi(n-1)$. Again we consider the equality $\psi \circ \mathrm{nat}_{\mathbb{N}} = \mathrm{nat}_A \circ N\psi$. We have in the same manner:

  $$\psi \circ \mathrm{nat}_{\mathbb{N}} \circ \iota_{\mathbb{N}}(n-1) = \psi((n-1)+1) = \psi(n)$$

  and

  $$\mathrm{nat}_A \circ N\psi(n-1) = f \circ \psi(n-1) = f(\psi(n-1))$$

  So that $\psi(n) = f(\psi(n-1))$ by induction hypothesis we have

  $$\psi(n) = f(\phi(n-1)) = \phi(n)$$

■

Note the use of induction in both the definition of ϕ and the proof of uniqueness. Indeed, the fact that $\mathbb{N}$ is initial in the category of N-algebras can be seen as a reformulation of the statement that it is possible to define functions by induction on the standard structure of $\mathbb{N}$: to define a function from $\mathbb{N}$ to A, it suffices to identify $a \in A$ and $f : A \to A$, and then appeal to the existence of a morphism ϕ between $\mathbb{N}$ and the resulting N-algebra.

2.2 Presheaf Algebras and Substitution Monoids

The example above gives us a clue about how to treat algebraic structures using category theory: instead of looking at sets with constructors and equations (or rewrite rules), we consider a suitable functor over **Set** and the associated category of algebras over that functor. Then the key insight is this: if instead of taking **Set** as the base category we take some category which is "aware" of free variables, we can define a notion of *higher order* algebra.

Definition 43 (Typed algebras) A *presheaf* on a category C is a functor from C to **Set**.

The *category of contexts* $\mathcal{G}$ is defined as the category with as objects contexts Γ on simple types $\mathcal{T}$, and as morphisms the functions $\iota : \mathrm{dom}(\Gamma) \to \mathrm{dom}(\Gamma')$ that preserve types,*i.e.* $\Gamma(x) = \Gamma'(\iota(x))$ for all x in the domain of Γ.

The *category of typed algebras* $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ is defined as the category of functors from the trivial category $\mathcal{T}$ of types to presheafs on $\mathcal{G}$. Given a presheaf $F \in \mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, a type T and a context Γ, we write $F_T(\Gamma)$ instead of $F(T)(\Gamma)$.

To be able to build the higher order algebras we are interested in, we need some more categorical structures.

Definition 44 Let C be a category, I be a set and $(A_i)_{i\in I}$ be an I-indexed family of objects of C. The *product* of the family $(A_i)_{i\in I}$ is an object $\prod_{i\in I} A_i$ and some morphisms $(\pi_i)_{i\in I}$ such that

$$\pi_i : \prod_{i\in I} A_i \to A_i$$

and with the *universal property*: for every object C such that there exists a family $(f_i)_{i\in I}$, $f_i : C \to A_i$, there exists a unique morphism $\langle f_i \mid i \in I\rangle : C \to \prod_{i\in I} A_i$ such that the triangle

$$\begin{tikzcd}
C \arrow[dr, "f_i"] \arrow[dd, "\langle f_i|i\in I\rangle"'] & \\
& A_i \\
\prod_{i\in I} A_i \arrow[ur, "\pi_i"'] &
\end{tikzcd}$$

commutes for each i.

Notice that for a 2-element family $(A_i)_{i\in\{1,2\}}$, the product $\prod_{i\in\{1,2\}} A_i$ is isomorphic to $A_1 \times A_2$. We define dually the coproduct $\sum_{i\in I} A_i$ of a family $(A_i)_{i\in I}$ of objects, equipped with morphisms $\iota_i : A_i \to \sum_{i\in I} A_i$, that satisfies the dual universal property, we write $\bigsqcup_{i\in I} f_i$ for the morphism from $\sum_{i\in I} A_i$ to B if $f_i : A_i \to B$ is a family of morphisms.

We say a category C *admits small products* (resp. *admits small coproducts*) if for every family of objects in C indexed by a *set*, the product of that family exists.

The following definition shows that $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ has the necessary structure to build the algebraic framework.

Property 45 The category $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ admits small products and coproducts.

Proof. The proof may be found in Mac Lane [Lan71]. The products and coproducts are just defined point-wise from those in **Set**.

We may define two presheaves which show that the category of functors in $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ is appropriate to speak of the algebra of terms.

Definition 46 (The presheaf of terms)

We define $\mathsf{dom} \in \mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ as the functor which takes a type T and a context Γ and returns the set of all variables x of type T in Γ, and the action on morphisms $\iota : \Gamma \to \Gamma'$ is just ι, that is $\mathsf{dom}(\iota) = \iota$. This functor can be seen as the functor that witnesses the Yoneda lemma.

Given a signature Σ, we define the functor $\mathcal{T}rm^{\Sigma}$ by

$$\mathcal{T}rm^{\Sigma}_{T}(\Gamma) = \{t \in \mathcal{T}rm^{\Sigma} \mid \Gamma \vdash t : T\}$$

With the action on arrows $\iota : \Gamma \to \Gamma'$ to be the renaming function:

$$\mathcal{T}rm^{\Sigma}_{T}(\iota)(t) = t\{\vec{x} \mapsto \iota(\vec{x})\}$$

It is easy to verify that $t\{\vec{x} \mapsto \iota(\vec{x})\}$ is indeed in $\mathcal{T}rm^{\Sigma}_{T}(\Gamma')$, by induction on the typing derivation $\Gamma \vdash t : T$, using the fact that every free variable in t is in the domain of Γ. If $\Gamma' = \Gamma, y_1 : U_1, \ldots, y_m : U_m$ and $\iota : \Gamma \to \Gamma'$ is the inclusion morphism, then $\mathcal{T}rm^{\Sigma}_{T}(\iota)(t)$ is called the *weakening* of t and sometimes written $\iota(t)$ or just t.

There is a natural transformation from dom to $\mathcal{T}rm^{\Sigma}$, which for each object Γ is just the inclusion $\mathsf{dom}_T(\Gamma) \to \mathcal{T}rm_T(\Gamma)$ which sends a variable to its representation as a term, using the fact that by rule **ax** if $\Gamma(x) = T$ then $\Gamma \vdash x : T$.

From here on we fix a signature Σ and an arity function $ar : \Sigma \to \mathcal{T}$. We define the signature functor Σ defined by

$$\Sigma_T(\Gamma) = \{f \in \Sigma \mid ar(f) = T\}$$

Σ acts on morphisms trivially, *i.e.* sends all morphisms to the identity, as $\Sigma_T(\Gamma)$ does not depend on Γ.

We can define a notion of algebra on $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ in a very similar way as example 10. The main difference, of course, involves abstraction, and we will define an *abstraction transformation*, which given a presheaf, a context Γ and a type, considers the set of elements in that presheaf with respect to Γ to which we add an extra *bound* variable x.

We use the Barendregt convention and suppose that x is not in the domain of Γ, as in the contrary case we can choose some Γ' that is isomorphic to Γ such that $x \notin \mathsf{dom}(\Gamma')$. This choice of presentation is somewhat of a departure from the usual presentation of binding algebras, which tend to have numbered variables, and giving a presentation of higher-order algebras using *De Bruijn levels*. Again the named version of the presentation may be formalized by appeal to *FM-sets*, which form the theoretical basis of the *nominal-logic* approach. We again refer to [Gab07], and will not worry about machine formalization here: $F^{x:A}$ is seen as a *notation* rather than a strict functor.

Definition 47 (Σ_λ-algebra)

For each functor $F \in \mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, variable x and type $T \in \mathcal{T}$, we define the *abstraction* $F^{x:T}$ of F to be the functor defined by:

$$F^{x:T}_U(\Gamma) \coloneqq F_U(\Gamma, x\colon T)$$

With $x \notin \mathrm{dom}(\Gamma)$.

The functor $F^{x:T}$ acts on arrows $\iota\colon \Gamma \to \Gamma'$ by $F^{x:T}_U(\iota)(\Gamma) = F(\iota')_U(\Gamma, x\colon T)$ where ι' is the function from $\Gamma, x\colon T$ to $\Gamma', x\colon T$ which is equal to ι on Γ and sends x to x.

We can now define the endo-functor Σ_λ by case on the types:

for a base type B,

$$\Sigma_\lambda(F)_B(\Gamma) = \Sigma_B(\Gamma) + \sum_{U \in \mathcal{T}} F_{U \to B}(\Gamma) \times F_U(\Gamma)$$

and for any types T and U,

$$\Sigma_\lambda(F)_{T\to U}(\Gamma) = \Sigma_{T\to U}(\Gamma) + F^{x:T}_U(\Gamma) + \sum_{V \in \mathcal{T}} F_{V\to T\to U}(\Gamma) \times F_V(\Gamma)$$

We define the $V + \Sigma_\lambda$ functor as

$$(V + \Sigma_\lambda)_T(\Gamma) = \mathrm{dom}_T(\Gamma) + (\Sigma_\lambda)_T(\Gamma)$$

Where V stands for Variable, and we adopt the notation adopted by Fiore *et al* and Hamana.

Given a functor $T\colon \mathbf{Set}^{\mathcal{G}}_{\mathcal{T}} \to \mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, a T-algebra is a presheaf $A \in \mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ equipped with a morphism (of presheafs) $ev_A\colon TA \to A$.

The category Alg_T is the category with as objects T-algebras and as morphisms the set of morphisms f in $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$ such that the following square commutes:

$$\begin{tikzcd}
TA \arrow[r, "Tf"] \arrow[d, "ev_A"'] & TB \arrow[d, "ev_B"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

Note that all $V + \Sigma_\lambda$-algebras are in particular Σ_λ-algebras.

A Σ_λ-algebra A is intuitively equipped with three *operations* which consist of: an element of A_T for each $f \in \Sigma$ of type T, an application which is a morphism from $A_{U\to T} \times A_U$ to A_T for each U, and an abstraction which is a morphism from $A^{x:T}_U$ to $A_{T\to U}$. A $V + \Sigma_\lambda$-algebra supplies the additional structure of free variables in Γ, which is essential for describing the semantics of terms.

As hoped, the presheaf $\mathcal{T}rm^\Sigma$ is indeed the initial object in the category of such algebras.

Theorem 48 (Term algebra)

The presheaf $\mathcal{T}rm^\Sigma$ is a $V + \Sigma_\lambda$-algebra and is initial in the category of $V + \Sigma_\lambda$-algebras.

Proof.

To define the morphism $ev_{\mathcal{T}rm}\colon V + \Sigma_\lambda \mathcal{T}rm^\Sigma \to \mathcal{T}rm^\Sigma$, it suffices to define a morphism

$$ev^T_{\mathcal{T}rm}\colon (V + \Sigma_\lambda \mathcal{T}rm^\Sigma)_T \to \mathcal{T}rm^\Sigma_T$$

for each T. We proceed by cases:

- $T = B$ a base type. By the universal properties of coproducts, it suffices to build morphisms $var\colon \mathsf{dom}_B \to \mathcal{T}rm^{\Sigma}_B$, $val\colon \Sigma_B \to \mathcal{T}rm^{\Sigma}_B$ and $app^U_B\colon \mathcal{T}rm^{\Sigma}_{U\to B} \times \mathcal{T}rm^{\Sigma}_U \to \mathcal{T}rm^{\Sigma}_B$ for each $U \in \mathcal{T}$. For the first, we choose the inclusion function defined earlier. For the second, we take the inclusion $\Sigma_B \subseteq \mathcal{T}rm^{\Sigma}_B$. Indeed, for every $f \in \Sigma_B$ and context Γ, we have $\Gamma \vdash f\colon B$ by rule **symb**. For app^U_B, we take the function which for each Γ, and pair $(t, u) \in \mathcal{T}rm^{\Sigma}_{U\to B}(\Gamma) \times \mathcal{T}rm^{\Sigma}_U(\Gamma)$ associates the term $t\ u$. By rule **app**, it is easy to check that $t\ u$ is in $\mathcal{T}rm^{\Sigma}_B$.

- Case $T \to U$. It suffices to build morphisms $var\colon \mathsf{dom}_{T\to U} \to \mathcal{T}rm^{\Sigma}_{T\to U}$, $val\colon \Sigma_{T\to U} \to \mathcal{T}rm^{\Sigma}_{T\to U}$, $app^V_{T\to U}\colon \mathcal{T}rm^{\Sigma}_{V\to T\to U} \times \mathcal{T}rm^{\Sigma}_V \to \mathcal{T}rm^{\Sigma}_{T\to U}$ for every $V \in \mathcal{T}$ and a morphism $abs^T_U\colon (\mathcal{T}rm^{\Sigma})^{x\colon T}_U \to \mathcal{T}rm^{\Sigma}_{T\to U}$ for some (fresh) variable x. In the first three cases we proceed exactly as above. In the forth case, given a context Γ, we may suppose by Barendregts convention that x is not in the domain of Γ. By definition, $(\mathcal{T}rm^{\Sigma})^{x\colon T}_U(\Gamma) = \mathcal{T}rm^{\Sigma}_U(\Gamma, x\colon T)$, and given a term t in that set we may apply the **abs** rule to build the term $\lambda x\colon T.t$ in the set $\mathcal{T}rm^{\Sigma}_{T\to U}(\Gamma)$.

It is now necessary to check that this indeed defines a presheaf morphism, that is that the functions between sets define a natural transformation. This reduces to checking stability with respect to "renaming" in contexts, which we may do for each component of the morphism:

- It is already established that the inclusion $\mathsf{dom} \to \mathcal{T}rm^{\Sigma}$ is a natural transformation.

- The inclusion function $\Sigma_T \subseteq \mathcal{T}rm_T$ clearly leads to a natural transformation, indeed there is nothing to check as the action of Σ_T on morphisms is trivial.

- for the app^U_V we need to check that $(t\ u)\{\vec{x} \mapsto \iota(\vec{x})\} = t\{\vec{x} \mapsto \iota(\vec{x})\}\ u\{\vec{x} \mapsto \iota(\vec{x})\}$ this follows by definition of substitution.

- for the function abs^T_U we need to check that $(\lambda x\colon T.t)\{\vec{x} \mapsto \iota(\vec{x})\} = \lambda x\colon T.(t\{\vec{x} \mapsto \iota'(\vec{x})\})$. By the Barendregt convention, we may suppose that $x \neq x_j$ for all j. We can then again conclude by definition of substitution.

We then need to check that $\mathcal{T}rm^{\Sigma}$ is initial. We proceed as in example 10, using structural induction on terms. So let A be a $V + \Sigma_\lambda$-algebra. The algebra structure gives the morphisms $var_A\colon \mathsf{dom} \to A$, $val_A\colon \Sigma \to A$, for each base type T a family of morphisms $(app(A)^U_T)_{U\in\mathcal{T}}$ such that $app(A)^U_T\colon A_{U\to T} \times A_U \to A_T$, and for each pair of types T and U and variable x, a morphism $abs(A)^T_U\colon A^{x\colon T}_U \to A_{T\to U}$. Let t be some well-typed term, that is there exist Γ and T such that $\Gamma \vdash t\colon T$. In particular, $t \in \mathcal{T}rm^{\Sigma}_T(\Gamma)$. We build an element $\phi(t)^{\Gamma}_T \in A_T(\Gamma)$ by induction on the derivation of $\Gamma \vdash t\colon T$

- $t = x$ a variable. As t is well-typed, x is in the domain of Γ, and so $x \in \mathsf{dom}_T(\Gamma)$. We can therefore define $\phi^{\Gamma}_T(x) = var(x)$.

- $t = f$ an element of Σ_T. In the same manner as above, we take $\phi(f)^{\Gamma}_T$ to be $val(A)(f)$.

- $t = u\ v$. We have $\Gamma \vdash u\colon U \to T$ and $G \vdash v\colon U$. We define

$$\phi^{\Gamma}_T(t) = app(A)^{T_1}_T(\phi^{\Gamma}_{U\to T}(u), \phi^{\Gamma}_U(v))$$

- $t = \lambda x\colon U.u$. We have $\Gamma, x\colon U \vdash u\colon V$ and $T = U \to V$. We can then define

$$\phi^{\Gamma}_T(t) = abs(A)^U_V(\phi^{\Gamma, x\colon U}_V(u))$$

We need to show that ϕ is indeed a morphism of Σ_λ-algebras, that is

$$ev_A \circ \Sigma_\lambda \phi = \phi \circ ev_{\mathcal{T}rm^\Sigma}$$

and that this morphism is unique. The proof is quite similar to that of example 10, and we refer to Hamana [Ham07] for the proof. ■

The presheaf algebras give us a satisfactory treatment of higher-order algebras. However, to be able to define a semantics in the hope of analyzing termination, we need to integrate the notion of reduction. We do this by introducing preordered T-algebras. Preordered algebras are defined in exactly the same manner as ordinary algebras, but by using the base category of *preordered sets* instead of sets.

Definition 49 Let **Pre** be the category of sets equipped with a preorder $(E, \geq)$, with as morphisms order preserving (monotone) maps. As usual we identify preordered sets with the underlying carrier.

We define the category $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ by taking the functors from the trivial category $\mathcal{T}$ to the category of functors from $\mathcal{G}$ to **Pre**, which we shall also call presheafs. Notice that for every presheaf F in $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, there is a presheaf $\tilde{F}$ in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ defined for each context Γ and type T by

$$\tilde{F}_T(\Gamma) = (F_T(\Gamma), \geq_{triv})$$

Where $\geq_{triv}$ is the trivial preorder defined by $x \geq_{triv} y \Leftrightarrow x = y$.

The category $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ has all small products and coproducts.

We may define Σ_λ-algebras and $V + \Sigma_\lambda$-algebras in the same manner as definition 47.

Given a set of left-algebraic rewrite rules $\mathcal{R}$ over the signature Σ, we can see $\mathcal{T}rm^\Sigma$ equipped with the relation $\mathcal{R} \cup \beta$ as a $V + \Sigma_\lambda$-algebra.

Property 50 Let $\mathcal{R}$ be a set of left-algebraic rewrite rules and define $\mathcal{T}rm^\Sigma_\mathcal{R}$ be the functor in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ defined on objects as:

$$(\mathcal{T}rm^\Sigma_\mathcal{R})_T(\Gamma) = (\mathcal{T}rm^\Sigma_T(\Gamma), \to^*_{\mathcal{R}\cup\beta})$$

and on morphisms as

$$(\mathcal{T}rm^\Sigma_\mathcal{R})_T(\iota) = (\mathcal{T}rm^\Sigma)_T(\iota)$$

The functor $\mathcal{T}rm^\Sigma_\mathcal{R}$ carries a structure of Σ_λ-algebra. In the following we suppose that Σ and $\mathcal{R}$ are fixed, and write $\mathcal{T}rm$ for $\mathcal{T}rm^\Sigma_\mathcal{R}$.

To show that $\mathcal{T}rm$ is well defined we need a classic lemma:

Lemma 51 $\mathcal{T}rm$ enjoys *subject reduction*: for any context Γ, type T and t, u terms, if $\Gamma \vdash t : T$ and $t \to^*_{\mathcal{R}\beta} u$ then $\Gamma \vdash u : T$.

The proof is classic, and we shall omit it.

Now we may prove the property 50:

Proof.

First we show that $\mathcal{T}rm$ is a functor. From subject reduction, we have that $\to^*_{\mathcal{R}\beta}$ is indeed an order on $\mathcal{T}rm_T(\Gamma)$ for every T and Γ. We need to show that for each T and $\iota : \Gamma \to \Gamma'$, $\mathcal{T}rm_T(\iota)$

is a morphism between $\mathcal{T}rm_T(\Gamma)$ and $\mathcal{T}rm_T(\Gamma')$. We have that it is a function, we only need to show that it is monotonous, that is if $t \to^*_{\mathcal{R}\beta} u$, then

$$t\{\vec{x} \mapsto \iota(\vec{x})\} \to^*_{\mathcal{R}\beta} u\{\vec{x} \mapsto \iota(\vec{x})\}$$

This can be shown by an easy induction on the number of steps, and then on rule application position.

We define the $ev_{\mathcal{T}rm}$ morphism as the $ev_{\mathcal{T}rm^\Sigma}$ morphism on $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, and prove that $(ev_{\mathcal{T}rm^\Sigma})^\Gamma_T(t) \geq (ev_{\mathcal{T}rm^\Sigma})^\Gamma_T(u)$ for all $t \geq u$ in $(\Sigma_\lambda \mathcal{T}rm)_T(\Gamma)$. It suffices to check this on each component of $ev_{\mathcal{T}rm^\Sigma}$:

- val^Γ_T: we need to prove that for each f, g in Σ_T, if $f \geq g$ in Σ_T then $f \to^*_{\mathcal{R}\cup\beta} g$ in $\mathcal{T}rm^\Sigma_T$. But the preorder on Σ_T is just the trivial order, so $f = g$.
- var^Γ_T: as above, the preorder on the variables in Γ of type T is the trivial order.
- app^U_T: take terms t, t', u, u' such that $\Gamma \vdash t, t' : U \to T$ and $\Gamma \vdash u, u' : U$. We have $(t, u) \geq (t', u')$ or equivalently $t \to^*_{\mathcal{R}\cup\beta} t'$ and $u \to^*_{\mathcal{R}\cup\beta} u'$. We need to show that $app^U_T(t, u) = t\ u \to^* t'\ u' = app^U_T(t', u')$. But this is true by the congruence rule of definition 11.
- abs^U_V: In this case, $T = U \to V$, we may proceed exactly as above.

■

The proposition follows quite easily from the definition of rewriting as a congruence. In fact, working with algebras in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ is just a way of requiring that the preorders on the components are compatible with the algebraic structure, and renaming of the variables. The notion of reduction is taken into account by working over the category of preorders. To adequately model rewriting however, we need to take into account the notion of substitution. Indeed, the fundamental properties of the rewriting preorder is that it is closed under contexts, which is handled by the notion of Σ_λ-algebra on **Pre**, and by substitution. To give a categorical account of substitution, we introduce the notion of *monoidal product*. In fact, we shall only consider one example of monoidal product and for a general treatment of the notion we shall refer the reader to Mac Lane [Lan71].

Given two functors A, B in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$, a context Δ and a type T, the set $(A \bullet B)_T(\Delta)$ can intuitively be seen as the set of pairs of an element $a \in A_T(\Gamma)$ for some context Γ and a *substitution* θ of the variables in a by elements of $B_U(\Delta)$, if the variable is of type U, taken modulo renaming of variables in a. To actually perform this substitution, we need a function subst such that $subst(a, \theta) \in A_T(\Delta)$. This can usually only make sense if $A = B$, and this leads us to use the notion of *monoid object* to make this precise.

In what follows, if $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ and $\Gamma' = y_1 : U_1, \ldots, y_m : U_m$, if $\iota : \Gamma \to \Gamma'$ we write $\iota(i)$ for the number j such that $y_j = \iota(x_i)$.

Definition 52 (Monoidal product)
Given three categories $\mathcal{C}, \mathcal{D}, \mathcal{E}$, a *bifunctor* $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ associates to each pair of objects $A \in \mathcal{C}$ and $B \in \mathcal{D}$ an object $F(A, B)$ of $\mathcal{E}$, and if $f : A \to A'$ and $g : B \to B'$ are morphisms, then $F(f, g)$ is a morphism between $F(A, B)$ and $F(A', B')$, such that F respects composition and identity in both components.

Given a category C a *monoidal product* $\otimes$ on C is a bifunctor $\_ \otimes \_ \colon C \times C \to C$ and an *unit object* I with three natural isomorphisms:

$$\alpha_{A,B,C} \colon A \otimes (B \otimes C) \to (A \otimes B) \otimes C$$

$$\lambda_A \colon I \otimes A \to A \qquad \rho \colon A \otimes I \to A$$

Satisfying the additional *coherence conditions*:

$$\alpha_{A \otimes B,C,D} \circ \alpha_{A,B,C \otimes D} = \alpha_{A,B,C} \otimes id_D \circ \alpha_{A,B \otimes C,D} \circ id_A \otimes \alpha_{B,C,D}$$

$$\rho_A \otimes id_B \circ \alpha_{A,I,B} = id_A \otimes \lambda_B$$

and

$$\lambda_I = \rho_I$$

For all objects A, B, C, D.

We define the *substitution monoidal product* $\_ \bullet \_ \colon \mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}} \times \mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}} \to \mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ on objects by the quotient:

$$(A \bullet B)_T(\Delta) = \left(\sum_{\Gamma \in \mathcal{G}} A_T(\Gamma) \times \prod_{x \in \mathrm{dom}(\Gamma)} B_{\Gamma(x)}(\Delta) \right) / \simeq$$

Where $\simeq$ is the symmetric transitive closure of the relation $\sim$:

$$(a, b_{\iota(1)}, \ldots, b_{\iota(n)}) \sim (A(\iota)(a), b_1, \ldots, b_m)$$

if $\mathrm{dom}(\Gamma) = x_1, \ldots, x_n$, $\mathrm{dom}(\Gamma') = y_1, \ldots, y_m$ and $\iota \colon \Gamma \to \Gamma'$. We give an order structure on $(A \bullet B)_T(\Delta)$ by taking the transitive closure of the rule:

$$\frac{a \geq a' \wedge b_1 \geq b'_1, \ldots, b_n \geq b'_n}{\overline{(a, b_1, \ldots, b_n)} \geq \overline{(a', b'_1, \ldots, b'_n)}}$$

if there is some Γ such that $a, a' \in A_T(\Gamma)$ and $(b_1, \ldots, b_n), (b'_1, \ldots, b'_n)$ are in $\Pi_{x \in \mathrm{dom}(\Gamma)} B_{\Gamma(x)}(\Delta)$. Therefore in order to prove that $m \geq n$ in $(A \bullet B)_U(\Delta)$ one must find a *common* Γ and T such that $m = \overline{(a, \vec{b})}$ and $n = \overline{(a', \vec{b}')}$ with $a, a' \in A_T(\Gamma)$, to then compare a and a' and $\vec{b}$ and $\vec{b}'$.

We can verify that this definition respects the relation $\sim$. If $\iota \colon \Gamma \to \Gamma'$ is an arrow, then $a \geq a' \Rightarrow A(\iota)(a) \geq A(\iota)(a')$ by functoriality of A, and of course $(b_1, \ldots, b_n) \geq (b'_1, \ldots, b'_n) \Rightarrow (b_{\iota(1)}, \ldots, b_{\iota(n)}) \geq (b'_{\iota(1)}, \ldots, b'_{\iota(n)})$.

On functions $j \colon \Delta \to \Delta'$, $(A \bullet B)(j)$ is defined as the quotient of the function $\sum_{\Gamma \in \mathcal{G}} A_T(\Gamma) \times \prod_{x \in \mathrm{dom}(\Gamma)} B_{\Gamma(x)}(j)$. That is, if $(a, b_1, \ldots, b_n)$ is in $A_T(\Gamma) \times \Pi B(\Delta)$ then

$$(A \bullet B)(j)(\overline{(a, \vec{b})}) = \overline{(a, B(j)(b_1), \ldots, B(j)(b_n))}$$

It is easy to check that this definition is independent of the choice of representative: if $\iota \colon \Gamma \to \Gamma'$, then

$$(a, B(j)(b_1), \ldots, B(j)(b_n)) \sim (A(\iota)(a), B(j)(b_{\iota(1)}), \ldots, B(j)(b_{\iota(n)}))$$

As unit object we take the presheaf dom.

For the proof that $\_ \bullet \_$ is well defined on morphisms (that the action on morphisms is indeed stable by the equivalence relation), that it is indeed a bifunctor, and that it is a monoidal product with identity dom, we refer to Zsido [Zsi10].

Definition 53 (Monoid object)

Given a category C and a monoidal product $\otimes\colon C \times C \to C$ with unit I, a *monoid object* M of $\otimes$ (or $\otimes$-monoid) is an object M of C equipped with two morphisms $\mu\colon M \otimes M \to M$ and $\eta\colon I \to M$, such that the following diagrams commute:

$$\begin{tikzcd}
M \otimes (M \otimes M) \arrow[r, "\alpha_{M,M,M}"] \arrow[d, "id_M \otimes \mu"'] & (M \otimes M) \otimes M \arrow[r, "\mu \otimes id_M"] & M \otimes M \arrow[d, "\mu"] \\
M \otimes M \arrow[rr, "\mu"] & & M
\end{tikzcd}$$

$$\begin{tikzcd}
I \otimes M \arrow[r, "\eta \otimes id_M"] \arrow[dr, "\lambda"'] & M \otimes M \arrow[d, "\mu"] & M \otimes I \arrow[l, "id_M \otimes \eta"'] \arrow[dl, "\rho"] \\
& M &
\end{tikzcd}$$

We will say *monoid* instead of $\otimes$-monoid if the context is clear.

A *morphism of monoids* between two monoids (M, μ_M, η_M) and $(M', \mu_{M'}, \eta_{M'})$ is a morphism $\phi\colon M \to M'$ such that

$$\phi \circ \mu_M = \mu_{M'} \circ (\phi \otimes \phi)$$

and

$$\phi \circ \eta_M = \eta_{M'}$$

The following proposition shows that the monoidal structure on $\bullet$ does indeed capture the algebraic nature of substitution. In the following, if M is a $\bullet$-monoid, we will often write $\mu_M(t, \theta)$ instead of $\mu_M\overline{(t, \theta(x_1), \ldots, \theta(x_n))}$, that is we lift μ_M to operate on tuples instead of equivalence classes of these.

Property 54 The functor $\mathcal{T}rm$ is a monoidal object for $\bullet$, with the following structural morphisms:

- $\mu\colon \mathcal{T}rm \bullet \mathcal{T}rm \to \mathcal{T}rm$ is defined by taking a class with representative $(t, u_{x_1}, \ldots, u_{x_n})$ in $\mathcal{T}rm_T(\Gamma) \times \prod_{x \in \mathrm{dom}(\Gamma)} \mathcal{T}rm_{\Gamma(x)}(\Delta)$ to the term $t\{\vec{x} \mapsto \vec{u}\}$ in $\mathcal{T}rm^{\Sigma}_T(\Delta)$. We will write interchangeably $\mu_{\mathcal{T}rm}$ and subst.

- $\eta\colon \mathrm{dom} \to \mathcal{T}rm$ we just take the inclusion functor *var* defined previously.

The proof relies principally on this classic lemma [Pie02]:

Lemma 55 (Substitution lemma)

We write $\Delta \vdash \theta\colon \Gamma$ if θ is a substitution and Δ, Γ are contexts such that $\Delta \vdash \theta(x)\colon \Gamma(x)$ for all $x \in \mathsf{dom}(\Gamma)$.

Let $\Gamma \vdash t\colon T$ and $\Delta \vdash \theta\colon \Gamma$. Then:

$$\Delta \vdash t\theta\colon T$$

Proof. We proceed by induction on $\Gamma \vdash t\colon T$. The only non-trivial case is **abs**: in this case we have $\Gamma, x\colon U \vdash u\colon V$ with $t = \lambda x.u$. Observe that the substitution $\theta' = \theta^x_x$ satisfies $\Delta, x\colon U \vdash \theta'\colon \Gamma, x\colon U$, so by induction hypothesis, $\Delta, x\colon U \vdash t\theta'\colon V$. We may then apply **abs** to obtain $\Delta \vdash \lambda x\colon U.(t\theta')\colon U \to V$ which, given that x is not in the domain of θ, implies

$$\Delta \vdash t\theta\colon T$$

as required. ■

To prove property 54 we need to verify 3 things (for details see [Zsi10]):

Proof.

1. The subst morphism is well defined: we verify that if $(t, \theta) \sim (t', \theta')$ then $t\theta = t'\theta'$. In this case if $t \in \mathcal{T}rm_T(\Gamma)$ and $t' \in \mathcal{T}rm_T(\Gamma')$ we have $t' = t\{\vec{x} \mapsto \vec{y}\}$, where $\vec{x}$ are the variables of $\mathsf{dom}(\Gamma)$ and $\vec{y}$ are those of $\mathsf{dom}(\Gamma')$, and $\theta(x_i) = \theta'(y_i)$ for each x_i. From this we can conclude that $t\theta = t'\theta'$.

2. The term $t\theta$ is indeed in $\mathcal{T}rm_T(\Delta)$ if (t, θ) is a representative of an element of $(\mathcal{T}rm \bullet \mathcal{T}rm)_T(\Delta)$. This follows from the lemma.

3. We need to verify the equations for monoid objects: this is a consequence of the equalities

$$(t\theta)\theta' = t(\theta' \circ \theta)$$

where $\theta' \circ \theta(x) = \theta(x)\theta'$ for all $x \in \mathsf{dom}(\theta)$.

$$x_i\{\vec{x} \mapsto \vec{t}\} = t_i \text{ and } t = t\{\vec{x} \mapsto \vec{x}\}$$

which are easily proven by induction on the term t.

■

2.3 Premodels

We can now use the monoidal structure on presheafs in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ to describe algebras that allow us to perform instantiation of the variables present in a Σ_λ-algebras A by the inclusion $\eta\colon \mathsf{dom} \to A$. We define the notion of premodel, which is just a Σ_λ-algebra which has the additional monoidal structure $\bullet$, into which we can interpret the presheaf of terms, where the substitution operation and the algebraic structure are compatible. To express compatibility of the $\bullet$-structure we need to define the *strength* morphism st, which intuitively expresses the commutation of the substitution with constructors of the algebra.

Definition 56 Let Σ be a signature and $\mathcal{R}$ be a well-typed left-algebraic rewrite system. A Σ_λ*-monoid* M is a $V + \Sigma_\lambda$-algebra, that is also a $\bullet$-monoid, and such that the following diagram commutes:

$$\begin{tikzcd}
\Sigma_\lambda(M) \bullet M \arrow[r, "st_M"] \arrow[d, "ev_M \bullet id_M"'] & \Sigma_\lambda(M \bullet M) \arrow[r, "\Sigma_l(\mu_M)"] & \Sigma_\lambda(M) \arrow[d, "ev_M"] \\
M \bullet M \arrow[rr, "\mu_M"] & & M
\end{tikzcd}$$

where $st_M : \Sigma_\lambda(M) \bullet M \to \Sigma_\lambda(M \bullet M)$ is the *strength* morphism defined piecewise on each component of $\Sigma_\lambda(M)$:

- signature case: we send $\overline{(f, \theta)}$ to f
- application case: we send $\overline{((a, b), \theta)}$ to $(\overline{(a, \theta)}, \overline{(b, \theta)})$
- abstraction case: we send $\overline{(t, \theta)}$ to $\overline{(t, \theta'^{x}_{x})}$ where θ' is the weakening of θ (from $M_\Gamma(\Delta)$ to $M_\Gamma(\Delta, x : T)$) if $t \in M_U(\Gamma, x : T)$, by observing that if $t \in M_T(\Gamma, x : A) = M_T^{x:A}(\Gamma)$ and $\theta \in M_\Gamma(\Delta)$ then $\overline{(t, \theta'^{x}_{x})}$ is in $M \bullet M_T(\Delta, x : A) = (M \bullet M)_T^{x:A}(\Delta)$.

This last condition expresses compatibility of substitution with the algebraic structure, that is that substitution is a congruence.

A *morphism* of Σ_λ-monoids from M to N is a morphism $\phi : M \to N$ that is simultaneously a $V + \Sigma_\lambda$-algebra morphism and a $\bullet$-monoid morphism.

For the proof that st_M is indeed a morphism (and that the definition given above is stable by $\sim$ equivalence classes), we once again refer to Zsido [Zsi10]. Notice that we only consider the Σ_λ-algebra structure in the definition of a Σ_λ-monoid, though we require that it is a $V + \Sigma_\lambda$-algebra.

Property 57 $\mathcal{T}rm^{\Sigma}_{\mathcal{R}}$ is a Σ_λ-monoid.

Proof. The fact that $\mathcal{T}rm$ does indeed satisfy the diagram above can be reduced to the simple identities:

$$f\theta = f \quad (t\ u)\theta = t\theta\ u\theta \quad \lambda x.t\theta = \lambda x.(t\theta)$$

Which are a direct consequence of the definition of substitution. ■

The definition of substitution was explicitly given to respect the term structure. The coherence diagram is just a reformulation of that fact, as mentioned above.

Definition 58 (Premodel)
A Σ_λ-monoid M in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ is a *premodel* for $\mathcal{R} \cup \beta$ if there is a morphism

$$⦇\_⦈^M : \mathcal{T}rm \to M$$

of Σ_λ-monoids.

In that case the morphism $⦇\_⦈$ is called the *interpretation morphism*, or just the interpretation.

We write $⦇\Gamma \vdash t : T⦈^M$ for application of the morphism to $t \in \mathcal{T}rm_T(\Gamma)$, and drop the superscript if there is no ambiguity.

If the order structure on M is trivial, then we say that M is a *model.*

It is not coincidental that we took a similar notation for the interpretation morphism as for the semantic function of example 2. Indeed the notion of premodel and interpretation morphism can capture this kind of analysis.

Note that the fact we are working with the presheafs in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ automatically forces any model to satisfy:

$$t \to^{*}_{\mathcal{R}\cup\beta} t' \wedge \Gamma \vdash t\colon T \Rightarrow \llparenthesis \Gamma \vdash t\colon T \rrparenthesis \geq \llparenthesis \Gamma \vdash t'\colon T \rrparenthesis$$

As the $\llparenthesis\_\rrparenthesis$ morphism must preserve the preorder on each component.

Notice also that $\llparenthesis\_\rrparenthesis$ must be equal (as a function) to the initial morphism when taken to be a morphism of $V + \Sigma_\lambda$-algebras in $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, as there is only one such possible morphisms.

We introduce the concept of M-valuations and a lemma, which allow us to go between valuations and substitutions, which will be helpful for proving that the semantics defined in chapter 4 are sound.

Definition 59 Given a premodel M and contexts Δ and Γ, a *M-valuation ϕ* from Γ to Δ is a tuple $\phi \in \Pi_{x\in \mathsf{dom}(\Gamma)} M_{\Gamma(x)}(\Delta)$. In this case we write $\phi \in M_\Gamma(\Delta)$ and treat ϕ as a function from $\mathsf{dom}(\Gamma)$ to $\bigcup_T M_T(\Gamma)$. Given a type T, we define

$$\llparenthesis \Gamma \vdash t\colon T \rrparenthesis_\phi := \mu_M(\llparenthesis \Gamma \vdash t\colon T \rrparenthesis, \phi(x_1), \ldots, \phi(x_n))$$

if $\Gamma = x_1\colon U_1, \ldots, x_n\colon U_n$. In addition, given a substitution θ such that $\Delta \vdash \theta\colon \Gamma$, we write $\llparenthesis \Delta \vdash \theta\colon \Gamma \rrparenthesis$ for the M-valuation from Γ to Δ that sends a variable x to $\llparenthesis \Delta \vdash \theta(x)\colon \Gamma(x) \rrparenthesis$.

We will sometimes refer to a Γ-*valuation* instead of a M-valuation from Γ to Δ, if M is clear in the context. Notice also that if $\overline{(t, \vec{v})} \in M \bullet M(\Gamma)$ for some context $\Gamma = x_1\colon T_1, \ldots, x_n\colon T_n$, then $\vec{v} \in \Pi_{x_1 \ldots x_n} M_{\Gamma(x_i)}(\Delta)$, and we may identify $\vec{v}$ with the Γ-valuation that sends x_i to v_i.

Property 60 (Substitution lemma for premodels)

We have, for all $t \in \mathcal{T}rm_T(\Gamma)$ and substitution θ such that $\Delta \vdash \theta\colon \Gamma$:

$$\llparenthesis \Gamma \vdash t\colon T \rrparenthesis_{\llparenthesis \Delta\vdash\theta\colon\Gamma \rrparenthesis} = \llparenthesis \Delta \vdash t\theta\colon T \rrparenthesis$$

Proof. The proof is essentially a reformulation of the fact that $\llparenthesis\_\rrparenthesis$ is a morphism of $\bullet$-monoids. Indeed, we have for each term t and substitution θ with $\Gamma \vdash t\colon T$ and $\Delta \vdash \theta\colon \Gamma$,

$$\llparenthesis\_\rrparenthesis \circ \mu_{\mathcal{T}rm}(t, \theta) = \llparenthesis \Delta \vdash t\theta\colon T \rrparenthesis$$

and

$$\mu_M \circ (\llparenthesis\_\rrparenthesis \bullet \llparenthesis\_\rrparenthesis)(t, \theta) = \llparenthesis \Gamma \vdash t\colon T \rrparenthesis_{\Delta\vdash\theta\colon\Gamma}$$

But the condition of $\llparenthesis\_\rrparenthesis$ being a monoid morphism implies that the diagram

$$\begin{tikzcd}
\mathcal{T}rm \bullet \mathcal{T}rm \arrow[r, "\llparenthesis\_\rrparenthesis\bullet\llparenthesis\_\rrparenthesis"] \arrow[d, "\mu_{\mathcal{T}rm}"'] & M \bullet M \arrow[d, "\mu_M"] \\
\mathcal{T}rm \arrow[r, "\llparenthesis\_\rrparenthesis"'] & M
\end{tikzcd}$$

Commutes, which is exactly the desired equality.

■

As a corollary, we have that substitution commutes with valuation, a fact that we shall use in the next section when dealing with the interpretation of β-reduction.

Definition 61 Let M be a premodel and ϕ be a valuation in $M_\Gamma(\Delta)$. If U is a type, u is a term such that $\Delta \vdash u\colon U$, and x is a variable that does not appear in the domain of Γ then ϕ^x_u is the valuation in $M_{\Gamma,x\colon U}(\Delta)$ that sends all variables $y \in \mathsf{dom}(\Gamma)$ to $\phi(y)$ and that sends x to u.

Corollary 62 Let M be a premodel, $t \in \mathcal{T}rm_T(\Gamma, x\colon A)$ and $u\colon \mathcal{T}rm_A(\Gamma)$. Let $var_\Gamma \in M_\Gamma(\Gamma)$ be the valuation that sends each variable $x\colon B$ in Γ to $var_M(B)(\Gamma)(x)$ (remember that $var_M\colon \mathsf{dom} \to M$). We have

$$(\!|\Gamma \vdash t\{x \mapsto u\}\colon T|\!) = (\!|\Gamma, x\colon A \vdash t\colon T|\!)_{(var_\Gamma)^x_{(\!|\Gamma \vdash u\colon A|\!)}}$$

Proof. First notice that $(\!|\Gamma \vdash x\colon B|\!)$ is equal to $var_M(B)(\Gamma)(x)$. Indeed $(\!|\_|\!)$ is necessarily the initial morphism from the Σ_λ-algebra $\mathcal{T}rm$ to the Σ_λ-algebra M. By the definition of an algebra morphism, $(\!|\_|\!) \circ var_{\mathcal{T}rm} = var_M \circ id_{\mathrm{dom}}$, which gives the desired equality.

Now the corollary follows directly from proposition 60 applied to the substitution θ which sends $y \in \mathsf{dom}(\Gamma)$ to y and x to u. ■

We may also deduce that the result still holds if we introduce an additional valuation.

Corollary 63 Let $t \in \mathcal{T}rm_T(\Gamma)$, $\theta \in \mathcal{T}rm_\Gamma(\Delta)$, and $\phi \in M_\Delta(\Theta)$. We define $\phi \circ \theta$ as the valuation in $M_\Gamma(\Theta)$ that sends $x \in \mathsf{dom}(\Gamma)$ to $(\!|\Delta \vdash \theta(x)|\!)_\phi$. Then

$$(\!|\Gamma \vdash t\colon T|\!)_{\phi\circ\theta} = (\!|\Delta \vdash t\theta\colon T|\!)_\phi$$

Proof. Applying associativity of μ (the first law of monoid objects, see definition 53), we have

$$(\!|\Gamma \vdash t\colon T|\!)_{\phi\circ\theta} = \mu_M((\!|\Gamma \vdash t\colon T|\!)_{(\!|\Delta\vdash\theta\colon\Gamma|\!)}, \phi)$$

which applying proposition 60 gives:

$$\mu_M((\!|\Delta \vdash t\theta\colon T|\!), \phi) = (\!|\Delta \vdash t\theta\colon T|\!)_\phi$$

■

Proving that the interpretation is indeed a morphism of *ordered* Σ_λ-algebras can be quite tedious, but by using monotonicity of μ on the $\mathcal{T}rm$ presheaf it can be made easier: indeed if $(\!|\_|\!)$ is a Σ_l-monoid morphism in $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, then all that is required to prove that it is a morphism in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ is to show that it preserves the order on each component $\mathcal{T}rm_T(\Gamma)$, that is that $(\!|\Gamma \vdash t\colon T|\!) \geq (\!|\Gamma \vdash u\colon T|\!)$ if $t \to^*_{\mathcal{R}\cup\beta} u$. We show that it is only necessary to do this for $t = l$ and $u = r$ with $l \to r \in \mathcal{R}$ and for $t \to^{hd}_\beta u$, as substitution respects the structure of $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$.

Property 64 (Simplified condition for premodels)

Suppose that M is a Σ_λ-monoid. If $(\!|\_|\!)$ is in fact a morphism of Σ_λ-monoids *in* $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, and for all rules $\Gamma \vdash l \to r\colon T$ in $\mathcal{R}$

$$(\!|\Gamma \vdash l|\!) \geq (\!|\Gamma \vdash r|\!)$$

And that in addition

$$(\!|\Gamma \vdash (\lambda x\colon A.t)u\colon T|\!) \geq (\!|\Gamma \vdash t\{x \mapsto u\}\colon T|\!)$$

for every $t \in \mathcal{T}rm_T(\Gamma)$ and $u \in \mathcal{T}rm_A(\Gamma)$. Then $(\!|\_|\!)$ is in fact a Σ_λ-monoid morphism for $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ and thus M is a premodel.

Lemma 65 Suppose that $\Delta \vdash t : T$, $\Gamma \vdash l : T$ and there is a substitution such that $t = l\theta$. Then $\theta \in \mathcal{T}rm_\Gamma(\Delta)$.

Proof. Easy induction on the derivation of $\Gamma \vdash l : T$.

We prove property 64:

Proof. The morphism $(\!|\_|\!)$ is the initial morphism of Σ_λ-algebras in $\mathbf{Set}^{\mathcal{G}}_{\mathcal{T}}$, and it is a morphism of $\bullet$-monoids. We only need to show that for each Δ, T and each $t, u \in \mathcal{T}rm_T(\Delta)$, if $t \rightarrow^*_{\mathcal{R}\beta} u$, then $(\!|\Delta \vdash t : T|\!) \geq (\!|\Delta \vdash u : T|\!)$. We will proceed by induction on the number of rewrite steps and on the position on which rewriting takes place, and use the rather strong requirements that are imposed on Σ_λ-monoids and the initial morphism $(\!|\_|\!)$.

- head rewrite: we distinguish two cases.
  - There is some rule $l \rightarrow r \in \mathcal{R}$ and some substitution θ such that $t = l\theta$. The rule $l \rightarrow r$ is typed by some Γ, and by the above lemma, we have $\theta \in \mathcal{T}rm_\Gamma(\Delta)$. We have by hypothesis $(\!|\Gamma \vdash l : T|\!) \geq (\!|\Gamma \vdash r : T|\!)$. As μ_M is a morphism in the category of presheafs in $\mathbf{Pre}^{\mathcal{G}}_{\mathcal{T}}$ (and therefore preserves the order on the first component of $M \bullet M$), we have

    $$(\!|\Gamma \vdash l : T|\!)_{(\!|\Delta \vdash \theta : \Gamma|\!)} \geq (\!|r|\!)_{(\!|\Delta \vdash \theta : \Gamma|\!)}$$

    Which by lemma 60 implies

    $$(\!|\Delta \vdash l\theta : T|\!) \geq (\!|\Delta \vdash r\theta : T|\!)$$

  - We have $t = (\lambda x : A.t')u'$ and $u' = t'\{x \mapsto u'\}$. We may conclude by direct application of the hypothesis.

- $t = t_1\ t_2$ with $t_1 \rightarrow u_1$ and $u = u_1\ t_2$. As $(\!|\_|\!)$ is a morphism of Σ_λ algebras

  $$(\!|\Gamma \vdash t_1\ t_2 : T|\!) = app_M((\!|\Gamma \vdash t_1 : A \longrightarrow T|\!), (\!|\Gamma \vdash t_2 : A|\!))$$

  for some type A. Then by induction

  $$(\!|\Gamma \vdash t_1 : A \longrightarrow T|\!) \geq (\!|\Gamma \vdash u_1 : A \longrightarrow T|\!)$$

  And as app_M is a morphism of presheafs,

  $$app_M((\!|\Gamma \vdash t_1 : A \longrightarrow T|\!), (\!|\Gamma \vdash t_2 : A|\!)) \geq app_M((\!|\Gamma \vdash u_1 : A \longrightarrow T|\!), (\!|\Gamma \vdash t_2 : A|\!))$$

- We proceed in the same way for rewriting in the right-hand side of an application and under an abstraction.

■

In the next chapter, we will see how to use models to build a modified version of a given rewrite system. The termination of this modified rewrite system implies the termination of the original system, and as it contains semantic information from the model, termination is (hopefully) more easy to prove.

3

Semantic Labelling

In this chapter we introduce semantic labelling, and show how we can transform rewrite systems into ones for which termination is equivalent. Semantic labelling was introduced by Zantema [Zan95] for first order rewrite systems. It was noticed by Hirokawa and Middeldorp [HM06] that it was not necessary to explicitly describe the semantics of *every* function symbol. The method was turned into a procedure for determining termination of rewrite systems, by Zantema and Koprowski [KZ06] and extended to equational systems by Zantema [Zan95] and Ohsaki *et al* [OMG00]. It was carried to higher-order rewriting by Makoto Hamana [Ham07] using presheaf-algebraic semantics.

The idea is to use the presheaf semantics presented in the previous chapter to label function symbols with the semantics of their arguments. This information fundamentally alters the difficulty of showing termination of the system, as certain syntactic properties absent from the original system are now satisfied. However labelling needs to respect the operational semantics: reductions in the original system need to be simulated by reductions in the labelled one. Whereas this was a relatively simple property in first order semantic labelling, the presence of bound variables and β-reduction makes this property fail for the naïve version of higher-order semantic labelling. We solve this problem by introducing *structural rules* which allow us to resolve the discrepancies between the reductions in the unlabeled system and those in the labeled one.

Our presentation presents the following differences from that of Hamana: we are in the restricted setting of algebraic rewrite rules with β-reduction, and treat *currified* function symbols. In addition, we do not make use of *meta-variables* distinct from ordinary variables: the algebraic semantics defined in the previous section are sufficient to treat substitution and rewriting in one stratum. The main difference lies in the definition of the labeled rewrite system: Whereas Hamana defines the reduct of a labelled term as the labelling of the reduct, we give a presentation that is closer to the first order case: the labelled rules are just the original rules, labelled with all possible semantics for the free variables. This has the advantage of having a more traditional presentation of rewriting. In particular, in the Hamana version, the labeled version of β-reduction is *not* β-reduction. We wish to apply a standard termination criterion, *e.g.* the General Schema [BJO02], which has not, at present, been generalized to the labelled β-reduction. The disadvantage of our presentation is the previously mentioned loss of the correspondence between the labeled system and the original one. To correct this we add *structural rules*, which complete the labeled system and allow it to simulate the original one. Sadly, the structural rules are *not* terminating, but we show that it suffices to show *relative termination* of the rewrite rules and β-reduction with respect to these structural rules to show termination of the original system.

3.1 Labelling of Terms and Rewrite Systems

In what follows we fix a signature Σ with a type assignment τ, a left-algebraic rewrite system $\mathcal{R}$ that is well-typed, and a $\mathcal{R}$-premodel M.

Definition 66 Suppose that $\phi \in M_\Gamma(\Delta)$ is a valuation. If x is a variable that does not occur in Γ and Δ, and U is a type we write ϕ^x_x for the valuation in $M_{\Gamma,x\colon U}(\Delta, x\colon U)$ that sends $y \in \mathsf{dom}(\Gamma)$ to $M(\iota)(\phi(y))$ and x to $var^M_U(\Delta)(x)$, where $\iota\colon \Delta \to \Delta, x\colon U$ is the inclusion function (also called *weakening*). Notice that U is implicit in this definition.

Note that by the algebra structure of $\mathcal{T}rm$, if $\Gamma \vdash t\colon T$ is a derivation, and $\iota\colon\Gamma \to \Gamma, x\colon U$ is the inclusion morphism, then $\Gamma, x\colon U \vdash t\colon T$, and $t = \mathcal{T}rm(\iota)(t) \in \mathcal{T}rm_T(\Gamma, x\colon U)$. Then by definition of monoid morphisms

$$M(\iota)((\!|\Gamma \vdash t\colon T|\!)) = (\!|\Gamma, x\colon U \vdash \iota(t)\colon T|\!)$$

and $\iota(t) = t$, as ι is the identity on $\mathsf{dom}(\Gamma)$.

Definition 67 (Labelling)
For each $f \in \Sigma$ let L_f be a set. We define the set Σ^L of *labeled functions* by:

$$\Sigma^L := \{f_l \mid f \in \Sigma, l \in L_f\}$$

For each element of Σ we suppose given a natural number n_f, the number of *recursive arguments of* f, and we suppose that for each rule $f\ l_1 \dots l_n \to r \in \mathcal{R}$, $n = n_f$.

Let $f \in \Sigma$. We define the set of M*-labels* for f (or f-labels) to be:

$$L^M_f := \left(\sum_{T\in\mathcal{T}} \sum_{\Gamma\in\mathcal{G}} M_T(\Gamma)\right)^{n_f}$$

Let the set of *labelled terms* $\overline{\mathcal{T}rm}^\Sigma$ or just $\overline{\mathcal{T}rm}$ be the set $\mathcal{T}rm^{\Sigma^{L^M}}$. The set of labels is intuitively the set tuples of all possible semantics of terms $t \in \mathcal{T}rm_T(\Gamma)$ for some T and Γ.

We define the *labelling function* as follows: given t such that $\Gamma \vdash t\colon T$, and a valuation $\phi \in M_\Gamma(\Delta)$ we define the *labelled term* $\overline{t}^\phi \in \overline{\mathcal{T}rm}^\Sigma$ by induction on the typing derivation:

- **ax**: $\overline{x}^\phi = x$.
- **app**: we distinguish two cases:

1. $t = f\ t_1 \dots t_k$ with $\Gamma \vdash t_i\colon T_i$, and $T = T_1 \to \dots T_k \to U_1 \to \dots U_m \to T_f$ with T_f an atomic type. Again we distinguish two cases:

(a) $k \geq n_f$: in this case,

$$\overline{t}^\phi = f_l\ \overline{t_1}^\phi \dots \overline{t_k}^\phi$$

where

$$l = ((\!|\Gamma \vdash t_1|\!)_\phi, \dots, (\!|\Gamma \vdash t_{n_f}|\!)_\phi)$$

(b) $k < n_f$: in this case,

$$\overline{t}^{\phi} = f_l\ \overline{t_1}^{\phi} \dots \overline{t_k}^{\phi}$$

where

$$l = (a_1, \dots, a_k, v_1, \dots, v_{n_f - k})$$

With $a_i = (\!|\Gamma, \vec{x} \colon \vec{U} \vdash t_i \colon T_i|\!)_{\phi^{\vec{x}}_{\vec{x}}}$ and $v_j = (\!|\Gamma, \vec{x} \colon \vec{U} \vdash x_i \colon U_i|\!)_{\phi^{\vec{x}}_{\vec{x}}} = var^{M}_{U_i}(x_i)$, where the x_i are fresh variables not in Γ or Δ, which gives $\phi^{\vec{x}}_{\vec{x}} \in M_{\Gamma, \vec{x} \colon \vec{U}}(\Delta, \vec{x} \colon \vec{U})$.

2. Otherwise $t = t_1\ t_2$ and

$$\overline{t}^{\phi} = \overline{t_1}^{\phi}\ \overline{t_2}^{\phi}$$

- **abs**: we have $t = \lambda x \colon U.u$, we define

$$\overline{t}^{\phi} = \lambda x \colon U.\overline{u}^{\phi^{x}_{x}}$$

We do not include a **symb** case, as it is included in the **app** case where $k = 0$.

The set of labelled terms contains semantic information in the labels. Intuitively, we just add the semantics of the recursive arguments of f as a label on f for each $f \in \Sigma$. In the higher order setting however, it is possible for f to not be applied to n_f arguments. In that case, we just add the appropriate number of fresh variables and replace the semantics of the missing arguments with the semantics of the added variables, and weaken ϕ accordingly. This is the same as treating the under-applied symbol as if it were preceded by a succession of abstractions: in the term $f\ t_1 \dots t_k$ with $k < n_f$, we label f with l, if

$$\overline{\lambda x_1 \colon U_1 \dots x_{n_f-k} \colon U_{n_f-k}.f\ t_1 \dots t_k\ x_1 \dots x_{n_f-k}} = \lambda x_1 \colon U_1 \dots x_{n_f-k} \colon U_{n_f-k}.f_l\ \overline{t_1} \dots \overline{t_k}\ x_1 \dots x_{n_f-k}$$

Labelling allows us to perform certain syntactic analyses (for example examining the function symbols in right hand sides), with a higher chance of success.

However, higher order rewriting presents certain caveats when trying to apply the semantic labelling technique. In particular, in the first order case, the fundamental *substitution lemma* is guaranteed to hold: for all terms t and substitutions θ, $\overline{t\theta} = \overline{t}\overline{\theta}$, in short, substitution commutes with labelling. This does *not* hold in the higher order case we have just outlined, due to the lack of “compositionality”: the labelling depends on the global form of the term, not just of the labelling of its subterms. To understand the formal statement of commutativity, we need to introduce labellings of substitutions.

Definition 68 (Labelling of a substitution)
Let θ be a substitution and Δ and Γ be two contexts such that $\Delta \vdash \theta \colon \Gamma$. Let Θ be a context and ϕ be a valuation in $M_{\Delta}(\Theta)$. We define $\overline{\theta}^{\phi}$ to be the substitution that sends each $x \in \mathsf{dom}(\theta)$ to $\overline{\theta(x)}^{\phi}$.

For all t, θ, ϕ, the commutativity lemma could then be expressed as: $\overline{t\theta}^{\phi} = \overline{t}^{\phi\circ\theta}\overline{\theta}^{\phi}$. Let us give an example where this rule fails in the higher-order case:

Example 11 (Counter-example to commutativity)
Let $\Sigma = \{f, c\}$ be a signature, such that f is given the type $T \to U$ and c is given the type T for T, U base types. Fix a Σ_{λ}-algebra M. Let $\emptyset$ be the empty valuation (the unique valuation in $M_{[]}([])$, with $[]$ the empty context). We write $\overline{t}$ instead of $\overline{t}^{\emptyset}$, and we have

$$\overline{x\ c}^{\emptyset\circ\{x \mapsto f\}} = x\ c$$

Since the constant c does not have recursive arguments ($n_c = 0$) and

$$\overline{f} = f_{\llparenthesis y:T \vdash y:T \rrparenthesis}$$

with y some fresh variable. Therefore

$$\overline{x\ c}^{\emptyset \circ \{x \mapsto f\}} \overline{\{x \mapsto f\}} = f_{\llparenthesis y:T \vdash y:T \rrparenthesis}\ c$$

On the other hand, we have

$$\overline{f\ c} = f_{\llparenthesis [] \vdash c:T \rrparenthesis}\ c$$

So the term $x\ c$ *does not* verify the commutation lemma for $\theta = \{x \mapsto f\}$ and $\phi = \emptyset$. Indeed the framework can not *predict* that f is going to be applied to the constant c, so $\overline{\theta}^{\emptyset}(x)$ must introduce a free variable y to denote the semantics of the argument of f. This is due to the fact that we work in a higher-order case, in which subterms do not contain all the information on the possible arguments of a defined function. Notice the similarity with the counter-example to completeness of our size-based termination criterion (example 9).

We need to introduce some mechanism to manage the failure of this lemma. The fundamental idea is that we may "instantiate" certain variables by a given argument. In the example above, we could allow the label $\llparenthesis y:T \vdash y:T \rrparenthesis$ to be instantiated to $\llparenthesis \vdash c:T \rrparenthesis$. We then add these possible instantiations to the rewrite rules.

Definition 69 (Instantiation order)
Let Γ, Δ be contexts, A a type and a an element of $M_A(\Gamma)$. We define the *instantiation* morphism $inst_a : M_T(\Gamma, x:A, \Delta) \to M_T(\Gamma, \Delta)$ by:

$$inst_a(t) = \mu_M(t, \phi_a)$$

Where ϕ_a is the valuation in $M_{\Gamma, x:A, \Delta}(\Gamma, \Delta)$ that sends every $y \in \mathrm{dom}(\Gamma, \Delta)$ to its inclusion $var^M(\Gamma, x:A, \Delta)(y)$ and x to a.

We define the *instantiation order* $>_{inst}$ on f-labels by taking the transitive closure of the following relation: if $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_n)$ with $a_i, b_i \in M_{T_i}(\Gamma)$ for a given Γ, then

$$\vec{a} >_{inst} \vec{b} \Leftrightarrow \exists a,\ \forall i, b_i = inst_a(a_i)$$

Finally we define the algebraic rewrite system $Inst$ on $\overline{\mathcal{T}rm}$ to be the set of rules

$$\{f_l \to f_r \mid f \in \Sigma, l, r \in L_f, l >_{inst} r\}$$

Notice that the instantiation order on labels is well founded, by simple induction on the size of the context involved in the labels, and that it is *uniform* in its arguments, that is if $\vec{a} >_{inst} \vec{b}$, then there is a *unique* a such that $b_i = inst_a(a_i)$. Uniformity will be important to prove the structural property we use to prove termination (lemma 129). Notice also that in example 11, we have

$$f_{\llparenthesis y:T \vdash y:T \rrparenthesis}\ c \to_{Inst} f_{\llparenthesis \vdash c:T \rrparenthesis}\ c$$

which is reassuring, as $Inst$ was built just for this purpose!

We shall need the following lemmas:

Lemma 70 Instantiation commutes with valuation: $\Gamma = x_1 : T_1, \ldots x_n : T_n$, $\Delta = y_1 : U_1, \ldots, y_m : U_m$, and let $\Gamma, x : A, \Delta \vdash t : T$. Then for any valuation $\phi \in M_\Gamma(\Theta)$ and element $a \in M_A(\Gamma)$,

$$inst_{\mu_M(a, \phi^{\vec{y}}_{\vec{y}})} \left(\llparenthesis \Gamma, x : A, \Delta \vdash t \rrparenthesis_{(\phi^x_x)^{\vec{x}}_{\vec{x}}} \right) = \mu_M(inst_a \llparenthesis \Gamma, x : A, \Delta \vdash t \rrparenthesis, \phi^{\vec{x}}_{\vec{x}})$$

Proof. Let $t \in M_T(\Gamma, x : A, \Delta)$, $\phi \in M_\Gamma(\Theta)$ and $a \in M_A(\Gamma)$. We need to show that

$$\mu_M(\mu_M(t, (\phi^x_x)^{\vec{y}}_{\vec{y}}), \phi_{\mu_M(a, \phi^{\vec{y}}_{\vec{y}})}) = \mu_M(\mu_M(t, \phi_a), \phi^{\vec{y}}_{\vec{y}})$$

With $\phi_{\mu_M(a, \phi^{\vec{y}}_{\vec{y}})} \in M_{\Theta, x : A, \Delta}(\Theta, \Delta)$ and $\phi_a \in M_{\Gamma, x : A, \Delta}(\Gamma, \Delta)$. Take $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ and $\Delta = y_1 : U_1, \ldots, y_m : U_m$.

Remember that the following diagram commutes:

$$\begin{tikzcd}
M \bullet (M \bullet M) \arrow[r, "\alpha"] \arrow[d, "id \bullet \mu_M"'] & (M \bullet M) \bullet M \arrow[r, "\mu_M \bullet id"] & M \bullet M \arrow[d, "\mu_M"] \\
M \bullet M \arrow[rr, "\mu"] & & M
\end{tikzcd}$$

Which gives

$$\mu_M(\mu_M(t, (\phi^x_x)^{\vec{y}}_{\vec{y}}), \phi_{\mu_M(a, \phi^{\vec{y}}_{\vec{y}})}) = \mu_M(t, \omega)$$

with

$$\omega(x_i) = \mu_M((\iota(\phi)(x_i), \phi_{\mu_M(a, \phi)}) = \iota(\phi(x_i))$$

$$\omega(x) = \mu_M(var^M(x), \phi_{\mu_M(a, \phi^{\vec{y}}_{\vec{y}})}) = \mu_M(a, \phi^{\vec{y}}_{\vec{y}})$$

and

$$\omega(y_j) = var^M(y_j)$$

Furthermore

$$\mu_M(\mu_M(t, \phi_a), \phi^{\vec{y}}_{\vec{y}}) = \mu_M(t, \omega')$$

with

$$\omega'(x_i) = \mu_M(var^M(x_i), \iota(\phi)) = \iota(\phi(x_i))$$

$$\omega'(x) = \mu_M(a, \iota(\phi)) = \mu_M(a, \phi^{\vec{y}}_{\vec{y}})$$

and

$$\omega'(y_j) = var^M(y_j)$$

Therefore $\omega = \omega'$, and the desired equality holds.

■

Lemma 71 Let t be a term such that $\Gamma, \Delta \vdash t : T$ and x be a variable that does not appear in t. If $a \in M_A(\Gamma)$, then

$$inst_a \llparenthesis \Gamma, x : A, \Delta \vdash t : T \rrparenthesis = \llparenthesis \Gamma, \Delta \vdash t : T \rrparenthesis$$

Proof. Simple induction on t.

Lemma 72 Let t, u be terms such that $\Gamma \vdash t : T \to U$ and $\Gamma \vdash u : T$. Let $\phi \in M_{\Gamma}(\Delta)$ be some valuation. Then

$$\overline{t}^{\phi}\ \overline{u}^{\phi} \to^{*}_{Inst} \overline{t\ u}^{\phi}$$

Proof. The proof proceeds by cases on the judgement $\Gamma \vdash t : T \to U$.

- **var**: By definition of labelling, $\overline{x}^{\phi}\overline{u}^{\phi} = x\ \overline{u}^{\phi} = \overline{x\ u}^{\phi}$.
- **abs**: We have by definition $\overline{\lambda x : T.t'\ u}^{\phi} = \overline{\lambda x : T.t'}^{\phi}\ \overline{u}^{\phi}$ and so $\overline{t}^{\phi}\ \overline{u}^{\phi}$ rewrites to $\overline{t\ u}^{\phi}$ in zero steps.
- **app**,**symb**: $t = f\ t_1 \ldots t_k$. We distinguish two cases,

1. if $k \geq n_f$ we have $\overline{f\ \vec{t}\ u}^{\phi} = f_{\llparenthesis \vec{t} \rrparenthesis_{\phi}} \overline{\vec{t}}^{\phi}\ \overline{u}^{\phi} = \overline{f\ \vec{t}}^{\phi}\overline{u}^{\phi}$ with $\llparenthesis \vec{t} \rrparenthesis_{\phi} = (\llparenthesis \Gamma \vdash t_1 : T_1 \rrparenthesis_{\phi}, \ldots, \llparenthesis \Gamma \vdash t_{n_f} : T_{n_f} \rrparenthesis_{\phi})$.

2. otherwise, $k < n_f$ and if $n = n_f - k$, and $U = U_1 \to \ldots \to U_n \to T_f$, then taking $\Gamma' = \Gamma, x_1 : U_1, \ldots, x_n : U_n$, and $\phi' = \phi^{\vec{x}}_{\vec{x}}$ we have

$$\overline{f\ t_1 \ldots t_k}^{\phi} = f_l\ \overline{t_1}^{\phi} \ldots \overline{t_k}^{\phi}$$

with $l = (l_1, \ldots, l_k, m_1, \ldots, m_n)$, where $l_i = \llparenthesis \Gamma' \vdash t_i : T_1 \rrparenthesis_{\phi'}$, and $m_i = \llparenthesis \Gamma' \vdash x_i : U_{k+i} \rrparenthesis_{\phi'}$.
Now take $a = \llparenthesis \Gamma'' \vdash u : U_1 \rrparenthesis_{\phi''}$ and $a' = \llparenthesis \Gamma \vdash u : U_1 \rrparenthesis$, and set $\Gamma'' = \Gamma, x_2 : U_2, \ldots x_n : U_n$ and $\phi'' = \phi^{x_2 \ldots x_n}_{x_2 \ldots x_n}$. By lemma 70, we have for each $1 \leq i \leq k$,

$$inst_a(l_i) = \mu_M(inst_a \llparenthesis \Gamma' \vdash t_i : T_i \rrparenthesis, \phi'')$$

And by lemma 71, as x_1 does not appear in t_i,

$$inst_a \llparenthesis \Gamma' \vdash t_i : T_i \rrparenthesis = \llparenthesis \Gamma' \vdash t_i : T_i \rrparenthesis$$

which gives

$$inst_a(l_i) = \llparenthesis \Gamma'' \vdash t_i : T_i \rrparenthesis_{\phi''}$$

Furthermore

$$inst_a(m_1) = \llparenthesis \Gamma'' \vdash u : U_1 \rrparenthesis_{\phi''}$$

and for $2 \leq i \leq n$

$$inst_a(m_i) = \llparenthesis \Gamma'' \vdash x_i : U_i \rrparenthesis_{\phi''}$$

again by application of lemma 71.

From the above we can deduce:

$$(l_1, \ldots, l_k, m_1, \ldots, m_n) >_{inst} (l'_1, \ldots, l'_k, a', m'_2, \ldots, m'_n)$$

Where $l'_i = \llparenthesis \Gamma'' \vdash t_i : T_i \rrparenthesis_{\phi''}$, $a' = \llparenthesis \Gamma'' \vdash u : U_1 \rrparenthesis_{\phi''}$ and $m'_i = \llparenthesis \Gamma'' \vdash x_i : U_i \rrparenthesis_{\phi''}$. But

$$\overline{f\ t_1 \ldots t_k\ u}^{\phi} = f_{l'} \overline{t_1}^{\phi} \ldots \overline{t_k}^{\phi} \overline{u}^{\phi}$$

Where $l' = (l'_1, \ldots, l'_k, a', m_2, \ldots, m'_n)$, which gives, by definition of the *Inst* rewrite rules

$$\overline{t}^{\phi}\ \overline{u}^{\phi} \to^{*}_{Inst} \overline{t\ u}^{\phi}$$

■

The commutativity lemma fails for a second reason: when substituting under a binder, it is necessary to *weaken* the context which is given for the valuation.

Example 12 Let Σ be as in example 11, take t to be the term $\lambda y\colon U \to U.y\ x$ and θ be the substitution that sends x to $f\ c$. Then $\overline{\theta}(x) = f_{\llparenthesis \vdash c \rrparenthesis}\ c$ and $\overline{t} = t$, which gives $\overline{t}\overline{\theta} = \lambda y.y\ (f_{\llparenthesis \vdash c \rrparenthesis}\ c)$. But

$$\overline{t\theta} = \lambda y.y\ (f_{\llparenthesis y \vdash c \rrparenthesis}\ c)$$

and so $\overline{t}\overline{\theta} \neq \overline{t\theta}$

To solve this discrepancy we introduce the *weakening reduction.*

Definition 73 (Weakening order)
The weakening order $>_{wk}$ is defined in the following manner: if $a \in M_A(\Gamma)$ and Γ' is an extension of Γ, then

$$a >_{wk} M(\iota)(a)$$

where $\iota\colon \Gamma \to \Gamma'$ is the inclusion function.

We turn this order into a rewrite relation on function symbols in the same way as for instantiation, with

$$f_{\vec{a}} \to_{Wk} f_{\vec{b}} \Leftrightarrow \vec{a} >_{wk} \vec{b}$$

One can already note that the order is *not* well founded. In order to prove termination of the original system, we will not be able to rely on termination of the labelled one. We will have to use a weaker notion, *relative termination.*

The following lemma will be useful for proving the commutativity lemma.

Lemma 74 Suppose $\Gamma \vdash t\colon T$, and let $\phi \in M_\Gamma(\Delta)$ be a valuation. If x is some variable not in Γ or Δ, then

$$\overline{t}^{\phi} \to^*_{Wk} \overline{t}^{\phi^x_x}$$

Where in the right hand side t is seen as a term typed in the context $\Gamma, x\colon A$.

Proof. We proceed by induction on the derivation.

- case **Ax**: trivial.
- case **App,Symb**: let $t = f\ t_1 \dots t_n$, and ι the inclusion of Γ into $\Gamma, x\colon A$. We have

  $$\overline{f\ t_1 \dots t_n}^{\phi} = f_l\ \overline{t_1}^{\phi} \dots \overline{t_n}^{\phi}$$

  with $l = (l_1, \dots, l_n, m_1, \dots, m_k)$ where $l_i = \llparenthesis \Gamma' \vdash t_i \rrparenthesis_{\phi'}$ and $m_j = \llparenthesis \Gamma' \vdash x_j \rrparenthesis_{\phi'}$ for some $\Gamma' = \Gamma, y_1\colon U_1, \dots, y_k\colon U_k$ and $\phi' = \phi^{\vec{y}}_{\vec{y}}$. Now using the functoriality of μ it is easy to verify that for each i:

  $$M(\iota)(\llparenthesis \Gamma' \vdash t_i\colon T \rrparenthesis_{\phi'}) = \llparenthesis \Gamma', x\colon A \vdash t_i\colon T \rrparenthesis_{\phi'^x_x}$$

  So taking $l' = (l'_1, \dots, l'_n, m'_1, \dots, m'_k)$ with $l'_i = \llparenthesis \Gamma', x\colon A \vdash t_i\colon T_i \rrparenthesis_{\phi'^x_x}$ and $m_j = \llparenthesis \Gamma', x\colon A \vdash x_j \rrparenthesis_{\phi'^x_x}$ we have $f_l \to_{wk} f_{l'}$. By induction hypothesis, $\overline{t_i}^{\phi} \to^*_{wk} \overline{t_i}^{\phi^x_x}$ and so

  $$\overline{f\ \vec{t}}^{\phi} \to^*_{Wk} f_{l'}\ \overline{t_1}^{\phi^x_x} \dots \overline{t_n}^{\phi^x_x} = \overline{f\ \vec{t}}^{\phi^x_x}$$

- case **App**: $t = t_1\ t_2$ with t_1 not in the shape $f\ u_1 \dots u_k$, we may conclude by immediate application of the induction hypothesis.

- case **Abs**: $t = \lambda y\colon T.u$. We may take $y \neq x$ by the Barendregt convention, and so

$$\overline{\lambda y\colon T.u}^{\phi} = \lambda y\colon T.\overline{u}^{\phi_y^y}$$

By the induction hypothesis $\overline{u}^{\phi_y^y} \to_{wk}^{*} \overline{u}^{\phi_y^y {}_x^x} = \overline{u}^{\phi_x^x {}_y^y}$ and so

$$\overline{\lambda y\colon T.u}^{\phi} \to_{Wk}^{*} \overline{\lambda y\colon T.u}^{\phi_x^x}$$

■

Using instantiation and weakening, we shall be able to "save" the lemma that failed in the strict version.

Property 75 (Weak commutativity lemma)
Let t be a term, θ be a substitution, ϕ be a valuation, such that

$$\Gamma \vdash t\colon T \quad \Delta \vdash \theta\colon \Gamma \quad \phi \in M_{\Delta}(\Theta)$$

for Γ, Δ, Θ contexts and T a type. Then

$$\overline{t}^{\phi\circ\theta}\ \overline{\theta}^{\phi} \to_{Wk\cup Inst}^{*} \overline{t\theta}^{\phi}$$

Proof. We proceed by induction on the judgement $\Gamma \vdash t\colon T$.

- **var** By definition, $\overline{x}^{\phi\circ\theta} = x$, $\overline{x}^{\phi\circ\theta}\overline{\theta}^{\phi} = \overline{\theta(x)}^{\phi}$.

- **lam** In this case, $t = \lambda x\colon T.u$ and x is supposed distinct from the variables in $\mathrm{dom}(\Gamma), \mathrm{dom}(\Delta), \mathrm{dom}(\Theta)$. We have

$$\overline{\lambda x.u}^{\phi\circ\theta}\overline{\theta}^{\phi} = \lambda x.(\overline{u}^{\phi\circ\theta_x^x}\overline{\theta}^{\phi})$$

As x is not in the domain of θ, taking θ' to be the substitution equal to θ on its domain, and that sends x to itself,

$$(\phi \circ \theta)_x^x = \phi_x^x \circ \theta'$$

And applying lemma 74, we have

$$\lambda x.(\overline{u}^{\phi\circ\theta_x^x}\overline{\theta}^{\phi}) \to_{wk}^{*} \lambda x.(\overline{u}^{\phi_x^x\circ\theta'}\,\overline{\theta'}^{\phi_x^x})$$

which, by the induction hypothesis reduces under $\to_{wk\cup Inst}^{*}$ to

$$\lambda x.\overline{u\theta'}^{\phi_x^x} = \overline{(\lambda x.u)\theta}^{\phi}$$

- **app**, **symb**: We treat 2 cases:

  - $t = f\ t_1 \dots t_n$. In this case

$$\overline{t}^{\phi\circ\theta} = f_l\ \overline{t_1}^{\phi\circ\theta} \dots \overline{t_n}^{\phi\circ\theta}$$

with

$$l = ((\!|\Gamma' \vdash t_1|\!)_{\phi\circ\theta_{\vec{y}}^{\vec{y}}}, \dots, (\!|\Gamma' \vdash t_1|\!)_{\phi\circ\theta_{\vec{y}}^{\vec{y}}}, (\!|\Gamma' \vdash y_1|\!)_{\phi\circ\theta_{\vec{y}}^{\vec{y}}}, \dots, (\!|\Gamma' \vdash y_m|\!)_{\phi\circ\theta_{\vec{y}}^{\vec{y}}})$$

For some extension $\Gamma' = \Gamma, y_1 : U_1, \ldots, y_m : U_m$. But $\phi \circ \theta^{\vec{y}}_{\vec{y}} = \phi^{\vec{y}}_{\vec{y}} \circ \theta'$ with θ' equal to θ on its domain and equal to the identity on $y_1, \ldots, y_m$, and by corollary 63, for each i

$$(\!|\Gamma' \vdash t_i|\!)_{\phi^{\vec{y}}_{\vec{y}} \circ \theta'} = (\!|\Delta' \vdash t_i\theta'|\!)_{\phi^{\vec{y}}_{\vec{y}}}$$

and

$$(\!|\Gamma' \vdash y_i|\!)_{\phi \circ \theta^{\vec{y}}_{\vec{y}}} = (\!|\Delta' \vdash \theta'(y_i)|\!)_{\phi^{\vec{y}}_{\vec{y}}} = (\!|\Delta' \vdash y_i|\!)_{\phi^{\vec{y}}_{\vec{y}}}$$

Where $\Delta' = \Delta, y_1 : U_1, \ldots, y_m : U_m$, and so $l = l'$ with

$$l' = ((\!|\Delta' \vdash t_1\theta'|\!)_{\phi^{\vec{y}}_{\vec{y}}}, \ldots, (\!|\Delta' \vdash t_n\theta'|\!)_{\phi^{\vec{y}}_{\vec{y}}}, (\!|\Delta' \vdash y_1|\!)_{\phi^{\vec{y}}_{\vec{y}}}, \ldots, (\!|\Delta' \vdash y_m|\!)_{\phi^{\vec{y}}_{\vec{y}}})$$

Now

$$\overline{t}^{\phi\circ\theta}\overline{\theta}^{\phi} = f_l\ \overline{t_1}^{\phi\circ\theta}\overline{\theta}^{\phi} \ldots \overline{t_n}^{\phi\circ\theta}\overline{\theta}^{\phi}$$

which by induction hypothesis and $l = l'$ reduces under $\rightarrow^*_{Wk \cup Inst}$ to

$$f_{l'}\ \overline{t_1\theta}^{\phi} \ldots \overline{t_n\theta}^{\phi} = \overline{(f\ t_1 \ldots t_n)\theta}^{\phi}$$

– Other cases: $t = t_1\ t_2$. In that case

$$\overline{t_1\ t_2}^{\phi\circ\theta}\overline{\theta}^{\phi} = \overline{t_1}^{\phi\circ\theta}\overline{\theta}^{\phi}\ \overline{t_2}^{\phi\circ\theta}\overline{\theta}^{\phi}$$

and by induction hypothesis this $\rightarrow^*_{Wk \cup Inst}$ reduces to

$$\overline{t_1\theta}^{\phi}\ \overline{t_2\theta}^{\phi}$$

which $\rightarrow^*_{Inst}$ reduces to $\overline{(t_1\ t_2)\theta}^{\phi}$ by lemma 72.

■

Furthermore, we show that the use of $\rightarrow^*_{Wk \cup Inst}$ is not necessary when labelling the instances of patterns, provided all constructors are fully applied to their arguments.

Lemma 76 (Commutativity lemma for patterns)
Suppose that $p = f\ l_1 \ldots l_{n_f}$ is an algebraic pattern. Suppose furthermore that in each l_i, every function symbol $c \in \Sigma$ is applied to n_c arguments. Then if $\Gamma \vdash p : T$, $\Delta \vdash \theta : \Gamma$ and $\phi \in M_\Delta(\Theta)$ as in property 75, we have

$$\overline{p}^{\phi\circ\theta}\overline{\theta}^{\phi} = \overline{p\theta}^{\phi}$$

Proof. The proof proceeds exactly as in the proof of property 75, observing that the cases in which we need to use the *Inst* and *Wk* reductions are avoided (there are no abstractions). ■

Our goal is to show that an unlabeled system can be turned into a labelled one while preserving reduction, provided we allow the extra reductions described above, and an additional *decrement relation*, to allow for the fact that we are working in a premodel and not necessarily a model. In particular, we need to be able to allow reduction in the semantic labels to match reduction at the term level.

Definition 77 (Structural rules)
If $f \in \Sigma$, we define the *decrement order* $>_{decr}$ on f-labels by taking:

$$(a_1, \ldots, a_n) >_{decr} (b_1, \ldots, b_n)$$

if $a_i >_{M_T(\Gamma)} b_i$ for some $1 \leq i \leq n$ and some Γ, T and $a_j \geq_{M_T(\Gamma)} b_j$ for $j \neq i$.
We define the rewrite system $Decr$ by

$$\{f_l \to_{Decr} f_{l'} \mid f \in \Sigma, l >_{decr} l'\}$$

We take $Struct$ to be the set of rules $Inst \cup Wk \cup Decr$.

The *labelled rewrite system* is defined by taking the labelling of the rules for all possible valuations.

Definition 78 (Labelled rewrite system)
Given the rewrite system $\mathcal{R}$, let $\overline{\mathcal{R}}$ be the rewrite system on $\overline{\mathcal{T}rm}$ defined by:

$$\{\overline{l}^{\phi} \to \overline{r}^{\phi} \mid l \to r \in \mathcal{R}, \phi \in M_\Gamma(\Delta) \text{ if } \Gamma \vdash l \to r : T\}$$

3.2 The Fundamental Lemma of Semantic Labelling

The following theorem will allow us to prove the simulation of the original rewrite system by the labelled one.

Theorem 79 (Fundamental theorem)
Let $t, t' \in \mathcal{T}rm$ such that $\Gamma \vdash t, t' : T$ and $t \to_{\mathcal{R}\cup\beta} t'$. Then for all $\phi \in M_\Gamma(\Delta)$,

$$\overline{t}^{\phi} \to^{+}_{\overline{\mathcal{R}}\cup\beta\cup Struct} \overline{t'}^{\phi}$$

The proof makes use of the following lemmas:

Lemma 80 For all elements $m \in M_T(\Gamma, x{:}A)$, $\phi \in M_\Gamma(\Delta)$ and $a \in M_A(\Delta)$

$$inst_a(\mu_M(m, \phi^x_x)) = \mu_M(m, \phi^x_a)$$

Proof. The proof is similar to that of lemma 70. We have

$$inst_a(\mu_M(m, \phi^x_x)) = \mu_M(\mu_M(m, \phi^x_x), \phi_a)$$

where ϕ_a is the valuation that sends every $y \in \mathsf{dom}(\Gamma)$ to $var(y)$ and x to a. By the first monadic law for μ_M, we have

$$\mu_M(\mu_M(m, \phi^x_x), \phi_a) = \mu(m, \psi)$$

where ψ is the valuation that sends $y \in \mathsf{dom}(\Gamma)$ to $\mu_M(\phi(y), \phi_a)$ and x to $\mu_M(var(x), \phi_a)$. But $(\phi(y), \phi_a) \sim (\phi(y), var)$, and by the second and third monadic law for μ_M

$$\mu_M(\phi(y), var) = \phi(y)$$

and

$$\mu_M(var(x), \phi_a) = a$$

which gives the desired result.

■

Lemma 81 For all terms t, and variables x fresh in Γ such that $\Gamma, x\colon A \vdash t\colon T$, $\phi \in M_\Gamma(\Delta)$ and $a \in M_A(\Delta)$

$$\overline{t}^{\phi^x_x} \rightarrow^*_{Inst} \overline{t}^{\phi^x_a}$$

Proof. We proceed by induction on the derivation of $\Gamma, x\colon A \vdash t\colon T$. The only interesting case is **App, Symb**, that is $t = f\ t_1 \dots t_n$. In this case

$$\overline{f\vec{t}}^{\phi^x_x} = f_l\ \overline{t_1}^{\phi^x_x} \dots \overline{t_n}^{\phi^x_x}$$

where $l = (\llparenthesis \Gamma' \vdash t_1 \rrparenthesis_{\phi'}, \dots, \llparenthesis \Gamma' \vdash t_n \rrparenthesis_{\phi'})$ with $\Gamma' = \Gamma, x\colon A, \Delta$ and $\phi' = (\phi^x_x)^{\vec{y}}_{\vec{y}}$ if $\Delta = y_1\colon U_1, \dots, y_m\colon U_m$. We have by lemma 80

$$\llparenthesis \Gamma' \vdash t_i\colon T_i \rrparenthesis_{\phi'} >_{inst} \llparenthesis \Gamma' \vdash t_i\colon T_i \rrparenthesis_{(\phi^x_a)^{\vec{y}}_{\vec{y}}}$$

and by the induction hypothesis

$$\overline{t_i}^{\phi^x_x} \rightarrow^*_{Inst} \overline{t_i}^{\phi^x_a}$$

Which allows us to conclude.

■

Proof of Theorem 79. The proof proceeds by induction on the position on which rewriting occurs.

- Head step: Suppose that it is the application of a rule $f\ \vec{l} \rightarrow r \in \mathcal{R}$: In that case there is some substitution θ such that $t = f\ \vec{l}\theta$ and $t' = r\theta$. We have by lemma 76

  $$\overline{f\ \vec{l}\theta}^{\phi} = \overline{f\ \vec{l}}^{\phi\circ\theta}\overline{\theta}^{\phi}$$

  Which by definition of $\overline{\mathcal{R}}$, reduces in one $\overline{\mathcal{R}}$ step to

  $$\overline{r}^{\phi\circ\theta}\overline{\theta}^{\phi}$$

  and by the substitution lemma 75

  $$\overline{r}^{\phi\circ\theta}\overline{\theta}^{\phi} \rightarrow^*_{Struct} \overline{r\theta}^{\phi}$$

  Now suppose that it is the application of the β rule: $t = (\lambda x\colon T.u)v$ and $t' = u\{x \mapsto v\}$. We have

  $$\overline{(\lambda x.u)v}^{\phi} = \lambda x.\overline{u}^{\phi^x_x}\overline{v}^{\phi} \rightarrow_\beta u^{\phi^x_x}\{x \mapsto \overline{v}^{\phi}\}$$

  By lemma 81,

  $$\overline{u}^{\phi^x_x} \rightarrow_{Inst} \overline{u}^{\phi^x_{\llparenthesis \Gamma \vdash v \rrparenthesis_\phi}}$$

  and by noticing $\phi^x_{\llparenthesis \Gamma \vdash v \rrparenthesis_\phi} = \phi \circ \{x \mapsto v\}$ and finally applying the substitution lemma (lemma 75) we get

  $$u^{\phi\circ\{x\mapsto v\}}\{x \mapsto \overline{v}^{\phi}\} \rightarrow_{Struct} \overline{u\{x \mapsto v\}}^{\phi}$$

- We have $t = \lambda x\colon T.t_1$ and $t_1 \rightarrow_{\mathcal{R}\cup\beta} t'_1$ with $t' = \lambda x\colon T.t'_1$. In that case

  $$\overline{\lambda x.t_1}^{\phi} = \lambda x.\overline{t_1}^{\phi^x_x} \rightarrow^*_{\overline{\mathcal{R}}\cup\beta\cup Struct} \lambda x.\overline{t'_1}^{\phi^x_x} = \overline{t'}^{\phi}$$

- We have $t = f\ t_1 \dots t_n$ with $t_i \rightarrow_{\mathcal{R}\cup\beta} t'_i$ and $t' = f\ t_1 \dots t'_i \dots t_n$. In this case

$$\overline{t}^{\phi} = f_l\ \overline{t_1}^{\phi} \dots \overline{t_n}^{\phi}$$

  with $l = (\llparenthesis\Gamma' \vdash t_1\rrparenthesis_{\phi'}, \dots, \llparenthesis\Gamma' \vdash t_n\rrparenthesis_{\phi'}, \llparenthesis\Gamma' \vdash y_1\rrparenthesis_{\phi'}, \dots, \llparenthesis\Gamma' \vdash y_m\rrparenthesis_{\phi'})$, where $\Gamma' = \Gamma, \vec{y}\colon \vec{U}$ and $\phi' = \phi^{\vec{y}}_{\vec{y}}$ as usual. In this case, as $\llparenthesis\_\rrparenthesis$ and μ_M are morphisms of prealgebras, we have $\llparenthesis\Gamma' \vdash t_i\rrparenthesis_{\phi'} \geq \llparenthesis\Gamma' \vdash t'_i\rrparenthesis_{\phi'}$, and so applying the induction hypothesis,

$$f_l\ \overline{t_1}^{\phi} \dots \overline{t_n}^{\phi} \rightarrow_{\overline{\mathcal{R}}\cup\beta\cup Struct} f_l\ \overline{t_1}^{\phi} \dots \overline{t'_i}^{\phi} \dots \overline{t_n}^{\phi} \rightarrow_{Decr} \overline{f\ t_1 \dots t'_i \dots t_n}^{\phi}$$

- If $t = u\ v$ with $u \neq f\vec{t}$ and $u \rightarrow_{\mathcal{R}\cup\beta} u'$ then applying the induction hypothesis gives:

$$\overline{u\ v}^{\phi} = \overline{u}^{\phi}\ \overline{v}^{\phi} \rightarrow_{\overline{\mathcal{R}}\cup\beta\cup Struct} \overline{u'}^{\phi}\ \overline{v}^{\phi}$$

  and applying lemma 72

$$\overline{u'}^{\phi}\ \overline{v}^{\phi} \rightarrow^{*}_{Inst} \overline{u'\ v}^{\phi}$$

- The symmetrical case is treated in the same way.

■

Notice that the converse also is true: define the *erasure* of a term by:

- $|x| = x$
- $|f_l| = f$
- $|\lambda x\colon T.t| = \lambda x\colon T.|t|$
- $|t\ u| = |t|\ |u|$

In this case it is easy to check that if $\overline{t}^{\phi}$ reduces under $\overline{\mathcal{R}} \cup \beta \cup Struct$ to some term u in one step, then t reduces to a term v in one or zero steps, and furthermore if the applied rule is β or in $\overline{\mathcal{R}}$, then the erased term rewrites with a β or $\mathcal{R}$ step.

3.3 Normalization of the original system

Now if every well typed labelled term was strongly normalizing under $\overline{\mathcal{R}} \cup \beta \cup Struct$, then normalization of unlabeled terms would easily follow, by application of the previous lemma. Sadly this can not be the case, as the presence of the *Wk* rules do not allow this system to be terminating. However these rewrite rules do not have a non-labeled counterpart. It is therefore more interesting to us to look at sequences of reductions which involve β and $\overline{\mathcal{R}}$ reduction steps. In particular we shall see that if there are no infinite sequence of reductions *which contain an infinite number of β or $\overline{\mathcal{R}}$ steps*, then the original system is strongly normalizing. This approach is quite common in first order rewriting, and forms the basis of many termination techniques. Details on relative termination can be found in Geser [Ges90] or Zantema [Zan04b].

Definition 82 (Relative normalization)
Let R and S be two rewrite systems. We say that a term t is strongly normalizing *in R relative to S* if there are no infinite $\rightarrow_S^* \circ \rightarrow_R$ reduction sequences starting with t, that is sequences $(t_i)_{i\in\mathbb{N}}$ and $(u_i)_{i\in\mathbb{N}}$ such that $t = t_0$, and for each i,

$$t_i \rightarrow_S^* u_i \rightarrow_R t_{i+1}$$

In that case we write $t \in \mathcal{SN}_{R/S}$.

Example 13 Let $\Sigma = \{plus, S, 0\}$ be a signature with types $\tau_{plus} = N \rightarrow N \rightarrow N$, $\tau_S = N \rightarrow N$ and $\tau_0 = N$ for some atomic type N and consider the rules:

$$\mathcal{R} = \{plus\ 0\ y \rightarrow y,\ \ plus\ (S\ \ x)\ y \rightarrow S\ \ (plus\ x\ y)\}$$

and

$$\mathcal{S} = \{plus\ x\ y \rightarrow plus\ y\ x\}$$

Then every term t is normalizing in $\mathcal{R}$ relative to $\mathcal{S}$.

Lemma 83 In the conditions of theorem 79, the reduction $\overline{t}^\phi \rightarrow^+_{\overline{\mathcal{R}}\cup\beta\cup Struct} \overline{t'}^\phi$ contains at least one rewrite step in $\overline{\mathcal{R}}$ or β.

Proof. By examination of the proof of theorem 79. ■
And the main theorem of this section can now be stated:

Theorem 84 (Preservation of termination)
The system $\mathcal{R} \cup \beta$ is strongly normalizing for well-typed terms if and only if $\overline{\mathcal{R}} \cup \beta$ is strongly normalizing relative to *Struct*.

Proof. In the "if" direction: suppose by contraposition that $\Gamma \vdash t : T$ and there is an infinite sequence $t = t_1 \rightarrow_{\mathcal{R}\cup\beta} t_2 \rightarrow_{\mathcal{R}\cup\beta} \ldots$. Then taking var to be the valuation in $M_\Gamma(\Gamma)$ that sends every $x \in \mathrm{dom}(\Gamma)$ to $var(x)$, we have by theorem 79 an infinite sequence

$$\overline{t_1}^{var} \rightarrow^*_{\overline{\mathcal{R}}\cup\beta\cup Struct} \overline{t_2} \rightarrow^*_{\overline{\mathcal{R}}\cup\beta\cup Struct} \ldots$$

and lemma 83 states that there are an infinite number of $\overline{\mathcal{R}} \cup \beta$ steps in that reduction, contradicting relative normalization of $\overline{\mathcal{R}} \cup \beta$ over *Struct*.

Conversely, suppose that $u_1 \rightarrow^*_{\overline{\mathcal{R}}\cup\beta\cup Struct} u_2 \rightarrow^*_{\overline{\mathcal{R}}\cup\beta\cup Struct} \ldots$ is an infinite reduction sequence with an infinite number of occurrences of $\overline{\mathcal{R}} \cup \beta$ steps. By taking the erasure of u_i for each i, we may build an infinite $\beta \cup \mathcal{R}$ reduction sequence of unlabeled terms.

In the next chapters, we give a realizability semantics that will be used to give an algebraic description of the size-based criterion of chapter 1 using the Σ_λ-algebra framework of the previous chapter. Then we will apply the above theorem and show relative normalization of the labelled system using a simple precedence criterion, described in chapter 5.

4

The Realizability Algebra

In this chapter we show how to define a realizability algebra which will provide the appropriate semantics to apply the labelling theorem (theorem 79) and prove the correctness of the type-based criterion presented in chapter 1.

The basic idea is to give the natural interpretation of terms by interpreting abstractions as functions, and elements of base type as tuples which encode the base type, then define the size of such an element in a natural manner, and define the interpretation of defined functions by well-founded induction on the size of its arguments.

However, if we attempt to interpret the function spaces as the full set-theoretical function space, then the sets that interpret inductive types must be huge. This comes from the fact that the interpretation space consists of a very large amount of set theoretic functions (a "sea of set theoretical functions" [MW98], attributed to Girard) some of which grow too quickly and would require working with very large sets if we desire to interpret inductive datatypes (for which constructors must be injective). We show how to remedy this problem by introducing a *realizability function space* that limits the size of the function spaces by only authorizing functions that are *realized* by some term. This allows us to work with the ordinals that occur in the traditional Tait-style proof of normalization.

Non-defined constructors can then be interpreted as their set-theoretic counterpart, that is tuples involving symbols $c \in C$ and the recursive arguments. For instance the term $S\,(S\ 0)$, if S and 0 are constructors, will be interpreted by the tuple $(S, (S, 0))$. To interpret terms which never reduce to a constructor, we introduce the special symbol $\diamond$. Defined functions are interpreted by well-founded induction using the (orthogonal) rules. We show that every term realizes its interpretation, and that the size of a term is modeled by the size-annotations of the type system presented in the previous chapter.

We interpret constructors in the standard manner, as the function that takes the supremum of its recursive arguments, and adds one. However giving a semantics to defined symbols in $\mathcal{D}$ is more difficult. We need to give an interpretation that respects the rewrite rules, in order to be able to build a model by applying proposition 64. To do this we need to proceed by well-founded induction, using the size labels in the type to justify this induction. The crucial fact is that if a term t is of type B^a in context Γ, then for every valuation of term variables ϕ and size variables μ in Γ that are "compatible" with the typing, $(\!|t|\!)_\phi \leq (\!|a|\!)_\mu$. This must be proven mutually inductively with the definition of the interpretation of the functions. Care must be taken to build only functions that are in fact in the realizability interpretation.

We then show that this realizability interpretation gives rise to a $\mathcal{R} \cup \beta$-premodel (in fact a $\mathcal{R} \cup \beta$-model) in the sense of definition 58.

4.1 The Realizability Space and the Rank function

We define the set theoretical interpretation space using the realizability semantics. The realizability relation is inspired from the classical body of research pioneered by Kleene [Kle45] (see van Oosten [vO02] for a historical survey).

We now suppose that we satisfy the hypotheses of the termination theorem (theorem 41), that is that $\Sigma = \mathcal{C} \cup \mathcal{D}$, and that there is a function τ' that associates a size type to each $f \in \Sigma$. We suppose that there are two well-founded preorders, $>_{\mathcal{B}}$ a declaration order on base types and $>_{\mathcal{D}}$ a call preorder on defined functions. For each f, $|\tau'_f| = \tau_f$, and τ' satisfies the conditions described in definition 40, namely that every $f \in \mathcal{D}$ is in elimination form and that every $c \in \mathcal{C}$ is a well-formed constructor. Finally we suppose that $\mathcal{R}$ is a left-algebraic rewrite system that passes the size criterion, and respects $>_{\mathcal{D}}$.

We are going to proceed by induction on $>_{\mathcal{B}}$ to mutually define the interpretation space $\llbracket T \rrbracket$ and the realizability relation of a type T. We are going to use the positivity conditions on the constructors, and we are going to proceed by cumulativity, by defining $\llbracket T \rrbracket^{\alpha}$ for an ordinal α (by well-founded induction) if B appears strictly positively in T and $\llbracket T \rrbracket$ is defined for any T containing only atomic types strictly smaller than B.

Definition 85 (Bounded interpretation)
We proceed by induction on the well-founded preorder $>_{\mathcal{B}}$. Take $B \in \mathcal{B}$ and suppose that $\llbracket A \rrbracket$ and the relation $\Vdash_A \subseteq \mathcal{T}rm \times \llbracket A \rrbracket$ is defined for every type A such that $A <_{\mathcal{B}} B$. Take T such that all atomic types A in T verify $A <_{\mathcal{B}} B$ we define $\llbracket T \rrbracket$ and $\Vdash_T \subseteq \mathcal{T}rm \times \llbracket T \rrbracket$ by induction on T.

- $T = A$: By hypothesis $A <_{\mathcal{B}} B$ and $\llbracket A \rrbracket$ and $\Vdash_A$ are already defined.
- $T = U \to V$. We define
$$t \Vdash_T f \Leftrightarrow \forall u\ x,\ u \Vdash_U x \Rightarrow t\ u \Vdash_V f(x)$$
and
$$\llbracket U \to V \rrbracket = \llbracket U \rrbracket \mathrel{-\!\!\triangleright} \llbracket V \rrbracket = \{ f \in \llbracket V \rrbracket^{\llbracket U \rrbracket} \mid \exists t,\ t \Vdash_{U \to V} f \}$$

We choose $\diamond$ to be a special symbol not contained in Σ.

We define $\mathcal{T}^B$ to be the set of types $T \in \mathcal{T}$ in which $C \simeq_{\mathcal{B}} B$ appears in a *strictly positive position* (see definition 30). If $T \in \mathcal{T}^B$ and α is an ordinal, we mutually define the *bounded interpretation* $\llbracket T \rrbracket^{\alpha}$ and the *bounded realizability relation* $\Vdash^{\alpha}_T \subseteq \mathcal{T}rm \times \llbracket T \rrbracket^{\alpha}$ by induction on T:

- $T = C \simeq_{\mathcal{B}} B$. We proceed by well-founded induction on α:
  - $\alpha = 0$ then we define
  $$\llbracket C \rrbracket^0 = \{\diamond\}$$
  and for every term t such that t never reduces to a term of the form $c\ t_1 \dots t_n$ with $c \in \mathcal{C}$
  $$t \Vdash^0_C \diamond$$
  - $\alpha = \beta + 1$ is a successor ordinal. We define
  $$\llbracket C \rrbracket^{\beta+1} = \Big\{ (c, v_1, \dots, v_n) \mid \tau_c = T_1 \to \dots \to T_n \to C,\ \forall i\ v_i \in \llbracket T_i \rrbracket^{\beta} \Big\} \cup \llbracket C \rrbracket^{\beta}$$
  Then we define
  $$t \Vdash^{\beta+1}_C v \Leftrightarrow t \to^* c\ t_1 \dots t_n \wedge v = (c, v_1, \dots, v_n) \wedge \forall_i\ t_i \Vdash^{\beta}_{T_i} v_i\ \vee\ t \Vdash^{\beta}_C v$$

– $\alpha = \lambda$ a limit ordinal. We then define

$$[\![C]\!]^{\lambda} = \bigcup_{\beta<\lambda} [\![C]\!]^{\beta}$$

And

$$t \Vdash_C^{\lambda} v \Leftrightarrow \exists \beta < \lambda,\ t \Vdash_C^{\beta} v$$

- $T = U \to V$: We define

$$[\![U \to V]\!]^{\alpha} = [\![U]\!] \rightharpoonup [\![V]\!]^{\alpha} = \{f \in ([\![V]\!]^{\alpha})^{[\![U]\!]} \mid \exists t,\ t \Vdash_{U\to V}^{\alpha} f\}$$

where

$$t \Vdash_{U\to V}^{\alpha} f \ \Leftrightarrow\ \forall u\ x,\ u \Vdash_U x \Rightarrow t\ u \Vdash_V^{\alpha} f(x)$$

Lemma 86 The interpretation is *monotonic*: for every $T \in \mathcal{T}^B$, if $\alpha \le \beta$ are two ordinals, then $[\![T]\!]^{\alpha} \subseteq [\![T]\!]^{\beta}$ and $\Vdash_T^{\alpha}\ \subseteq\ \Vdash_T^{\beta}$.

Proof. Without loss of generality we may suppose that $\alpha\ <\ \beta$. We proceed by induction on T:

- $T = C \simeq_{\mathcal{B}} B$: by well-founded induction on β:

– $\beta = 0$ may not occur.

– $\beta = \beta' + 1$. It is clear that $[\![C]\!]^{\beta'} \subseteq [\![C]\!]^{\beta}$. We have $\alpha \le \beta'$ and so by induction hypothesis

$$[\![C]\!]^{\alpha} \subseteq [\![C]\!]^{\beta'} \subseteq [\![C]\!]^{\beta}$$

A similar reasoning shows

$$\Vdash_C^{\alpha}\ \subseteq\ \Vdash_C^{\beta}$$

– $\beta = \lambda$ a limit ordinal. Simple application of the induction hypothesis.

- $T = U \to V$: Take some $f \in [\![U \to V]\!]^{\alpha}$ such that $t \Vdash_T^{\alpha} f$. Take an arbitrary $x \in [\![U]\!]$ and u such that $u \Vdash_U x$. We have by definition $t\ u \Vdash_V^{\alpha} f(x)$ and so by induction hypothesis $t\ u \Vdash_V^{\beta} f(x)$, by which we conclude $t \Vdash_T^{\beta} f$ and $f \in [\![T]\!]^{\beta}$.

■

Now that we have the interpretation of B at level α, we need to find an ordinal for which the interpretation is stable, that is an ordinal for which $[\![B]\!]^{\lambda} = [\![B]\!]^{\lambda+1}$. We shall show that such an ordinal exists and that it is *countable*.

Lemma 87 (Existence of a realizer)
Let $T \in \mathcal{T}^B$ and $v \in [\![T]\!]^{\alpha}$ for some ordinal α. There exists a term t such that $t \Vdash_T^{\alpha} v$.

Proof. As usual, we proceed by induction on the structure of T:

- $T = C \simeq_{\mathcal{B}} B$. By induction on α:

- $\alpha = 0$. Then $[\![C]\!]^\alpha = \{\diamond\}$ and for any variable x

$$x \Vdash_C^0 \diamond$$

- $\alpha = \alpha' + 1$. Remember that

$$[\![C]\!]^{\alpha'+1} = \{(c, v_1, \ldots, v_n \mid v_i \in [\![T_i]\!]^{\alpha'}\} \cup [\![C]\!]^{\alpha'}$$

  Take $v \in [\![C]\!]^{\alpha'+1}$, and suppose $v \notin [\![C]\!]^{\alpha'}$ (otherwise we are done). We have $v = (c, v_1, \ldots, v_n)$ with $v_i \in [\![T_i]\!]^{\alpha'}$. By induction hypothesis we may take for each i, $t_i \Vdash_{T_i}^{\alpha'} v_i$ and we have

$$c\ t_1 \ldots t_n \Vdash_C^{\alpha'+1} (c, v_1, \ldots, v_n)$$

- $T = U \to V$. We can conclude by definition of $[\![U \to V]\!]$.

■

Property 88 (Limit ordinal for the interpretation)
Let Ω be the smallest uncountable ordinal. There exists an ordinal O_B which verifies for each $T \in \mathcal{T}^B$:

$$[\![T]\!]^{O_B} = [\![T]\!]^{O_B+1}$$

Proof. Note first that Ω exists by a well-known theorem of Hartogs [Har15], and can be constructed as the set of all countable well-founded orders, ordered by initiality.

Given $T \in \mathcal{T}^B$, we consider the sets Rea_T^α of realizers of elements of $[\![T]\!]^\alpha$:

$$Rea_T^\alpha = \{t \in \mathcal{T}rm \mid \exists f \in [\![T]\!]^\alpha, t \Vdash_T^\alpha f\}$$

and we show

1. If $\alpha \leq \beta$ then $Rea_T^\alpha \subseteq Red_T^\beta$.
2. If $Rea_T^\alpha = Red_T^\beta$ then $[\![T]\!]^\alpha = [\![T]\!]^\beta$.

This will allow us to conclude using countability of $\mathcal{T}rm$. The first point follows directly from the inclusions $\Vdash_T^\alpha\ \subseteq\ \Vdash_T^\beta$.

For the second point we suppose without loss of generality that $\alpha\ <\ \beta$ we proceed by induction on T:

- $T = C \simeq_{\mathcal{B}} B$. By contraposition suppose that there is some $v \in [\![C]\!]^\beta \backslash [\![C]\!]^\alpha$. By lemma 87 there is some t such that $t \Vdash_C^\beta v$. We show that for every $v' \in [\![C]\!]^\alpha$, $t\ \nVdash_C^\alpha\ v'$. We proceed by well-founded induction on β.

  - $\beta = 0$ may not occur.
  - $\beta = \beta' + 1$. By contraposition suppose that there is some $v \in [\![T]\!]^\beta \backslash [\![B]\!]^\alpha$. By lemma 87 there is some t such that $t \Vdash_C^\beta v$. If $v \in [\![B]\!]^{\beta'}$ then we can conclude by induction hypothesis. In the other case we have $v = (c, v_1, \ldots, v_n)$ for some constructor c of type $T_1 \to \ldots T_n \to C$, and $t \to^* c\ t_1 \ldots t_n$ with $t_i \Vdash^{\beta'} v_i$. We finally proceed by well-founded induction on α:
    - ∗ $\alpha = 0$. Then $v' = \diamond$ and $t\ \nVdash_C^\alpha\ v'$

  - $*$ $\alpha = \alpha' + 1$. Then by induction hypothesis we may suppose that $v' \notin [\![C]\!]^{\alpha'}$. So we have $v' = (c', v'_1, \ldots, v'_m)$. If $c \neq c'$ we are done. Otherwise $c = c'$ and $n = m$. As we have $v \notin [\![C]\!]^{\alpha}$, $n \geq 0$ (otherwise $v = v'$) and there is some inductive index i such that $v_i \notin [\![T_i]\!]^{\alpha'}$. From the induction hypothesis we therefore have

$$t_i \nVdash^{\alpha'}_{T_i} v'_i$$

and so $t \nVdash^{\alpha}_{C} v'$.

  - $*$ $\alpha = \lambda$ a limit ordinal, we have for any $v' \in [\![C]\!]^{\lambda}$, $v' \in [\![C]\!]^{\alpha'}$ with $\alpha' < \lambda < \beta$ and we may conclude by induction hypothesis.

- – Easy by induction hypothesis.

- Easy by induction hypothesis.

Now suppose that for every $\alpha < \beta < \Omega$, $[\![T]\!]^{\alpha} \subsetneq [\![T]\!]^{\beta}$. Then we have $Rea^{\alpha}_{T} \subsetneq Rea^{\beta}_{T}$ and we can build an uncountably long sequence

$$Rea^{0}_{T} \subsetneq Rea^{1}_{T} \subsetneq \ldots Rea^{\Omega}_{T} \subseteq \mathcal{T}rm$$

which contradicts countability of $\mathcal{T}rm$. Furthermore it is easy to show if $T = U \to V$ then for any ordinal λ that

$$[\![U \to V]\!]^{\lambda} = [\![U \to V]\!]^{\lambda+1} \Leftrightarrow [\![V]\!]^{\lambda} = [\![V]\!]^{\lambda+1}$$

And so the limit ordinal for $T_1 \to \ldots \to T_n \to B$ is the same as that for B.

We have not shown that $\mathcal{O}_B = \mathcal{O}_C$ if $C \simeq_{\mathcal{B}} B$. This is only true if the inductive types C and B are *truely* mutual, that is C has a constructor c with an inductive index which depends on B and vice versa. We avoid the problem by simply setting

$$\lambda_B = \inf \left\{ \alpha \in \Omega \mid [\![B]\!]^{\alpha} = [\![B]\!]^{\alpha+1} \right\}$$

and taking

$$\mathcal{O}_B = \sup_{C \simeq_{\mathcal{B}} B} \lambda_C$$

■

We can finally define the interpretation of $C \simeq_{\mathcal{B}} B$ by taking

$$[\![C]\!] = [\![C]\!]^{\mathcal{O}_B}$$

if $[\![A]\!]$ is defined for every $A <_{\mathcal{B}} B$. This allows us to define the type interpretation for any type.

Definition 89 (Type interpretation)
Let T be a type. We define the *type interpretation* $[\![T]\!]$. We proceed by induction on $>_{\mathcal{B}}$: suppose that for each atomic type B in T and every atomic type $A <_{\mathcal{B}} B$ that $[\![A]\!]$ is defined. Then we have by induction on T:

- $T = C \simeq_{\mathcal{B}} B$. We can carry out the bounded interpretation construction described in definition, and we take $[\![C]\!] = [\![C]\!]^{\mathcal{O}_B}$, as described above.

- $T = U \to V$ we take $[\![T]\!] = [\![U]\!] \rightarrowtriangle [\![V]\!]$.

We can define the interpretation of simply typed terms into these sets in the standard manner, given an appropriate interpretation of function symbols.

Definition 90 (Interpretation of Terms)
Suppose given a function I which to each $f \in \Sigma$ associates a function $I_f \in [\![\tau_f]\!]$. Suppose given $\Gamma \vdash t : T$, and a valuation ϕ that to each $x \in \mathsf{dom}(\Gamma)$ associates an element of $[\![\Gamma(x)]\!]$ (in which case we write $\phi \in [\![\Gamma]\!]$).

We can define the conditional interpretation by induction on the typing derivation:

- **var** case: $(\!|x|\!)_\phi = \phi(x)$
- **symb** case: $(\!|f|\!)_\phi = I_f$
- **app** case: $(\!|t\ u|\!)_\phi = (\!|t|\!)_\phi((\!|u|\!)_\phi)$
- **abs** case: $(\!|\lambda x : T.t|\!)_\phi = u \mapsto (\!|t|\!)_{\phi^x_u}$

This give us an interpretation for each well-typed term, but we do not know that each term is in the interpretation of their type. To prove this we need to find realizers for the interpretation of terms. This will essentially be supplied by the term itself.

Definition 91 Given a context Γ, a substitution σ of same domain as Γ, and a valuation $\theta \in [\![\Gamma]\!]$, we write $\sigma \Vdash_\Gamma \theta$, or just $\sigma \Vdash \theta$, if for each $x \in \mathsf{dom}(\Gamma)$

$$\sigma(x) \Vdash_{\Gamma(x)} \theta(x)$$

Lemma 92 (Head expansion of realizers)
Suppose that $f \in [\![U \to V]\!]$ and that t is a term such that for each $y \in [\![U]\!]$ and $u \Vdash_U y$

$$t\{x \mapsto u\} \Vdash_V f(y)$$

then

$$\lambda x : U.t \Vdash_{U \to V} f$$

Proof. Let $u \Vdash_U y$. We proceed by induction on V to show $(\lambda x : T.t)u \Vdash_V f(y)$. The only interesting case is for base type:

We need to show that $(\lambda x : T.t)u \to^* c\ t_1 \dots t_n$ with c a constructor if and only if $t\{x \mapsto u\} \to^* c\ t_1 \dots t_n$. This is a consequence of the *standardization theorem* [Plo75] (see also Terese [BKdV03], page 568, theorem 10.2.48).

We can now state and prove the relative correctness of the interpretation:

Lemma 93 (Relative correctness of the realizability semantics)
Suppose given for each f an interpretation function $I_f \in [\![\tau_f]\!]$. Furthermore suppose that for each $f \in \Sigma$, $f \Vdash_{\tau_f} I_f$. Then if $\Gamma \vdash t : T$, $\theta \in [\![\Gamma]\!]$ and σ is a substitution such that $\sigma \Vdash \theta$, then

$$t\sigma \Vdash_T (\!|t|\!)_\theta$$

Proof. We proceed by induction on the typing derivation:

- **var**. Follows from the definition of $\sigma \Vdash \theta$.
- **symb**. Directly from hypothesis.

- **app**. Let $t = u\ v$. We have by induction hypothesis $u\sigma \Vdash_{U\to V} (\!|u|\!)_\theta$ and $v\sigma \Vdash_U (\!|v|\!)_\theta$. Therefore we have $u\sigma\ v\sigma \Vdash_V (\!|u|\!)_\theta((\!|v|\!)_\theta)$ and so

$$(u\ v)\sigma \Vdash_V (\!|u\ v|\!)_\theta$$

- **abs**. Let $t = \lambda x\colon U.v$. We may suppose that x is not in the domain of σ or θ, and we have by induction hypothesis, for any $\sigma' \Vdash_{\Gamma, x\colon U} \theta'$

$$v\sigma' \Vdash_V (\!|v|\!)_{\theta'}$$

Let y be an arbitrary element of $[\![U]\!]$ and $u \Vdash_U y$. Take $\sigma' = \sigma^x_u$ and $\theta' = \theta^x_y$. We therefore have

$$v\sigma\{x \mapsto u\} \Vdash (\!|v|\!)_{\theta^x_y}$$

application of head expansion for realizers (lemma 92) gives

$$(\lambda x\colon T.v)\sigma \Vdash_{U\to V} (\!|\lambda x\colon T.v|\!)_\theta$$

■

Property 94 If I is an interpretation such that for all $f \in \Sigma$, $I_f \in [\![\tau_f]\!]$, then for each $\Gamma \vdash t\colon T$, and each valuation $\theta \in [\![\Gamma]\!]$,

$$(\!|t|\!)_\theta \in [\![T]\!]$$

Proof. Simple induction on the derivation of t. ■

4.2 The Interpretation of Symbols

The non-defined constructors, *i.e.* the elements of $\mathcal{C}\backslash\mathcal{D}$ can be directly defined in these semantics, by using the natural interpretation into the set-theoretical description of the inductive types. To define the interpretation of defined symbols however, we will need a semantic notion of size which we will show that the syntactic annotations adequately model.

Definition 95 (Interpretation of constructors)
We define the function I which associates a function $I_c \in [\![\tau_c]\!]$ to the elements $c \in \mathcal{C}\backslash\mathcal{D}$. Let c be such a symbol, and $\tau_c = T_1 \to \ldots \to T_n \to B$. Then we define

$$I_c(v_1)\ldots(v_n) = (c, v_1, \ldots, v_n)$$

We need to show that for every $c \in \mathcal{C}$, I_c is indeed in $[\![\tau_c]\!]$.

Property 96 For every $c \in \mathcal{C}$, $I_c \in [\![\tau_c]\!]$ and

$$c \Vdash_{\tau_c} I_c$$

Proof. Suppose $\tau_c = T_1 \ldots T_n \to B$. We need to show that for any $v_1 \in [\![T_1]\!], \ldots, v_n \in [\![T_n]\!]$ and any $t_1 \Vdash_{T_1} v_1, \ldots, t_n \Vdash_{T_n} v_n$, $(c, v_1, \ldots, v_n) \in [\![B]\!]$ and

$$c\ t_1 \ldots t_n \Vdash (c, v_1, \ldots, v_n)$$

By definition of $\Vdash_{T_i}$ and $[\![T_i]\!]$, for each i there is an α_i such that $v_i \in [\![T_i]\!]^{\alpha_i}$ and $t_i \Vdash_{T_i}^{\alpha_i} v_i$. Take α to be the maximum of $\alpha_1, \dots, \alpha_n$. Then we have by definition $(c, \vec{v}) \in [\![B]\!]^{\alpha+1}$ and

$$c\ t_1 \dots t_n \Vdash_B^{\alpha+1} (c, v_1, \dots, v_n)$$

■

The semantic notion of size that we ask for appears naturally in our cumulative description of inductive types.

Definition 97 (Rank)
Let T be a type and $v \in [\![T]\!]$. The *rank* of v at type T is defined inductively by:

- $T = B$: we define
$$\mathrm{rk}_B(v) = \inf\{\alpha \mid v \in [\![B]\!]^\alpha\}$$

- $T = U \to V$:
$$\mathrm{rk}_{U\to V}(f) = \sup_{x\in[\![U]\!]} \mathrm{rk}_V\, f(x)$$

Note that while $\mathrm{rk}_{T_1\to\dots T_n\to B}(f)$ is a countable ordinal, it may be the case that it is *equal* to O_B.

The rank functional is going to be the semantic counterpart to the syntactic annotations.

Lemma 98 (Rank of a value of base type)
Let $v = (c, v_1, \dots, v_n) \in [\![B]\!]$. We have

$$\mathrm{rk}_B(v) = \max(\mathrm{rk}_{T_{i_1}}(v_{i_1}), \dots, \mathrm{rk}_{T_{i_k}}(v_{i_k})) + 1$$

Proof. First let us prove that if $T \in \mathcal{T}^B$ and $f \in [\![T]\!]$ then

$$\mathrm{rk}_T(f) = \inf\{\alpha \mid f \in [\![T]\!]^\alpha\}$$

We proceed by induction on T:

- $T = C \simeq_{\mathcal{B}} B$: follows by definition of rk_C.

- $T = U \to V$: By definition
$$f \in [\![U \to V]\!]^\alpha \Leftrightarrow \forall x \in [\![U]\!], f(x) \in [\![V]\!]^\alpha$$
and
$$\mathrm{rk}_T(f) = \sup_{x\in[\![U]\!]} \mathrm{rk}_V\, f(x)$$
which by induction hypothesis is equal to
$$\sup_{x\in[\![U]\!]} \inf\{\alpha \mid f(x) \in [\![U]\!]^\alpha\}$$
which by definition means
$$\mathrm{rk}_T(f) = \inf\left\{\alpha \mid \forall x,\ \alpha \geq \inf\{\beta \mid f(x) \in [\![V]\!]^\beta\}\right\}$$
and so
$$\mathrm{rk}_T(f) = \inf\{\alpha \mid \forall x,\ f(x) \in [\![V]\!]^\alpha\}$$

Now if $v = (c, v_1 \ldots, v_n) \in [\![B]\!]$, then $v \in [\![B]\!]^{\alpha}$ for a certain minimal $\alpha = \alpha' + 1$ with $v_i \in [\![T_i]\!]^{\alpha'}$. By definition, $v_i \in [\![T_i]\!]^0$ if i is *not* an inductive index and for each inductive index i, $\alpha' \leq \mathrm{rk}_{T_i}(v_i)$ by the above statement. Therefore if $i_1, \ldots, i_k$ are the inductive indexes, we have

$$\alpha' \geq \max(\mathrm{rk}_{T_{i_1}}(v_{i_1}), \ldots, \mathrm{rk}_{T_{i_k}}(v_{i_k}))$$

and this is an equality by minimality of α.

■

It is necessary, for the definition of the semantics of defined functions and after that the proof of strong normalization, that the semantics of our functions correspond in some fashion to the syntactic annotations on the types given by the size criterion.

Definition 99 (Interpretation of size annotations)
Let $a \in \mathcal{A}$, and let $\mu : \mathcal{V} \to \Omega$ be an assignment of countable ordinals to size variables. If a that does not contain ∞ as a subterm, the *ordinal interpretation of* a, writen $(\!|a|\!)_\mu$ is the ordinal in Ω defined by:

- $(\!|\alpha|\!)_\mu = \mu(\alpha)$
- $(\!|0|\!)_\mu = 0$
- $(\!|\,\mathrm{s}(a)|\!)_\mu = (\!|a|\!)_\mu + 1$
- $(\!|\max(a, b)|\!)_\mu = \max((\!|a|\!)_\mu, (\!|b|\!)_\mu)$

Furthermore, if a does contain ∞ as a subterm, then $a \simeq \infty$ and we define $(\!|a|\!)_\mu = (\!|\infty|\!)_\mu = \Omega$.

Notice that $(\!|\_|\!)$ is well-defined on equivalence classes of $\mathcal{A}$. To see this, one can notice by simple induction on the definition of $\simeq$, that if $a \simeq b$, then for all μ, $(\!|a|\!)_\mu \simeq (\!|b|\!)_\mu$. Notice that we need to treat the ∞ case separately, as $\infty \simeq \mathrm{s}(\infty)$ but $\Omega \neq \Omega + 1$. Notice also that if θ is a Γ-size-valuation, and $x : \vec{T} \to B^{\infty}$, then it cannot be the case that $\sup_{\vec{v} \in [\![|\vec{T}|]\!]} \theta(x)(\vec{v}) = \Omega$, for in this case $\sup_{\vec{v} \in [\![|\vec{T}|]\!]} \theta(x)(\vec{v}) \leq \mathcal{O}_B$, which is a countable ordinal.

Definition 100 (Size valuations)
Let Γ be a sized-type context and $\theta \in [\![|\Gamma|]\!]$ a valuation. We say that θ is a Γ*-size-valuation* if there is $\hat{\theta} : \mathcal{V} \to \Omega$ such that for every variable $x \in \mathrm{dom}(\Gamma)$, if x is of type $\Gamma(x) = T_1 \to \ldots \to T_n \to B^{a_x}$ then

$$\mathrm{rk}_{|\Gamma(x)|}\, \theta(x) \leq (\!|a_x|\!)_{\hat{\theta}}$$

A Γ-size-valuation is *minimal* if the inequality is in fact an equality for all $x \in \mathrm{dom}(\Gamma)$. We suppose in addition that $\hat{\theta}(\alpha) = 0$ if α does not appear in Γ.

It may seem curious that, in the definition of Γ-size-valuations, if x is of type $\vec{T} \to B^a$, we do not take into account the size annotations contained within the T_i. As a consequence of this fact is that we do not take into account the "second order" size information of the context. However this size information may never actually be useful, as the shape of the size annotations on function symbols, described in definitions 36 and 31 shows that this type information may never be used, as there is no "third order" size information in the types of elements $f \in \Sigma$: there is no size annotation different from ∞ under two nested arrows, *i.e.* a type of the shape $(B^a \to A) \to A'$ with $a \not\simeq \infty$. While this is a limitation on the expressive power of our criterion, in practical cases it is rare that size information of order higher than 2 is needed.

Lemma 101 (Correctness of the order on sizes)
Let $a, b \in \mathcal{A}$ such that $a \leq b$. For all ordinal assignments μ

$$(\!|a|\!)_\mu \leq (\!|b|\!)_\mu$$

Furthermore, if $a < b$ then

$$(\!|a|\!)_\mu < (\!|b|\!)_\mu$$

Proof. We prove $(\!|a|\!)_\mu \leq (\!|b|\!)_\mu$ by simple induction on the derivation of $a \leq b$. If $a < b$ then $\mathrm{s}(a) \leq b$ and $a, b \neq \infty$. We therefore have $(\!|\,\mathrm{s}(a)|\!)_\mu = (\!|a|\!)_\mu + 1 \leq (\!|b|\!)_\mu$ which gives

$$(\!|a|\!)_\mu < (\!|b|\!)_\mu$$

■

We need the following lemmas:

Lemma 102 (Existence of minimal-size-valuations)
Suppose that Γ is a context and that for all $x \in \mathsf{dom}(\Gamma)$, $\Gamma(x)$ is minimal for Γ, as described in definition 39. Then if θ is a valuation in $[\![|\Gamma|]\!]$, then we may find $\hat{\theta}$ which makes θ a minimal Γ-size-valuation.

Proof. If $\Gamma(x) = \vec{T} \to B^a$, then a is equal to some variable α that does not appear at another place in Γ. We can then take $\hat{\theta}(\alpha) = \mathrm{rk}_{|\Gamma(x)|}\,\theta(x)$, and $\hat{\theta}(\beta) = 0$ for all size variables not in Γ. It is easy to check that $\hat{\theta}$ satisfies the necessary conditions. ■

In fact minimal typing gives strong constraints on the shape of the size annotations of types of inductive terms.

Lemma 103 Suppose that $c \in C$ and that $\Gamma \vdash_{min} c\ l_1 \ldots l_n : B^a$. Suppose that $i_1, \ldots, i_k$ are the inductive indexes of c. Then we have

$$a = \mathrm{s}(\max(a_1, \ldots, a_k))$$

with

$$\Gamma \vdash_{min} l_{i_j} : T^j_1 \to \ldots T^j_{n_j} \to B^{a_j}$$

Proof. By well-formedness of c we have $\tau'_c = T_1 \to \ldots \to T_n \to B^b$, with $b = \mathrm{s}(\max(\alpha_1, \ldots, \alpha_k))$, if $T_{i_j} = U^j_1 \to \ldots U^j_{n_j} \to B^{\alpha_j}$. Then by inversion, the derivation $\Gamma \vdash_{min} c\ l_1 \ldots l_n$ ends with n times the application rule. Notice that if

$$(U^j_1 \to \ldots U^j_{n_j} \to B^{\alpha_j})\phi = T^j_1 \to \ldots T^j_{n_j} \to B^{a_j}$$

Then in particular $\phi(\alpha_j) = a_j$. Then we may conclude by using the fact that α_j does not appear in T_i, if $i \neq i_j$. ■

We can now prove correctness of minimal typing. It is important to note that the size annotations involved in the typing can not be equal to ∞, as this would prohibit the existence of minimal size-valuations.

Lemma 104 (Correctness of minimal typing)
Suppose that l is a constructor term such that $\Gamma \vdash_{min} l : B^a$, and that θ is a minimal Γ-size-valuation. We have

$$\mathrm{rk}_B(\!|l|\!)_\theta = (\!|a|\!)_{\hat{\theta}}$$

Proof. First notice that $a \neq \infty$. We prove by induction on the derivation that if $\Gamma \vdash_{min} t : T_1 \to \ldots \to T_n \to B^a$ then $a \neq \infty$:

- If $l = x$ then the type of x is minimal in Γ, which implies $a = \alpha$ for some variable $\alpha \in \mathcal{V}$.
- otherwise $l = c\ l_1 \ldots l_n$, and we may apply the previous lemma to see that $a = \mathrm{s}(\max(a_1, \ldots, a_k))$. By induction hypothesis $a_i \neq \infty$ for $1 \leq i \leq k$, from which we can conclude.

Then notice that $(\!|l|\!)_\theta$ is indeed defined, as I_c is defined for every $c \in \mathcal{C}$, and all function symbols of l are in $\mathcal{C}$, as $\Gamma \vdash_{min} l : T$. Finally we prove the (slightly) stronger lemma: if $\Gamma \vdash_{min} l : T_1 \to \ldots \to T_n \to B^a$, and all constructors in l are fully applied, then

$$(\!|a|\!)_{\hat{\theta}} = \mathrm{rk}_{|\vec{T}| \to B} (\!|l|\!)_\theta$$

by induction on the structure of l.

- $l = x$: The lemma follows from the definition of $\hat{\theta}$. Indeed, if $\Gamma \vdash_{min} x : B^a$, then $a = \alpha$ for some variable α, which does not appear in Γ other than in the type annotation for x. The definition of $\hat{\theta}(\alpha)$ is then exactly
$$\mathrm{rk}_B\, \theta(x)$$
- $l = c\ l_1 \ldots l_n$: In this case if the recursive indexes are $i_1, \ldots, i_k$, then by lemma 103, we have
$$a = \mathrm{s}(\max(a_1, \ldots, a_k))$$
With $\Gamma \vdash_{min} l_{i_j} : \vec{T}_j \to B^{a_j}$. By the induction hypothesis,
$$(\!|a_i|\!)_{\hat{\theta}} = \mathrm{rk}_{|\vec{T}_j| \to B} (\!|l_{i_j}|\!)$$
But on the other hand by definition of $(\!|\_|\!)_\theta$ and lemma 98
$$rk_B (\!|c\ l_1 \ldots l_n|\!)_\theta = \max(o_{i_1}, \ldots, o_{i_k}) + 1$$
with $o_i = \mathrm{rk}_{|\vec{T}_i| \to B} (\!|l_{i_j}|\!)$, and so
$$rk_B((\!|l|\!)_\theta) = \max(o_{i_1}, \ldots, o_{i_k}) + 1 = \max((\!|a_{i_1}|\!)_{\hat{\theta}}, \ldots, (\!|a_{i_k}|\!)_{\hat{\theta}}) + 1 = (\!|a|\!)_{\hat{\theta}}$$
which is the desired result.

■

We continue to build a correspondence between the semantic world and the labels on sizes. We show that size-valuations are compatible with subtyping.

Lemma 105 Suppose θ is a Γ-size-valuation, and Γ' is such that $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$, and for each x in that domain, $\Gamma(x) \leq \Gamma'(x)$. Then θ is a Γ'-size-valuation.

Proof. Note first that for any types T, U, if $T \leq U$ then $|T| = |U|$. We proceed by induction on the size of Γ. The empty case is trivial.

Suppose $\Gamma = \Delta, x : U$ and $\Gamma' = \Delta', x : T$ with $U \leq T$.

By the induction hypothesis θ is a size valuation for Δ'. Suppose $T = V_1 \to \dots V_n \to B^a$ and $U = W_1 \to \dots W_n \to B^b$. We have $|V_i| = |W_i|$ for $1 \leq i \leq n$, therefore

$$\mathrm{rk}_{|\vec{V}| \to B}\, \theta(x) = \mathrm{rk}_{|\vec{W}| \to B}\, \theta(x)$$

then by hypothesis we have

$$\mathrm{rk}_{|\vec{V}| \to B}\, \theta(x) \leq \llparenthesis b \rrparenthesis_{\hat{\theta}}$$

and by lemma 101

$$\llparenthesis b \rrparenthesis_{\hat{\theta}} \leq \llparenthesis a \rrparenthesis_{\hat{\theta}}$$

■

We want to build I_f for defined f *i.e.* $f \in \mathcal{D}$. To do this we proceed by well-founded induction on the rank of the recursive arguments of the functions, using the decrease criterion. This is somewhat reminiscent of proofs of strong normalization *realizability function spaces*, namely Altenkirch's Λ-*sets* [Alt94]. In this case, the semantics are built using the well-foundedness argument, which are themselves used to give the semantics required to apply the labelling argument, and do not directly lead to a proof of strong normalization.

We will mutually recursively define I_f and show validity with respect to the size annotations, using a *relative correctness lemma* which allows us to separately treat function symbols and general terms, as we have done for realizability (with lemma 93).

Definition 106 We say that I_f is *valid* if each $f \in \Sigma$ is in $[\![\tau_f]\!]$ and it verifies the following property: if $\tau'_f = T_1 \to \dots \to T_n \to B^{a_f}$, then given $\Gamma = x_1 : T_1, \dots, x_n : T_n$ and θ a Γ-size-valuation, then

$$\mathrm{rk}_B \llparenthesis f\ x_1 \dots x_n \rrparenthesis_\theta \leq \llparenthesis a_f \rrparenthesis_{\hat{\theta}}$$

Lemma 107 (Correctness of the size-annotations)
Suppose that I_f is valid. Then for every context Γ and term t such that $\Gamma \vdash_{size} t : B^a$ and every Γ-size-valuation θ

$$\mathrm{rk}_B \llparenthesis t \rrparenthesis_\theta \leq \llparenthesis a \rrparenthesis_{\hat{\theta}}$$

Proof. We proceed by induction on the typing derivation, and prove that if $\Gamma \vdash_{size} t : T_1 \to \dots \to T_n \to B^a$, then if $\Gamma' = \Gamma, x_1 : T_1, \dots, x_n : T_n$ with x_i fresh in Γ and θ is a Γ'-size-valuation then

$$\mathrm{rk}_B \llparenthesis t\ x_1 \dots x_n \rrparenthesis_\theta \leq \llparenthesis a \rrparenthesis_{\hat{\theta}}$$

- case **var**: If $t = x$, then we have

  $$\llparenthesis t\ x_1 \dots x_n \rrparenthesis_t = \theta(x)(\theta(x_1)) \dots (\theta(x_n))$$

  By definition of $\hat{\theta}$,

  $$\llparenthesis a \rrparenthesis_{\hat{\theta}} \geq \mathrm{rk}(\theta(x)) = \sup_{\vec{v}} \mathrm{rk}(\theta(x)(\vec{v}))$$

  Which immediately gives

  $$\llparenthesis a \rrparenthesis_{\hat{\theta}} \geq \mathrm{rk}(\theta(x)(\theta(\vec{x})))$$

- case **symb**: If $t = f$, then $T = \tau_f \phi$ for some valuation ϕ. If $\tau_f = T_1 \to \dots \to T_n \to B^{a_f}$, then by validity if I, for all $\Gamma, x_1 : T_1, \dots, x_n : T_n$-size-valuation ψ,

  $$\mathrm{rk}\, I_f(\psi(x_1)) \dots (\psi(x_n)) \leq \llparenthesis a_f \rrparenthesis_{\hat{\psi}}$$

  We proceed by case distinction:

- $f \in \mathcal{D}$: Then a_f only depends on a number k of recursive arguments of types $T_i = B_i^{\alpha_i}$ for $1 \leq i \leq k$. We can then define the $\Gamma, x_1 : T_1, \ldots, x_n : T_n$-valuation ψ as follows: ψ sends x_i to $\theta(x_i)$ and $\hat{\psi}$ sends α_i to $\llparenthesis \phi(\alpha) \rrparenthesis_{\hat{\theta}}$ for $1 \leq i \leq k$ and is equal to $\hat{\theta}$ on other variables. It is easy to check that this is indeed a size-valuation for that context and that:

$$\llparenthesis a_f \phi \rrparenthesis_{\hat{\theta}} = \llparenthesis a_f \rrparenthesis_{\hat{\psi}}$$

  from which

$$\mathrm{rk}_B \ \llparenthesis f \ x_1 \ldots x_n \rrparenthesis_\theta \leq \llparenthesis a_f \phi \rrparenthesis_{\hat{\theta}}$$

- $f \in \mathcal{C} \backslash \mathcal{D}$. In the same way, a_c only depends on a number k of inductive arguments of type $\vec{T_i} \to B_i^{\alpha_i}$ and such that $\vec{T_i}$ does not contain any size annotation different from ∞. We proceed as above to build ψ and $\hat{\psi}$ which verify:

$$\llparenthesis a_c \phi \rrparenthesis_{\hat{\theta}} = \llparenthesis a_c \rrparenthesis_{\hat{\psi}}$$

- case **abs**: We have $t = \lambda x : |T_1|.t'$. By inversion, we have in this case $T = T_1 \to \ldots \to T_n \to B^a$ (and $n \geq 1$). By induction hypothesis, for every $\Gamma, x : T_1, x_2 : T_2, \ldots, x_n : T_n$-size-valuation ψ,

$$\mathrm{rk}_B \llparenthesis t' \ x_2 \ldots x_n \rrparenthesis_\psi \leq \llparenthesis a \rrparenthesis_{\hat{\psi}}$$

  We need to show that for the $\Gamma, x_1 : T_1, \ldots, x_n : T_n$ valuation θ,

$$\mathrm{rk}_B \llparenthesis (\lambda x : T_1.t') \ \vec{x} \rrparenthesis_\theta = \mathrm{rk}_B \llparenthesis t' \rrparenthesis_{\theta^x_{\theta(x_1)}} (\theta(x_2)) \ldots (\theta(x_n)) \leq \llparenthesis a \rrparenthesis_{\hat{\theta}}$$

  For this we simply take $\psi(y) = \theta(y)$ variables other than x and $\psi(x) = \theta(x_1)$, and take $\hat{\psi}$ to be equal to $\hat{\theta}$. It is easy to verify that this is a size-valuation, and it yields the desired inequality.

- case **app**: Take $t = t_1 \ t_2$. By induction hypothesis we have for every $\Gamma, x : T, x_1 : T_1, \ldots, x_n : T_n$-size-valuation ψ,

$$\mathrm{rk}_B \llparenthesis t_1 \ x \ x_1 \ldots x_n \rrparenthesis_\psi \leq \llparenthesis a \rrparenthesis_{\hat{\psi}}$$

  We take ψ equal to $\theta(x_i)$ for $i = 1 \ldots n$ and $\psi(x) = \llparenthesis t_2 \rrparenthesis_\theta$, and take $\hat{\psi}$ to be equal to $\hat{\theta}$. We verify that this is indeed a $\Gamma, x : T, x_1 : T_1, \ldots, x_n : T_n$-size-valuation, as by induction hypothesis if t_2 is of type $U_1 \to \ldots \to U_m \to C^b$ then for every $v_1, \ldots v_m$ in $[\![|U_1|]\!], \ldots, [\![|U_m|]\!]$ we have

$$\mathrm{rk}_B \llparenthesis t_2 \rrparenthesis_\theta (v_1) \ldots (v_m) \leq \llparenthesis b \rrparenthesis_{\hat{\theta}}$$

  Then we compute

$$\llparenthesis t_1 \ t_2 \rrparenthesis_\theta (\theta(\vec{x})) = \llparenthesis t_1 \rrparenthesis_\theta (\llparenthesis t_2 \rrparenthesis_\theta)(\theta(\vec{x})) = \llparenthesis t_1 \rrparenthesis_\psi (\psi(x))(\psi(\vec{x}))$$

  and by hypothesis

$$\mathrm{rk}_B \llparenthesis t_1 \rrparenthesis_\psi (\psi(x))(\psi(\vec{x})) \leq \llparenthesis a \rrparenthesis_{\hat{\psi}}$$

  From which we can conclude

$$\mathrm{rk}_B \llparenthesis t_1 \ t_2 \rrparenthesis_\theta (\theta(\vec{x})) \leq \llparenthesis a \rrparenthesis_{\hat{\theta}}$$

- case **sub**: We have $T' \leq T$, with $\Gamma \vdash_{size} t : T'$. We proceed by induction on the derivation of $T' \leq T$.

  - Atomic case: follows from inductive hypothesis and from correctness of the order on sizes (lemma 101).

– Arrow case: follows from lemma 105 and the induction hypothesis.

■

We may now define the interpretation of I_f by well-founded induction in the semantics. Remember that if f is of type $T_1 \to \ldots \to T_n \to B$, then $>_f$ is a well-founded order on $[\![T_1]\!] \times \ldots \times [\![T_k]\!]$, with k the number of recursive arguments of f, that respects the order $\geq$ of ordinals.

Lemma 108 Suppose that $\vec{a} >_f \vec{b}$. Then for any valuation μ,

$$(\!|\vec{a}|\!)_\mu >_f (\!|\vec{b}|\!)_\mu$$

Proof. By lemma 101, $(\!|\_|\!)_\mu$ preserves $>$. We get the result directly from definition of $>_f$ (definition 37).

The interpretation I_f of the defined function f is defined in the following manner: given values $v_1, \ldots, v_k$, with k the number of recursive arguments of f, we examine all possible rules $f\,\vec{l} \to r \in \mathcal{R}$, and all possible valuations θ such that $(\!|\vec{l}|\!)_\theta = \vec{v}$. We can show that by orthogonality, at most rule and valuation may possibly exist, and in that case we define $I_f(v_1)\ldots(v_k) = (\!|r|\!)_\theta$.

Definition 109 (Value matching)
Let $\vec{l}$ be a tuple of constructor terms and $\vec{v} \in [\![\vec{T}]\!]$ for some types $T_1 \ldots T_n$.
We define the set of *value matches* $Val_{\vec{l}}(\vec{v})$ to be

$$Val_{\vec{l}}(\vec{v}) := \{\theta \mid \forall i\ (\!|l_i|\!)_\theta = v_i\}$$

If $f : B_1 \to \ldots \to B_k \to T_f$ where the B_i are the recursive arguments of f and $v_1 \in [\![B_1]\!], \ldots, v_n \in [\![B_k]\!]$, we define the set of value matches for f to be

$$Val_f(\vec{v}) = \bigcup_{\rho = f\vec{l} \to r \in \mathcal{R}} \{\rho\} \times Val_{\vec{l}}(\vec{v})$$

Orthogonality of $\mathcal{R}$ is going to guarantee that $Val_f(\vec{v})$ has at most one inhabitant.

Lemma 110 Let $f \in \mathcal{D}$ of type $B_1 \to \ldots \to B_k \to T_f$ and $v_1 \in [\![B_1]\!], \ldots, v_k \in [\![B_k]\!]$. Then $Val_f(\vec{v})$ contains at most one element.

Proof. Suppose by contradiction that there is $\vec{l}, \vec{l'}$ with rules $\rho = f\vec{l} \to r, \rho' = f\vec{l'} \to r' \in \mathcal{R}$ and θ, θ' such that for each i

$$v_i = (\!|l_i|\!)_\theta = (\!|l'_i|\!)_{\theta'}$$

We show that there is some overlap between ρ and ρ'. We proceed by induction on k, and therefore suppose that there is a substitution σ such that $l_1\sigma = l'_1\sigma, \ldots, l_j\sigma = l'_j\sigma$ for $j < k$ and show that we can extend σ such that $l_{j+1}\sigma = l'_{j+1}\sigma$. By induction on l_{j+1}:

- $l_{j+1} = x$ by orthogonality, $x \notin \mathrm{dom}(\sigma)$ and we can extend σ to be equal to l'_{j+1} on x and the identity on variables of l'_{j+1} which gives

$$l_{j+1}\sigma = l'_{j+1}\sigma$$

- $l_{j+1} = c\ m_1 \ldots m_p$. Then by induction on l'_{j+1}:

- $l'_{j+1} = x$ a variable. We may conclude as above.
- $l'_{j+1} = d\ m'_1 \dots m'_q$. We have

$$(\!| c\ m_1 \dots m_p |\!)_\theta = (\!| d\ m'_1 \dots m'_q |\!)_{\theta'}$$

by orthogonality again, no rules with head c or d may apply, and so we have

$$(\!| c\ m_1 \dots m_p |\!)_\theta = (c, (\!| m_1 |\!)_\theta, \dots, (\!| m_p |\!)_\theta$$

and

$$(\!| c\ m'_1 \dots m'_q |\!)_{\theta'} = (c, (\!| m'_1 |\!)_{\theta'}, \dots, (\!| m'_q |\!)_{\theta'}$$

which gives $c = d$, $p = q$ and by induction m_i and m'_i are unifyable, which by linearity gives l_{j+1} and l'_{j+1} unifyable.

■

Definition 111 (Interpretation of defined functions)

We define I by well-founded induction on $>_{\mathcal{D}}$.

Let $f \in \mathcal{D}$ with k recursive arguments, and suppose that I_g is defined for every $g <_{\mathcal{D}} f$. Let $v_1 \dots v_n$ be elements of $[\![T_1]\!], \dots, [\![T_n]\!]$. We define $I_f(v_1) \dots (v_n)$ by induction on $(\operatorname{rk} v_1, \dots, \operatorname{rk} v_k)$ ordered by the well-founded order $>_f$: we therefore suppose that $I_g(v'_1) \dots (v'_k)$ is defined for any $g \simeq_{\mathcal{D}} f$ and $(\vec{v}) >_f (\vec{v}')$. We distinguish two cases:

- $f \in \mathcal{D} \backslash \mathcal{C}$:
  - There is a rule $\rho = f\ \vec{l} \to r$ and a valuation such that $(\rho, \theta) \in Val_f(\vec{v})$. Then we define

$$I_f(v_1) \dots (v_n) = (\!| r |\!)_\theta (v_{k+1}) \dots (v_n)$$

  - The set $Val_f(\vec{v})$ is empty. In that case we set

$$I_f(v_1) \dots (v_n) = \diamond$$

- $d \in \mathcal{D} \cap \mathcal{C}$
  - There is a rule $\rho = d\ \vec{l} \to r$ and a valuation such that $(\rho, \theta) \in Val_d(\vec{v})$. Then we define

$$I_d(v_1) \dots (v_n) = (\!| r |\!)_\theta (v_{k+1}) \dots (v_n)$$

  as above.
  - If $Val_d(\vec{v})$ is empty then we set

$$I_d(v_1) \dots (v_n) = (d, v_1, \dots, v_n)$$

We need to show that $(\!| r |\!)_\theta$ is well defined by induction on the derivation $\Gamma \vdash_{size} r : T$: The only interesting case is

- **symb**: In this case we have $r = g\ t_1 \dots t_k$ as by hypothesis, every function symbol is fully applied to its inductive arguments. Then either $g <_{\mathcal{D}} f$ and I_g is already defined or $g \simeq_{\mathcal{D}} f$. Now by hypothesis there exists Δ such that $\Delta \vdash_{min} f\vec{l} : T_f$, and an extension Δ' of Δ such that $\Delta' \vdash_{size} g\vec{t} : T_g$ and furthermore:

- $\Delta \vdash_{min} l_i : B_i^{a_i}$
- $\Delta' \vdash_{size} t_i : B_i^{b_i}$
- $\vec{a} >_f \vec{b}$

Now by lemma 102 θ may be equipped with a Δ-minimal-size-valuation structure, and by lemma 104, for every $1 \le i \le n$, $\mathrm{rk}_{B_i}(\!|l_i|\!)_\theta = (\!|a_i|\!)_{\hat{\theta}}$. Now we need to show that for any θ' a $|\Delta'|$-valuation, that extends θ

$$(\mathrm{rk}(\!|l_1|\!)_\theta, \ldots, \mathrm{rk}(\!|l_k|\!)_\theta) >_f (\mathrm{rk}(\!|t_1|\!)_{\theta'}, \ldots, \mathrm{rk}(\!|t_k|\!)_{\theta'})$$

To show this we define the following Δ'-size-valuation θ': $\hat{\theta}'$ is equal to $\hat{\theta}$ on its domain, and sends all other size-variables in Δ' to Ω. It is routine to check that this is indeed a Δ'-size-valuation.

Now notice that by lemma 107

$$(\!|t_i|\!)_{\theta'} \le (\!|b_i|\!)_{\hat{\theta}'}$$

and we may conclude

$$\mathrm{rk}(\!|\vec{l}|\!)_\theta = (\!|\vec{a}|\!)_{\hat{\theta}} >_f (\!|\vec{b}|\!)_{\hat{\theta}'} \ge \mathrm{rk}(\!|\vec{t}|\!)_{\theta'}$$

using lemma 108 and the fact that $>_f$ respects $\ge$ (definition 37).

■

We first show that this interpretation is valid:

Property 112 (Validity of the interpretation)
The interpretation I_f is valid.

Proof. We treat both cases

- $c \in \mathcal{C}\backslash\mathcal{D}$. Suppose $\tau'_c = T_1 \to \ldots \to T_n \to B^{a_c}$, and let $\Delta = x_1 : T_1, \ldots, x_n : T_n$ and let θ be a Δ-size-valuation. Let $i_1, \ldots, i_k$ be the inductive arguments of c. We have $T_{i_j} = U_1 \to \ldots \to U_m \to B^{\alpha_i}$ with $\mathrm{rk}(\theta(x_{i_j})) \le \hat{\theta}(\alpha_i)$ which gives

$$\mathrm{rk}\ (c, \theta(x_1), \ldots, \theta(x_n)) = \max(\mathrm{rk}\ \ \theta(x_{i_1}), \ldots, \mathrm{rk}\ \ \theta(x_{i_k})) + 1 \le (\!|a_c|\!)_{\hat{\theta}}$$

- Otherwise, let $f \in \mathcal{D}$. We proceed by induction on $>_{\mathcal{D}}$. Let $\tau'_f = T_1 \to \ldots \to T_n \to B^{a_f}$, and set $\Delta = x_1 : T_1, \ldots, x_n : T_n$. Take a Δ-size-valuation θ, and set $\theta(x_1) = v_1, \ldots, \theta(x_n) = v_n$. We proceed by induction on $(\mathrm{rk}\, v_1, \ldots, \mathrm{rk}\, v_k)$ ordered by $>_f$, as in the definition of I_f.

  Now we have

$$(\!|f\ x_1 \ldots x_n|\!)_\theta = I_f(v_1) \ldots (v_n)$$

  and we examine both possible cases:

  - $Val_f(\vec{v})$ is empty. Then either $f \in \mathcal{D}\backslash\mathcal{C}$ and then $I_f(\vec{v}) = \diamond$, in which case rk $I_f(\vec{v}) = 0$ and we are done, or $f \in \mathcal{D} \cap \mathcal{C}$ and we may proceed as above.

- Otherwise a rule $\rho = f\ \vec{l} \to r$ and a valuation θ' such that $(\!|\vec{l}|\!)_{\theta'} = \vec{v}$, we have a size-context Δ' and a substitution ϕ such that $\Delta' \vdash_{min} f\ \vec{l} : \tau'_f \phi$. Now by lemma 102 we may extend θ' into a Δ'-minimal-size-valuation. The induction hypothesis and the correctness of the size annotations (lemma 107) then gives us

$$I_f(\vec{v}) = (\!|r|\!)_{\theta'} \leq (\!|a_f \phi|\!)_{\hat{\theta}'}$$

Now for each recursive arguments $T_1 = B_1^{\alpha_1}, \ldots, T_k = B_k^{\alpha_k}$, and we have by definition of minimal typing

$$\Gamma \vdash_{min} l_i : B^{\phi(\alpha_i)}$$

and by definition of a minimal-size-valuation

$$\phi(\alpha_i)\hat{\theta}' = \mathrm{rk}\ \ v_i \leq \hat{\theta}(\alpha_i)$$

This gives us

$$(\!|a_f|\!)_{\hat{\theta}' \circ \phi} \leq (\!|a_f|\!)_{\hat{\theta}}$$

By monotony of all operations in $\mathcal{A}$, and so finally

$$\mathrm{rk}\, I_f(\vec{v}) \leq (\!|a_f|\!)_{\hat{\theta}}$$

■

Next, we need to show that a symbol realizes its interpretation.

Lemma 113 Suppose that $f\ \vec{l} \to r$ is a rule in $\mathcal{R}$ and that for some $l_i = l$ that there is a valuation θ and a term t such that

$$t \Vdash_T (\!|l|\!)_\theta$$

Then there is a substitution σ such that $t \to^* l\sigma$ and

$$\sigma \Vdash_\Gamma \theta$$

Proof. By induction on the structure of l:

- $l = x$, then we take $\sigma(x) = t$ and we are done.
- $l = c\ m_1 \ldots m_p$. In this case, by orthogonality of $\mathcal{R}$, $c\ \vec{m}$ is in normal form and we have

$$(\!|l|\!)_\theta = (c, (\!|m_1|\!)_\theta, \ldots, (\!|m_p|\!)_\theta)$$

and so $t \to^* c\ t_1 \ldots t_p$ such that

$$t_i \Vdash_{T_i} (\!|m_i|\!)_\theta$$

And by induction hypothesis, there exists $\sigma_1, \ldots, \sigma_n$ such that

$$t_i \to^* m_i \sigma_i$$

and $\sigma_i \Vdash_\Gamma \theta$ on its domain. By linearity of l we can take σ to be the disjoint sum of the σ_i, and it is easy to verify that it satisfies the necessary properties.

■

Property 114 For every function symbol $f \in \Sigma$ of type τ_f

$$f \Vdash_{\tau_f} I_f$$

Proof. We proceed as above, by induction on $>_{\mathcal{D}}$. If $\tau_f = T_1 \to \ldots \to T_k \to T_f$ let $v_1 \in [\![T_1]\!], \ldots, v_k \in [\![T_k]\!]$ where k is the number of recursive arguments of f, and $t_1 \Vdash_{T_1} v_1, \ldots, t_k \Vdash_{T_k} v_k$. We only treat the case $Val_f(v_1, \ldots, v_k) \neq \emptyset$. In that case we proceed by well-founded induction on (rk $v_1, \ldots,$rk v_n) ordered by $>_f$. There is a rule $\rho = f\ \vec{l} \to r \in \mathcal{R}$, a context Γ that types that rule and a valuation $\theta \in [\![\Gamma]\!]$ such that

$$(\!|\vec{l}|\!)_\theta = \vec{v}$$

We have by lemma 113, that there exists some σ such that $\vec{t} = \vec{l}\sigma$, and $\sigma \Vdash_\Gamma \theta$. Then relative correctness (lemma 93) and the inductive hypothesis give

$$f\ t_1 \ldots t_k \to r\sigma \Vdash_{T_f} (\!|r|\!)_\theta$$

from which we conclude

$$f \Vdash I_f$$

■

Finally we may conclude that our interpretation is correct:

Corollary 115 For each $f \in \Sigma$, $f \in [\![\tau_f]\!]$.

We can then prove the non-relative version of correctness:

Property 116 (Correctness of the realizability semantics)
Suppose that $\Delta \vdash \theta : \Gamma$, $\Gamma \vdash t : T$ and that θ is a valuation into $[\![\Gamma]\!]$, and σ is a substitution that verifies $\mathsf{dom}(\theta) = \mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma)$ and $\sigma(x) \Vdash \theta(x)$ for each x in that domain. Then

$$t\sigma \Vdash (\!|t|\!)_\theta$$

Proof. This is a consequence of the lemma 114 and relative correctness of the realizability (lemma 93).

■

We have almost all we need to build our realizability model, except for the crucial fact that the interpretation is stable by reduction on terms.

Lemma 117 (Substitution lemma) Given two terms t and u, such that $\Gamma \vdash_{size} u : U$ and $\Gamma, x : U \vdash_{size} t : T$. If furthermore θ is a Γ-valuation, then

$$(\!|t\{x \mapsto u\}|\!)_\theta = (\!|t|\!)_{\theta^x_{(\!|u|\!)_\theta}}$$

Proof. simple induction on the structure of t.

■

Property 118 (Compatibility of the interpretation with reduction)
The interpretation is compatible with reduction, that is if $\Gamma \vdash_{size} t : T$ is a derivation and $t \to_{\mathcal{R}\cup\beta} t'$, then for every $\theta \in [\![\Gamma]\!]$ a valuation,

$$(\!|t|\!)_\theta = (\!|t'|\!)_\theta$$

Proof. We proceed by induction on the position at which rewriting occurs:

- The rewrite step occurs at the root. Then we proceed by case

  – $t = (\lambda x.u)v$ and $t' = u\{x \mapsto v\}$. Then application of the substitution lemma gives

$$\llparenthesis(\lambda x.u)v\rrparenthesis_\theta = \llparenthesis u\rrparenthesis_{\theta^x_{\llparenthesis v\rrparenthesis_\theta}} = \llparenthesis u\{x \mapsto v\}\rrparenthesis_\theta$$

  – $t = f\ \vec{t}$ with $\vec{t} = \vec{l}\sigma$ for some rule $f\ \vec{l} \to r \in \mathcal{R}$ and some substitution σ. Then by definition of I_f, and the substitution lemma

$$I_f(\llparenthesis t_1\rrparenthesis_\theta)\dots(\llparenthesis t_k\rrparenthesis_\theta) = I_f(\llparenthesis l_1\rrparenthesis_{\theta\circ\sigma})\dots(\llparenthesis l_k\rrparenthesis_{\theta\circ\sigma}) = \llparenthesis r\rrparenthesis_{\theta\circ\sigma} = \llparenthesis r\sigma\rrparenthesis_\theta$$

- $t\ u \to t'\ u$ with $t \to t'$ then we have by induction

$$\llparenthesis t\ u\rrparenthesis_\theta = \llparenthesis t\rrparenthesis_\theta(\llparenthesis u\rrparenthesis_\theta) = \llparenthesis t'\rrparenthesis_\theta(\llparenthesis u\rrparenthesis_\theta) = \llparenthesis t'\ u\rrparenthesis_\theta$$

- $t\ u \to t\ u'$ with $u \to u'$. Same as above.

- $\lambda x{:}T.t \to \lambda x{:}T.t'$ with $t \to t'$, then by induction hypothesis

$$\llparenthesis\lambda x{:}T.t\rrparenthesis_\theta = v \mapsto \llparenthesis t\rrparenthesis_{\theta^x_v} = v \mapsto \llparenthesis t'\rrparenthesis_{\theta^x_v} = \llparenthesis\lambda x{:}T.t'\rrparenthesis_\theta$$

■

4.3 The Realizability Model

We now build a $\mathcal{R}\cup\beta$-model structure for the realizability interpretation. We have to show first that it is a $V+\Sigma_\lambda$-algebra, then that it is a •-monoid, prove compatibility of the two structures, and finally that the structure respects the rewrite rules. The model is simply a categorical reformulation of the informal model we have described above, the requirement of being a $V+\Sigma_l$-algebra is easily fulfilled by our ability to interpret terms, and the •-monoid structure being satisfied by any structure in which interprets substitution by *function application.*

Definition 119 (Realizability model)
We define the functor $\mathrm{Rea} \in Pre^{\mathcal{G}}_{\mathcal{T}}$ by

$$\mathrm{Rea}_T(\Gamma) = [\![T_1]\!] \rightarrowtriangle \dots \rightarrowtriangle [\![T_n]\!] \rightarrowtriangle [\![T]\!]$$

if $\Gamma = x_1 : T_1, \dots, x_n : T_n$.

In this case we will sometimes write $[\![\Gamma]\!] \rightarrowtriangle [\![T]\!]$. The order on elements of $\mathrm{Rea}_T(\Gamma)$ is the trivial order: $x \geq y \Leftrightarrow x = y$.

If $\iota : \Gamma \to \Gamma'$ we define the renaming morphism $\mathrm{Rea}(\iota)\colon \mathrm{Rea}_T(\Gamma) \to \mathrm{Rea}_T(\Gamma')$ by taking

$$\mathrm{Rea}(\iota)(f)(v_1)\dots(v_m) = f(v_{\iota(1)})\dots(v_{\iota(n)})$$

with $\Gamma = x_1 : T_1, \dots, x_n : T_n$ and $\Gamma' = y_1 : U_1, \dots, y_n : U_m$, and noticing that if $t \Vdash f$ then

$$\lambda y_1 : U_1. \dots .\lambda y_m : U_m.t\ y_{\iota(1)} \dots y_{\iota(n)} \Vdash \mathrm{Rea}(\iota)(f)$$

We can often show that a function is realized by simply "mimicking" the function using the term abstraction. In fact the most elegant, if not the most concise approach to building a Rea premodel would have been to do things categorically by building a notion of "Cartesian Closed Model" and then showing that the realizability space is a cartesian closed model. Sadly this is not the case in general, as we do not have products *a priori*, though products can exist for certain choices of base types in $\mathcal{B}$, constructors in C and rules.

Property 120 Rea has the structure of a $V + \Sigma_\lambda$-algebra.

Proof. We give the four morphisms that suffice to define the evaluation *ev*:

- $var^\Gamma_T : \mathrm{dom}_T(\Gamma) \to \mathrm{Rea}_T(\Gamma)$. If $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, then if $x_i \in \mathrm{dom}_T(\Gamma)$ (then $T_i = T$) we take $var^\Gamma_T(x)$ to be equal to the function which takes elements $v_1 \in [\![T_1]\!], \ldots, v_n \in [\![T_n]\!]$ and returns v_i (in other words the i-th projection). The function is realized by the term

$$\lambda x_1 : T_1 . \ldots \lambda x_n : T_n . x_i$$

- $val^\Gamma_T : \Sigma_T \to \mathrm{Rea}_T(\Gamma)$. Given $f \in \Sigma$ of type τ_f, the function $val^\Gamma_{\tau_f}(f)$ sends $\vec{v} \in [\![\Gamma]\!]$ to the (constant) element I_f given in definitions 95 and 111, which is in $[\![\tau_f]\!]$ by proposition 114.
- $abs^T_U : \mathrm{Rea}_U(\Gamma, x : T) \to \mathrm{Rea}_{T\to U}(\Gamma)$ is defined as the identity! Indeed we have $\mathrm{Rea}_U(\Gamma, x : T) = \mathrm{Rea}_{T\to U}(\Gamma)$. There is nothing more to verify here.
- $app^U_T : \mathrm{Rea}_{U\to T}(\Gamma) \times \mathrm{Rea}_U(\Gamma) \to \mathrm{Rea}_T(\Gamma)$. We take the function that takes f and v and returns the function that takes $v_1, \ldots, v_n \in [\![\Gamma]\!]$ and returns

$$w = f\ (v_1) \ldots (v_n)(v(v_1) \ldots (v_n))$$

  If $t \Vdash f$ and $u \Vdash v$, then

$$\lambda x_1 : T_1 . \ldots . \lambda x_n : T_n . (t\ \vec{x})(u\ \vec{x}) \Vdash w$$

It is easy to verify that these applications are in fact morphisms (stable by renaming with respect to an arbitrary $\iota : \Gamma \to \Gamma'$).

Definition 121 The presheaf Rea is a $\bullet$-monoid, with the following structural morphisms:

- μ: $\mathrm{Rea} \bullet \mathrm{Rea} \to \mathrm{Rea}$ is defined on a class with representative

$$(f, v_1, \ldots, v_n) \in \mathrm{Rea}_T(\Gamma) \times \prod_{x \in \mathrm{dom}(\Gamma)} \mathrm{Rea}_{\Gamma(x)}(\Delta)$$

  as the function that takes $w_1, \ldots, w_m$ in $[\![\Delta]\!]$ and returns

$$z = f(v_1(\vec{w})) \ldots (v_n(\vec{w}))$$

  We can check that if $t \Vdash f$ and for each i, $u_i \Vdash v_i$ then

$$\lambda y : U_1 . \ldots \lambda y : U_m . t(u_1 \vec{y}) \ldots (u_n \vec{y}) \Vdash z$$

  Monotony follows easily.
- We take $\eta = var$ defined earlier.

We need first to prove that μ is well defined, that is that it is constant on $\sim$ equivalence classes. We then need to prove that the monoid laws are satisfied.

Proof. Suppose $\iota\colon\Gamma\to\Gamma'$, and $(f,v_1,\dots,v_n)\sim(g,w_1,\dots,w_m)$. By definition of the $\sim$ relation we have

- for all i, $v_i = w_{\iota(i)}$
- for all $a_1,\dots,a_m$, $g(a_1,\dots,a_m) = f(a_{\iota(1)})\dots(a_{\iota(n)})$.

From this we deduce

$$f(v_1(\vec{z}))\dots(v_n(\vec{z})) = f(w_{\iota(1)}(\vec{z}))\dots(w_{\iota(n)}(\vec{z}))$$

and

$$f(w_{\iota(1)}(\vec{z}))\dots(w_{\iota(n)}(\vec{z})) = g(w_1(\vec{z}))\dots(w_m(\vec{z}))$$

And so μ is stable on equivalence classes.

The monoid laws are straightforward. For instance for the associativity law:

$$\begin{tikzcd}
\mathrm{Rea}\bullet(\mathrm{Rea}\bullet\mathrm{Rea}) \arrow[r, "(\mu\bullet id)\circ\alpha"] \arrow[d, "id\bullet\mu"'] & \mathrm{Rea}\bullet\mathrm{Rea} \arrow[d, "\mu"] \\
\mathrm{Rea}\bullet\mathrm{Rea} \arrow[r, "\mu"] & \mathrm{Rea}
\end{tikzcd}$$

It is easy to verify, given contexts Γ,Δ and Θ, a type T, and elements $f\in\mathrm{Rea}_T(\Gamma)$, $\vec{g}\in\mathrm{Rea}_\Gamma(\Delta)$ and $\vec{x}\in\mathrm{Rea}_\Delta(\Theta)$, that $\mu(f,\mu(g_1,\vec{x}),\dots,\mu(g_n,\vec{x}))$ and $\mu(\mu(f,\vec{g}),\vec{x})$ are both equal to the function that takes $\vec{z}\in[\![\Theta]\!]$ and returns

$$f(g_1(x_1(\vec{z}))\dots(x_m(\vec{z})))\dots(g_n(x_1(\vec{z}))\dots(x_m(\vec{z})))$$

■

Lemma 122 The presheaf Rea is a Σ_λ-monoid, that is the Σ_λ-monoid structure of Rea and the $\bullet$-monoid structure are compatible.

Proof. We need to show that the following diagram commutes:

$$\begin{tikzcd}
\Sigma_\lambda(\mathrm{Rea})\bullet\mathrm{Rea} \arrow[r, "st"] \arrow[d, "ev\bullet id"'] & \Sigma_\lambda(\mathrm{Rea}\bullet\mathrm{Rea}) \arrow[r, "\Sigma_l(\mu)"] & \Sigma_l(\mathrm{Rea}) \arrow[d, "ev"] \\
\mathrm{Rea}\bullet\mathrm{Rea} \arrow[rr, "\mu"] & & \mathrm{Rea}
\end{tikzcd}$$

Take two contexts Γ,Δ, a type T, an element $v\in\Sigma_\lambda(\mathrm{Rea})_T(\Delta)$ and a substitution $\sigma\in\mathrm{Rea}_\Delta(\Gamma)$, where we set $\sigma(y_1)=v_1,\dots,\sigma(y_m)=v_m$ if $\Delta = y_1\colon U_1,\dots,y_m\colon U_m$.

We proceed by cases on the structure of v.

- **symb**: $v = f \in \Sigma$. In this case, both sides of the diagram are trivially equal to the function that sends $\vec{w} \in [\![\Gamma]\!]$ to I_f.
- **app**: $v = (a, b)$ with $a \in \mathrm{Rea}_{T\to U}(\Gamma)$ and $b \in \mathrm{Rea}_T(\Gamma)$. On one side we have

  $$\mu(ev(a,b),\sigma) = \mu(\vec{x} \mapsto a(\vec{x})(b(\vec{x})),\sigma)$$

  which is finally equal to the function which sends $\vec{z} \in [\![\Gamma]\!]$ to

  $$a(v_1(\vec{z}))\ldots(v_m(\vec{z}))(b(v_1(\vec{z}))\ldots(v_m(\vec{z})))$$

  On the other hand we have

  $$ev(\mu(a,\sigma),\mu(b,\sigma)) = ev(\vec{z} \mapsto a(v_1(\vec{z}))\ldots(v_m(\vec{z})), \vec{z} \mapsto b(v_1(\vec{z}))\ldots(v_m(\vec{z})))$$

  which is equal to the function that sends $\vec{z} \in [\![\Gamma]\!]$ to

  $$a(v_1(\vec{z}))\ldots(v_m(\vec{z}))(b(v_1(\vec{z}))\ldots(v_m(\vec{z})))$$
- **abs**: trivial, as ev is the identity in this case.

■

Theorem 123 The realizability model is a $\mathcal{R} \cup \beta$-model.

Proof. To show this we need to build an interpretation function from $\mathcal{T}rm$ to Rea *that respects reduction.* For the interpretation function, if $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ we simply take the function that takes $t \in \mathcal{T}rm_T(\Gamma)$ and returns

$$v_1 \ldots v_n \mapsto (\!|t|\!)_\theta$$

where θ sends x_i to v_i. This function is realized by the term

$$\lambda x_1 : T_1 . \ldots \lambda x_n : T_n . t$$

using property 116. Then we use property 64 to show that it is in fact a model. We need to show 4 properties:

1. The $(\!|\_|\!)$ function defines a morphism of presheafs: We need to verify that for each type T and contexts Γ, Γ', if $\iota : \Gamma \to \Gamma'$ is a morphism of contexts, then the following square commutes:

$$\begin{tikzcd}
\mathcal{T}rm_T(\Gamma) \arrow[r, "(\!|\_|\!)"] \arrow[d, "\mathcal{T}rm(\iota)"'] & \mathrm{Rea}_T(\Gamma) \arrow[d, "\mathrm{Rea}(\iota)"] \\
\mathcal{T}rm_T(\Gamma') \arrow[r, "(\!|\_|\!)"] & \mathrm{Rea}_T(\Gamma')
\end{tikzcd}$$

Let $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, $\Gamma \vdash_{size} t : T$, $v_1 \in [\![T_1]\!], \ldots, v_n \in [\![T_n]\!]$, and θ be the valuation that sends x_i to v_i. We define $\iota(\theta)$ to be the substitution that sends x_i to $v_{\iota(i)}$. It is then easy to show that

$$(\!|\_|\!) \circ \mathrm{Rea}(\iota)(t)(v_1)\ldots(v_n) = (\!|t|\!)_{\iota(\theta)}$$

Then by a simple induction, we may show that

$$(\!|\iota(t)|\!)_\theta = (\!|t|\!)_{\iota(\theta)}$$

2. This morphism is the initial morphism from $\mathcal{T}rm$. This again is a straightforward induction.

3. $(\!|\_|\!)$ respects the $\bullet$-monoid structure. Take a term $t \in \mathcal{T}rm_T(\Gamma)$ and a substitution σ. Set $(\!|\sigma|\!)$ to be the valuation that sends $x \in \mathsf{dom}(\Gamma)$ to $(\!|\sigma(x)|\!)$. Then the substitution lemma (lemma 117) expresses exactly
$$\mu((\!|t|\!), (\!|\sigma|\!)) = (\!|\mu(t, \sigma)|\!)$$
furthermore it is immediate from the definitions that
$$(\!|var_{\mathcal{T}rm}(x_i)|\!) = (\!|x_i|\!) = \vec{v} \mapsto v_i = var_{\mathrm{Rea}}(x_i)$$
for each i.

4. Finally we need to show that if $l \to r$ is a rewrite rule in $\mathcal{R} \cup \beta$, then $(\!|l|\!)_\theta = (\!|r|\!)_\theta$ for each θ. But this is an immediate consequence of the correctness lemma 118.

■

We believe that the realizability model is a candidate for the intuitive reading of the semantics of size-based termination: in this framework, abstractions are read as meta-theoretical functions, and application is simply the function application. Care must be taken, using the realizability framework, to allow inductive types like Brouwer ordinals to have their intended interpretation, without resorting to very large ordinals. This corresponds to the intuition "every ordinal in computer science is countable", which can be seen as an ordinal-theoretic counterpart of the Church thesis "every function is general recursive". The size annotations in types then correspond to a restriction of the behavior of the semantic elements: a term of type $Nat^\alpha \to Nat^\alpha$ can not possibly sent an element to a strictly larger one.

In the next chapter we see that this interpretation is sufficient to apply the labelling technique, using a termination criterion that operates by precedence on terms.

5

Correctness of the Criterion

In this chapter we show that if a higher-order rewrite system satisfies the size-based termination criterion of definition 40, then the system labeled with the algebra exposed in chapter 4 is terminating relatively to the structural rules, and can furthermore be shown to be so with a simple (and general) criterion.

The criterion simply states that if a (algebraic higher-order) rewrite system matches on (strictly) positive constructors, that for some well-founded order *compatible with the structural rules*, every symbol in the right hand side is strictly smaller than the head symbol in the left hand side, then the rewrite system is normalizing relatively to the structural rules. The criterion is an adaptation to the case of relative normalization of the General Schema [JO91], by Jouannaud and Okada, extended by Blanqui, Jouannaud and Okada [BJO02] and further by Blanqui to *rewriting modulo* [Bla03] to allow rewriting relative to *equations*, that is rules that go in both directions, an important sub-case of relative normalization. In this framework, recursive calls are also allowed, if made on *structurally smaller terms*. These structural calls are not necessary to prove correction of our size based criterion however, as all the information necessary to show decrease is contained in the model, which in turn is explicit in the labels we add to the function symbols. The criterion presented here can be seen as a generalization of the first order *precedence termination* criterion. The criterion described in Blanqui [Bla03] can not *as such* be applied to our problem, as it is defined for equations and not rewrite rules, and requires equivalence classes of terms to be *finite*, which is not our case. However our approach is very similar in spirit.

To apply this criterion to our labeled system, we need to describe a precedence on function symbols. The natural way to do this, and indeed the motivation for this approach, is to look at an order on the labels themselves. In the case of terms labelled with the realizability model, we wish order labels by the pointwise extension of the natural size order, carried to tuples by the *status* of the function symbol considered (see definition 37). However this will not suffice to be able to apply the criterion. We also need this order to be *compatible* with the application of structural rules. For this reason we will not simply take the ranks of labels, ordered by the status $>_f$, but allow a number of instantiations and weakenings to occur beforehand. To show that this order is still well-founded, we apply a lemma from Doornbos and von Karger that allows us to prove well-foundedness of an order by showing certain commutation properties.

5.1 The Termination Criterion

Theorem 124 (The precedence criterion for relative termination)

Let $\Sigma = \mathcal{D} \cup \mathcal{C}$ be a signature and τ a type assignment for Σ. Let $>_{\mathcal{B}}$ be a well-founded preorder on base types, and $>_{prec}$ be a well-founded preorder on elements $f \in \mathcal{D}$. Let $\mathcal{R}$ be an arbitrary left algebraic rewrite system and $\mathcal{S}$ be an *algebraic* rewrite system such that for all rules $f\vec{l} \rightarrow r \in \mathcal{R} \cup \mathcal{S}$, $f \in \mathcal{D}$. For each $f \in \mathcal{D}$ we suppose given a natural number n_f. We furthermore suppose that the following conditions are satisfied:

- The constructors are strictly positive.
- All terms in matched position are constructor terms.
- For each rule $f\ l_1 \dots l_n \rightarrow r \in \mathcal{R}$, $n = n_f$.
- The system $\mathcal{R}$ satisfies the *decrease condition*: for each rule $f\ l_1 \dots l_n \rightarrow r \in \mathcal{R}$, every function symbol $g \in \mathcal{D}$ that appears in r verifies

$$f >_{prec} g$$

- The preorder $>_{prec}$ is *compatible* with $\mathcal{S}$: if $f\ t_1 \dots t_n \rightarrow^*_{\mathcal{S}} u$, then $u = g\ u_1 \dots u_m$ and:
  - for every $h \in \Sigma$:

$$g >_{prec} h \Rightarrow f >_{prec} h$$

  - for every j, there exists i such that $t_i \rightarrow^*_{\mathcal{S}} u_j$, and the type of the i-th argument of f is equal to the type of the j-th argument of g.

Then for all Γ, t, T:

$$\Gamma \vdash t : T \Rightarrow t \in \mathcal{SN}_{\mathcal{R}\cup\beta/\mathcal{S}}$$

The compatibility condition is quite restrictive. However it is sufficient in our case, as we deal with structural rewrite rules that are of the form $f \rightarrow g$ with $f, g \in \Sigma$. We write $\rightarrow_{\mathcal{S}^*;\mathcal{R}\beta}$ for $\rightarrow^*_{\mathcal{S}} \circ \rightarrow_{\mathcal{R}\cup\beta}$. We sometimes write $\rightarrow$ for $\rightarrow_{\mathcal{S}^*;\beta}$ and $\rightarrow^*$ for $\rightarrow^*_{\mathcal{S}^*;\mathcal{R}\cup\beta}$.

The proof proceeds in a very similar manner as the proof of theorem 18.

Definition 125 A term is a *value* if it is of the form

- $\lambda x : T.t$
- $c\ t_1 \dots t_m$ with $c \in \mathcal{C}$ and m smaller or equal to the arity of c.
- $f\ t_1 \dots t_k$ with $k < n_f$

A term is neutral if it is not a value.

A *reducibility candidate* is a set $C \subset \mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$ that verifies:

- Stability by reduction: if $t \in C$ then for every u such that $t \rightarrow_{\mathcal{S}^*;\mathcal{R}\beta} u$, $u \in C$.
- Sheaf condition: if t is a neutral term and for every u such that $t \rightarrow_{\mathcal{S}^*;\mathcal{R}\beta} u$, $u \in C$ then $t \in C$.

The proof is a variation on the proof of theorem 18 in the preliminary chapter. The goal is to interpret each type as a reducibility candidate. We shall need the *Knaster-Tarski* fixed-point theorem to be able to assert the existence of certain reducibility candidates.

Property 126 (Knaster & Tarski [Tar55])
Let L be a complete lattice. If $\psi : L \to L$ is a monotone function, then it has a least fixed-point.

This theorem is folklore, and we will omit the proof.

Proof.
First we define the interpretation of sets as follows. Given two subsets X and Y of $\mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$, we define

$$X \multimap Y := \{t \in \mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}} \mid \forall u \in X, t\ u \in Y\}$$

Now we proceed by induction on $>_{\mathcal{B}}$ to define $\mathcal{I}_B$ for each $B \in \mathcal{B}$. If E is the set of $C \in \mathcal{B}$ such that $C \simeq_{\mathcal{B}} B$, then we define F_B as a function from $\mathcal{P}(\mathcal{T}rm)^E$ to $\mathcal{P}(\mathcal{T}rm)$ as:

$$F_B(X) = \{t \in \mathcal{SN} \mid t \to^* c\ t_1 \dots t_n \Rightarrow \tau_c = T_1 \to \dots T_n \to B \wedge t_1 \in [\![T_1]\!]_X \dots t_n \in [\![T_n]\!]_X\}$$

Where $X \in \mathcal{P}(\mathcal{T}rm)^E$ and $[\![\_]\!]_X$ is defined by

- $[\![C]\!]_X = \mathcal{I}_C$ if $C <_{\mathcal{B}} B$,
- $[\![C]\!]_X = X(C)$ if $C \simeq_{\mathcal{B}} B$
- $[\![C]\!]_X = \mathcal{N}$ the set of all neutral terms otherwise
- $[\![T \to U]\!]_X = [\![T]\!]_X \multimap [\![U]\!]_X$.

Then by the Tarski fixed-point theorem (theorem 126) applied to the lattice of subsets of $\mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$, the function $F_\_ : \mathcal{P}(\mathcal{T}rm)^E \to \mathcal{P}(\mathcal{T}rm)^E$ has a fixed point, which we set to be $\mathcal{I}_\_$.
We show that $\mathcal{I}_B$ satisfies the girard conditions (definition 21)

- Strong normalization: by definition.
- Stability by reduction: if t is in $\mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$ then so are all its reducts. Furthermore if $t \to_{\mathcal{S}^*;\mathcal{R}\beta} t'$ then all the reducts of t' are in particular reducts of t, which allows us to conclude.
- Sheaf condition. Suppose that t is neutral and every (one step) reduct of t is in $\mathcal{I}_C$. It is clear that t is in $\mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$. Suppose $t \to^* c\ t_1 \dots t_n$. Then by neutrality $t \neq c\vec{t}$ and so $t \to t' \to^* c\ t_1 \dots t_n$ and we may conclude by the hypothesis $t' \in \mathcal{I}_B$ that the t_i are in the appropriate sets.

We must then show that if C_1 and C_2 are reducibility candidates, then so is $C_1 \multimap C_2$. We proceed as in the proof of lemma 21.

- SN: by definition.
- Stability by reduction: suppose that $t \in C_1 \multimap C_2$. Let $t \to t'$ and $u \in C_1$. Then $t\ u \to t'\ u$, and by the induction hypothesis, $t'\ u$ is in C_2. As u was taken to be arbitrary, $t' \in C_1 \multimap C_2$.
- Sheaf condition: let t be neutral, and suppose that $u \in C_1$. Neutrality of t implies that all reducts of $t\ u$ are of the form $t'\ u$ or $t\ u'$ with t' a reduct of t and u' a reduct of u. We may proceed by well-founded induction on the reducts of u (which is in $\mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$) to show that $t\ u' \in C_2$, and by hypothesis $t'\ u \in C_2$. Therefore $t\ u$ is in C_2 by hypothesis, as $t\ u$ is again neutral. From this we conclude that $t \in C_1 \multimap C_2$.

We may now finally conclude that

$$\forall T \in \mathcal{T}, [\![T]\!] \text{ satisfies the Girard conditions}$$

Now suppose $\Gamma \vdash t : T$ and $\theta \models \Gamma$. We have

$$\theta \models t : T$$

We show this in two steps:

1. first we show that if $f \in [\![\tau_f]\!]$ for each f occurring in t, then $t\theta \in [\![T]\!]$.
2. then we show that each f is indeed in $[\![\tau_f]\!]$.

The proof of the first point proceeds by induction on the typing derivation and is identical to the proof of theorem 18, except for the **symb** case, which follows by hypothesis.

It remains to show that for every function symbol $f \in \Sigma$, $f \in [\![\tau_f]\!]$.

We proceed by cases:

- $c \in \mathcal{C} \backslash \mathcal{D}$: Easy by definition of $[\![B]\!]$.
- $f \in \mathcal{D} \backslash \mathcal{C}$. Suppose that $\tau_f = \Theta_1 \to \ldots \to T_n \to B$ and $t_1, \ldots, t_n$ in $[\![T_1]\!], \ldots, [\![T_n]\!]$ respectively. We need to show that $t = f\ t_1 \ldots t_n$ is in $[\![B]\!]$. Using the fact that $f\vec{t}$ is neutral, we only need to show that all reducts are in $[\![B]\!]$. We prove that for all $t' = g\ u_1 \ldots u_m$ such that $f\vec{t} \to^*_{\mathcal{S}} g\vec{u}$, t' is in $[\![B]\!]$. By the strong normalization condition, $t_i \in \mathcal{SN}$ for each i. We can therefore proceed by induction on the tuple $(f, t_1, \ldots, t_n)$ ordered by $>_{prec} \times \to^+_{\mathcal{S}^*;\mathcal{R}\beta} \times \ldots \times \to^+_{\mathcal{S}^*;\mathcal{R}\beta}$ (the product order of $>_{prec}$ and $\mathcal{S}^*;\mathcal{R}\beta$-reduction). There are three possible kinds of reducts t' for t:
  - $f\ t_1 \ldots t'_i \ldots t_n$ with $t_i \to t'_i$. We can conclude by induction.
  - $f\vec{t} \to^*_{\mathcal{S}} g\ u_1 \ldots u_m \to_{\mathcal{S}^*;\mathcal{R}\beta} g\ u_1 \ldots u'_i \ldots u_m$ with $u_i \to_{\mathcal{S}^*;\mathcal{R}\beta} u'_i$. In this case, by the compatibility hypothesis, there is some j such that $t_j \to^*_{\mathcal{S}} u_i$, and therefore $t_j \to^* u'_i$, and we may conclude again by the induction hypothesis.
  - $f\ \vec{t} \to^*_{\mathcal{S}} g\ u_1 \ldots u_m$ and there is some rule $g\ l_1 \ldots l_k \to r \in \mathcal{R}$ with $k \leq m$, a substitution θ such that $l_1\theta = u_1, \ldots, l_k\theta = u_k$, and $t' = r\theta\ u_{k+1} \ldots u_m$. Now we have each u_i computable, as t_i is computable for each i, $t_j \to^*_{\mathcal{S}} u_i$ for some j and stability by reduction. Now by typeability hypothesis, we have $\Gamma \vdash r : T$. Let us show that $\theta \models \Gamma$: we proceed by induction on the structure of l_i.
    - $l_i = x$, in this case $\theta(x) = u_i$, and we may conclude by computability of u_i.
    - $l_i = c\ p_1 \ldots p_n$, in this case we have $u_i = c\ v_1 \ldots v_n$, and u_i computable implies v_k computable, and we may conclude by induction hypothesis.

    It remains to show that $r\theta$ is in $[\![T]\!]$. For this we proceed by induction on the derivation of $\Gamma \vdash r : T$. We only need to treat the **symb** case: in this case, $r = hv_1 \ldots v_n$, with $g >_{prec} h$, and $v_i\theta$ computable. We conclude by the induction on f, as by compatibility $f >_{prec} h$.
- $c \in \mathcal{D} \cap \mathcal{C}$: We first show the following lemma: if $c \in \mathcal{D} \cap \mathcal{C}$ is of type $T_1 \to \ldots \to T_n \to B$, $t_1, \ldots, t_n$ are such that $t_i \in [\![T_i]\!]$, and for every u such that $c\ t_1 \ldots t_n \to_{\mathcal{S}^*;\mathcal{R}\beta} u$, $u \in [\![B]\!]$ then

  $$c\ t_1 \ldots t_n \in [\![B]\!]$$

  To show this we simply need to show

- $c\ \vec{t} \in \mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$. This is straightforward from the hypothesis and $[\![B]\!] \subseteq \mathcal{SN}_{\mathcal{R}\beta/\mathcal{S}}$.
- If $c\ \vec{t} \rightarrow^* d\ \vec{u}$ with d a constructor of B of type $U_1 \rightarrow \ldots \rightarrow U_m \rightarrow B$, then for each $1 \leq j \leq m$, $u_j \in [\![U_j]\!]$. Suppose $c\ \vec{t} \rightarrow^*_{\mathcal{S}} d\ \vec{u}$. Then by the compatibility hypothesis for each j, there is some t_i such that $t_i \in [\![U_j]\!]$ and $t_i \rightarrow^*_{\mathcal{S}} u_j$. By stability by reduction of computable terms, $u_j \in [\![U_j]\!]$. Now suppose that $c\ \vec{t} \rightarrow^+_{\mathcal{S}^*;\mathcal{R}\beta} d\ \vec{u}$. Then by hypothesis $d\ \vec{u} \in [\![B]\!]$ and by definition $u_j \in [\![U_j]\!]$.

Now to show that $c \in [\![\tau_c]\!]$ we may proceed as in the case $f \in \mathcal{D} \backslash \mathcal{C}$.

■

5.2 Termination of the Labelled System

With this theorem in hand, and given a rewrite system satisfying the termination conditions, we wish to prove termination of the labeled system as described in theorem 79 using the semantics described by proposition 120, and then apply the main theorem of semantic labelling (theorem 84) and prove termination of the labelled system. To do this we need to find the appropriate precedence on function symbols.

Definition 127 Let Σ be a signature and $\mathcal{R}$ be a rewrite system that satisfies the criterion of theorem 41. Consider the labeled system $\overline{\mathcal{R}}$ as defined in definition 78, labeled using the realizability model of theorem 123. Given a function symbol $f \in \mathcal{D}$ of type $T_1 \rightarrow \ldots \rightarrow T_k \rightarrow T_f$, we extend $>_f$ to elements of $\mathsf{Rea}_{T_1}(\Gamma) \times \ldots \times \mathsf{Rea}_{T_k}(\Gamma)$ for each Γ by taking

$$(a_1, \ldots, a_k) >_f (b_1, \ldots, b_k) \Leftrightarrow \forall \vec{x} \in [\![G]\!],\ (\mathrm{rk}_{T_1}\, a_1(\vec{x}), \ldots, \mathrm{rk}_{T_k}\, a_k(\vec{x})) >_f (\mathrm{rk}_{T_1}\, b_1(\vec{x}), \ldots, \mathrm{rk}_{T_k}\, b_k(\vec{x}))$$

and we define the order $>_{prec}$ on labels of f by the transitive closure of

$$>_{prec} = \geq_{inst} \circ (\geq_{wk} \circ >_f)$$

The order $>_{Prec}$ on labeled function symbols (in $\mathcal{D}$) is defined by

$$f_l >_{Prec} g_{l'} \Leftrightarrow f >_{\mathcal{D}} g\ \ \vee\ \ f \simeq_{\mathcal{D}} g \wedge l >_{prec} l'$$

The $>_{Prec}$ order is exactly what is needed to apply the termination theorem given above to our labelled system. We show below that it is compatible (in the sense given in the termination theorem 124) with the *Struct* rewrite system, and that it is well-founded.

The strict order on the components of the **Rea** functor is trivially well-founded, as it is the strict point-wise order of a function into a well-founded domain. However it is far from trivial that the $>_{prec}$ order is well-founded also. Indeed, arbitrary weakening of the context may occur in alternation with the instantiation of certain variables in the context.

To prove well-foundedness of $>_{prec}$, we shall apply a combinatorial lemma which allows us to prove well-foundedness of the union of two orders, provided they are well-founded themselves and satisfy a certain compatibility property. The combinatorial lemma was first stated and proved by Doornbos and von Karger [DK98].

Lemma 128 (Doornbos & von Karger)
Let $>_R$ and $>_B$ be well-founded relations on a set E. Then if

$$>_R \circ >_B \subseteq >_B \circ (>_B \cup >_R)^* \cup\ >_R$$

Then $>_R \cup >_B$ is well-founded.

We will not reproduce the proof here, see [DK98] for a complete demonstration. We can then show well-foundedness of $>_{prec}$ by a simple combinatorial proof. We prove inclusion properties of the orders involved, which shall also be useful for the proof of compatibility.

Lemma 129 The following inclusions are verified:

$$\geq_{wk} \circ >_{inst} \ \subseteq \ (>_{inst} \circ \geq_{wk}) \cup \geq_{wk} \tag{a}$$

$$>_f \circ >_{inst} \ \subseteq >_{inst} \circ >_f \tag{b}$$

Proof. First let us recall that the orders involved are on f-labels, that is tuples of functions, where each function is in the same domain. The $\geq_{wk}$ order allows increase of the domain, $>_{inst}$ allows instantiation of one of the arguments by a function, which may depend on other elements of the domain.

Inclusion (a): take $(a_1, \ldots, a_n) \geq_{wk} (b_1, \ldots, b_n) >_{inst} (c_1, \ldots, c_n)$. There are two possibilities:

- each a_i is in $[\![\Gamma, x\colon A, \Gamma']\!] \to [\![T_i]\!]$, b_i in $[\![\Gamma, x\colon A, \Gamma', \Delta]\!] \to [\![T_i]\!]$ and $c_i \in [\![\Gamma, \Gamma', \Delta]\!] \to [\![T_i]\!]$ with the following relations, for each $\vec{x} \in [\![\Gamma]\!], x \in [\![A]\!], \vec{y} \in [\![\Gamma']\!]$ and $\vec{z} \in [\![\Delta]\!]$, and some $d \in [\![\Gamma]\!] \to [\![A]\!]$.

  $$a_i(\vec{x})(x)(\vec{y}) = b_i(\vec{x})(x)(\vec{y})(\vec{z})$$

  $$b_i(\vec{x})(d(\vec{x}))(\vec{y})(\vec{z}) = c_i(\vec{x})(\vec{y})(\vec{z})$$

  Then taking $a'_i(\vec{x})(\vec{y}) = a_i(\vec{x})(d(\vec{x}))(\vec{y})$ we have

  $$(a_1, \ldots, a_n) >_{inst} (a'_1, \ldots, a'_n) \geq_{wk} (c_1, \ldots, c_n)$$

- Otherwise a_i is in $[\![\Gamma]\!] \to [\![T_i]\!]$ and b_i in $[\![\Gamma, \Delta, y\colon A, \Delta']\!] \to [\![T_i]\!]$, $c_i \in [\![\Gamma, \Delta, \Delta']\!] \to [\![T_i]\!]$ with

  $$a_i(\vec{x}) = b_i(\vec{x})(\vec{y})(y)(\vec{z})$$

  and

  $$c_i(\vec{x})(\vec{y})(\vec{z}) = b_i(\vec{x})(\vec{y})(d(\vec{x})(\vec{y}))(\vec{z})$$

  And so, taking $y = d(\vec{x})(\vec{y})$:

  $$c_i(\vec{x})(\vec{y})(\vec{z}) = b_i(\vec{x})(\vec{y})(d(\vec{x})(\vec{y}))(\vec{z}) = a_i(\vec{x})$$

  and so we directly have

  $$(a_1, \ldots, a_n) \geq_{wk} (c_1, \ldots, c_n)$$

The proof of (b) is simpler:take $(a_1, \ldots, a_n) >_f (b_1, \ldots, b_n) >_{inst} (c_1, \ldots, c_n)$ with a_i, b_i functions in $[\![\Gamma, x\colon A, \Delta]\!] \to [\![T_i]\!]$ and c_i a function in $[\![\Gamma, \Delta]\!] \to [\![T_i]\!]$, as well as a function $d\colon [\![\Gamma]\!] \to [\![A]\!]$ with

$$\forall i, \vec{z} \in [\![\Gamma, x\colon A, \Delta]\!],\ \mathrm{rk}(\vec{a})(\vec{z}) >_f \mathrm{rk}(\vec{b})(\vec{z})$$

and

$$\forall i, \vec{x} \in [\![\Gamma]\!], \vec{y} \in [\![\Delta]\!],\ c_i(\vec{x})(\vec{y}) = b_i(\vec{x})(d(\vec{x}))(\vec{y})$$

Then it suffices to take a'_i defined by $a'_i(\vec{x})(\vec{y}) = a_i(\vec{x})(d(\vec{x}))(\vec{y})$ and we obtain

$$(a_1, \ldots, a_n) >_{inst} (a'_1, \ldots, a'_n) >_f (c_1, \ldots, c_n)$$

. ■

This structural lemma will allow us to apply the lemma from Doornbos and Von Karger, and prove well-foundedness of $>_{prec}$[1].

Property 130 The order $>_{prec}$ is well-founded.

Proof. First notice that well-foundedness of $>_R \cup >_B$ trivially implies that of $\geq_B \circ >_R$. We therefore apply lemma 128, taking $>_R$ to be $\geq_{wk} \circ >$ and $>_B$ to be $>_{inst}$.

We show the stronger statement

$$(\geq_{wk} \circ >_f) \circ >_{inst} \subseteq (>_{inst} \circ \geq_{wk} \circ >_f) \cup (\geq_{wk} \circ >_f)$$

By inclusion (e) from lemma 129, we have $>_f \circ >_{inst} \subseteq >_{inst} \circ >_f$, which gives

$$\geq_{wk} \circ >_f \circ >_{inst} \subseteq \geq_{wk} \circ >_{inst} \circ >_f$$

Then, applying the inclusions (d) and (e) we have

$$\geq_{wk} \circ >_f \circ >_{inst} \subseteq (>_{inst} \circ \geq_{wk} \cup \geq_{wk}) \circ >_f$$

And finally

$$(>_{inst} \circ \geq_{wk} \cup \geq_{wk}) \circ >_f \subseteq (>_{inst} \circ \geq_{wk} \circ >_f) \cup (\geq_{wk} \circ >_f)$$

Now we need to check that both $>_{inst}$ and $\geq_{wk} \circ >_f$ are well-founded. In the first case it is a simple induction on the size of the context. In the second case, for every type T, $[\![T]\!]$ is non-empty, so given a sequence of elements $l_1 \geq_{wk} \circ >_f l_2 \geq_{wk} \circ >_f \ldots$, we can take a tuple $\vec{x}$ and a sequence of tuples $(\vec{y}_k)_{k\in\mathbb{N}}$ such that if $l_n(\vec{v}) = (a^n_1(\vec{v}), \ldots, a^n_m(\vec{v}))$, then $\mathrm{rk}(l_1(\vec{x})) >_f \mathrm{rk}(l_2(\vec{x})(\vec{y}_0)) >_f \mathrm{rk}(l_1(\vec{x})(\vec{y}_0)(\vec{y}_1)) >_f \ldots$ which is a contradiction by well-foundedness of $>_f$. ■

And we can finally prove the well-foundedness of the precedence.

Corollary 131 The order $>_{Prec}$ is well-founded.

Proof. It suffices to take the lexicographic combination of $>_{\mathfrak{D}}$ and $>_{prec}$, both of which are well-founded, the first by hypothesis and the second by property 130.

■

To apply theorem 124, we need to show *compatibility* of the preorder $>_{Prec}$ just described, and the *Struct* rewrite system described in chapter 3.

Lemma 132 (Compatibility)
The preorder $>_{Prec}$ is compatible with *Struct*.

[1] We thank Frederic Blanqui, Gilles Dowek and Guillaume Burel for technical contributions to the proof of this lemma.

Proof. Remember that by definition $\mathit{Struct} = \mathit{Inst} \cup \mathit{Wk} \cup \mathit{Decr}$. First note that Decr is trivial, as we are working in a model rather than a premodel. Now if $f_l\ t_1 \dots t_n \to^*_{\mathit{Struct}} u$, then as these rules only modify labels of function symbols, we have $u = g\ u_1 \dots u_n$ with $g = f_{l'}$ and $t_i \to^*_{\mathit{Struct}} u_i$. If in addition $g >_{\mathit{Prec}} h$, then either $h = f'_m$ with $f >_{\mathcal{D}} f'$, in which case $f_l >_{\mathit{Prec}} h$ or $h = f_m$ with $l \geq_{\mathit{struct}} l'$ and $l' >_{\mathit{prec}} m$. In this case we need to show that $l >_{\mathit{prec}} m$.

We have $\geq_{\mathit{struct}} = \geq_{\mathit{inst}} \cup \geq_{\mathit{wk}}$ and $>_{\mathit{prec}} = \geq_{\mathit{inst}} \circ \geq_{\mathit{wk}} \circ >_f$. We therefore need to show that

$$(\geq_{\mathit{inst}} \cup \geq_{\mathit{wk}}) \circ (\geq_{\mathit{inst}} \circ \geq_{\mathit{wk}} \circ >_f) \ \subseteq \ \geq_{\mathit{inst}} \circ \geq_{\mathit{wk}} \circ >_f$$

This can be shown using the inclusions of lemma 129.

■

The only thing left to show is that the rules do indeed admit decrease with respect to $>_{\mathit{Prec}}$, which can be shown by induction on the typing (in the size-type framework) of the rule.

Lemma 133 (Decrease)
The rewrite system $\mathcal{R} \cup \mathit{Struct}$ admits the decrease condition with respect to the $>_{\mathit{Prec}}$ preorder.

Proof. Consider a rule $f\vec{l} \to r$ in $\mathcal{R}$ typed by a context Γ, and $\phi \in \mathrm{Rea}_\Gamma(\Delta)$ be some valuation. Then

$$\overline{f\vec{l}}^{\phi} = f_m \overline{\vec{l}}^{\phi}$$

with $m = (\llparenthesis \Gamma \vdash l_1 \rrparenthesis_\phi, \dots, \llparenthesis \Gamma \vdash l_k \rrparenthesis_\phi)$. Suppose that $g\ t_1 \dots t_k$ appears in r. We show that

$$g_n\ \overline{t_1}^{\phi'} \dots \overline{t_k}^{\phi'}$$

appears at the same position in $\overline{r}^{\phi}$ with ϕ' the extension of ϕ to $\mathrm{Rea}_{\Gamma,\Theta}(\Delta, \Theta)$ which sends variables in Θ to themselves and $n = (\llparenthesis \Gamma, \Theta \vdash t_1 \rrparenthesis_{\phi'}, \dots, \llparenthesis \Gamma, \Theta \vdash t_k \rrparenthesis_{\phi'})$ and that furthermore $f_m >_{\mathit{Prec}} g_n$.

We proceed by induction on the derivation of $\Gamma \vdash_{\mathit{size}} r : T$ in the size framework.

- **var**: $g\ \vec{t}$ cannot occur as the subterm of a variable.

- **abs**: we have $r = \lambda x : T.t$. By definition

  $$\overline{\lambda x : T.t}^{\phi'} = \lambda x : T.\overline{t}^{\phi'^x_x}$$

  and we conclude by simple induction hypothesis on t.

- **app,symb**: We treat two cases: $r = t\ u$ with $t \neq g\ t_1 \dots t_i$ for some $g, \vec{t}, i$ with $i \geq k$. In this case we conclude by simple application of the induction hypothesis. Otherwise $r = g\ t_1 \dots t_i$ with $i \geq k$, and $\overline{g\ t_1 \dots t_i}$ is as above.

  Now either $f >_{\mathcal{D}} g$ and we are finished, or $f \simeq_{\mathcal{D}} g$ and we proceed as in definition 111. By the decrease condition we have $\Gamma \vdash_{\mathit{size}} l_i : B_i^{a_i}$ for some base type B_i. Take arbitrary $\vec{v}$ in $\llbracket \Delta \rrbracket$. Define $\phi(\vec{v})$ to be the valuation that sends every x in Γ to $\phi(x)(\vec{v})$. By lemma 102 $\phi(\vec{v})$ is a Γ-minimal size-valuation, and by lemma 104, for each i

  $$\mathrm{rk}\ \llparenthesis l_i \rrparenthesis_{\phi(\vec{v})} = \llparenthesis a_i \rrparenthesis_{\hat{\phi}(\vec{v})}$$

  Furthermore, take any $\vec{w}$ in $\llbracket \Theta \rrbracket$. By the definition of the rule for abstraction (figure 1.1) we have for every $x \in \mathsf{dom}(\Theta)$, $\Theta(x) = T^\infty$ for some T, from which we have that $\phi'(\vec{v}, \vec{w})$ is a

Γ, Θ-size-valuation, by taking $\hat{\phi}'(\vec{v}, \vec{w})$ the valuation that sends variables α in Γ to $\hat{\phi}(\vec{v})(\alpha)$ and all other variables to $\mathbf{\Omega}$. Then applying lemma 107 gives

$$\text{rk } (\!|t_i|\!)_{\phi'(\vec{v},\vec{w})} \leq (\!|b_i|\!)_{\hat{\phi}'(\vec{v},\vec{w})}$$

Where $\Gamma, \Theta \vdash_{size} t_i : B_i^{b_i}$. By the decrease condition and the fact that $>_f$ respects $\geq$, we therefore may conclude

$$\text{rk } (\!|\vec{l}|\!)_{\phi(\vec{v})} >_f \text{rk } (\!|\vec{t}|\!)_{\phi'(\vec{v},\vec{w})}$$

which gives

$$(\!|\Gamma \vdash \vec{l}|\!)_{\phi} >_{prec} (\!|\Gamma, \Theta \vdash \vec{t}|\!)_{\phi'}$$

By definition of the $>_{prec}$ order. This finally gives

$$f_m >_{Prec} g_n$$

■

Notice in fact that we only have used $\geq_{wk} \circ >_f$ instead of the full $>_{prec}$ order. However we can not weaken our definition, as taking $>_{prec} = \geq_{wk} \circ >_f$ would not allow us to prove the compatibility with *Struct* (lemma 132).

Theorem 134 Suppose that the conditions of theorem 41 are satisfied. The system $\mathcal{R} \cup \beta$ is terminating on well-typed terms.

Proof. The proof applies the fundamental theorem of semantic labelling 84 using the Rea algebra. We prove relative normalization of $\overline{\mathcal{R}} \cup \beta$ with respect to *Struct*, by applying the precedence termination theorem 124 with $>_{Prec}$ as the precedence. Compatibility is assured by lemma 132, and by the previous lemma the decrease condition is verified. The other conditions of the termination theorem trivially follow from the conditions of theorem 41.

■

We have given a two-tier proof of correctness of the size-type criterion for termination, by first giving a sufficient condition in terms of termination of the labelled system, and then applying a generic termination lemma based on computability predicates. This is a way of separating concerns: the "user" of the termination theorem does not need to worry about computability or the Girard conditions when building the realizability semantics of chapter 4, as this is all handled in the proof of theorem 124. We only need to find an appropriate precedence, and prove compatibility with the structural rules.

While powerful, this approach is not complete: The system described in example 9 *can not* be shown to be terminating with the "labelling + precedence" termination approach. Indeed all information on the semantics of the argument of f in the recursive call is hidden by the abstraction. This is a common difficulty in higher-order rewriting. The next part describes a criterion that may deal with this kind of "non local" decrease information. Let us note that the size-type systems described, for example, by Barthe *et al* and Abel can easily handle these kinds of systems, though of course they are not complete either, as they are decidable criteria. The presentation of semantic labelling by Hamana [Ham07] is complete for left-algebraic rewriting, but at the cost of having to label abstraction and application with the semantics of their arguments, and as previously noted, the labelling of β-reduction is *not* β-reduction.

5.3 Further applications

Let us give another application of this approach. We will use the fundamental theorem of semantic labelling (theorem 84), coupled with the relative normalization theorem (theorem 124) to prove a well-known theorem: the termination of a first-order rewrite system implies termination of the currified system with β-reductions. This was proven independently by Okada [Oka89] and Breazu-Tannen and Gallier [GBT89] for the simply typed λ-calculus and system F, respectively. We present a novel proof for the case of the monomorphic λ-calculus. The idea is to label (higher-order) terms with a syntactic model of the first-order terms, ordered by the first order rewrite system.

Definition 135 Given a signature Σ, a *first-order arity ar* is a function $ar\colon\Sigma\to\mathbb{N}$. Given such an arity, the set of *first-order terms* is defined by

$$t_1,\dots,t_n \in \mathcal{T}rm^{\Sigma}_{fo} \coloneqq y \mid f(t_1,\dots,t_n)$$

where $f\in\Sigma$ and $ar(f)=n$, and $y\in\mathcal{Y}$ a set of *first order variables*. A *rewrite rule* on first-order terms is a pair $l\to r$ of terms with the free variables of r contained in those of l. We define reduction as for higher-order terms: given a set $\mathcal{R}$ of rewrite rules t *head reduces* to u under $\mathcal{R}$ if there is a rule $l\to r\in\mathcal{R}$ and a substitution θ such that $l\theta=t$ and $u=r\theta$. We define reduction $\to_{\mathcal{R}}$ by context closure and $\to^{+}_{\mathcal{R}}$ and $\to^{*}_{\mathcal{R}}$ as the transitive and symmetric transitive closure like in definition 12.

Now fix a signature Σ an arity ar, and a first-order rewrite system $\mathcal{R}$ on $\mathcal{T}rm^{\Sigma}_{fo}$. We wish to define the currified version of $\mathcal{R}$. We take an injective correspondence $i\colon\mathcal{Y}\to\mathcal{X}$ between first-order variables and higher-order ones. Given a term $t\in\mathcal{T}rm_{fo}$ we define $curry(t)$ by induction on the structure:

- $curry(y)=i(y)$
- $curry(f(t_1,\dots,t_n)) = f\ curry(t_1)\dots curry(t_n)$

And we define the *curryfied rewrite system* by

$$curry(\mathcal{R}) = \{curry(l)\to curry(r)\mid l\to r\in\mathcal{R}\}$$

Take D as single base type, that is $\mathcal{B}=\{D\}$. Define the type assignment τ that sends f to $D\to\dots\to D\to D$ with $n+1$ occurrences of D if $ar(f)=n$. We can prove by induction the following fact:

Lemma 136 If t is a first-order term, and Γ is the context that sends $i(y)$ to D if y is a variable of t, then

$$\Gamma\vdash curry(t)\colon D$$

Proof. Simple induction on t. ■

We can now build the *syntactic model*: the base type D is interpreted as the set of first-order terms, ordered by the rewrite relation. Then arrows are interpreted by the full space of functions, as there are no cardinality restrictions when dealing with first-order rewrite systems. The definition is quite similar to that of the Rea premodel, but quite simpler as there is no need to work in the realizability space, and the syntactic nature of the model makes certain properties hold trivially.

Theorem 137 We can build the following $curry(\mathcal{R})$-premodel $\mathcal{F}o$:

- We map each type T to a set $[\![T]\!]$ as follows:
  - $[\![D]\!] = \mathcal{T}rm_{fo}$, ordered by the relation $\geq_{\mathcal{R}}$ defined by
    $$t >_{\mathcal{R}} u \Leftrightarrow t \to^*_{\mathcal{R}} u$$
  - $[\![T_1 \to T_2]\!] = [\![T_2]\!]^{[\![T_1]\!]}$ the full set of monotonous functions from $[\![T_1]\!]$ to $[\![T_2]\!]$, ordered by the pointwise order.
- We define $\mathcal{F}o$ as the presheaf
  $$\mathcal{F}o_T(\Gamma) := [\![\Gamma]\!] \to [\![T]\!]$$
  With $[\![\Gamma]\!] \to [\![T]\!]$ defined as $[\![T_1]\!] \to \ldots \to [\![T_n]\!] \to [\![T]\!]$ if $\Gamma = x_1 : T_1, \ldots, x_n : T_n$. We take as action under renaming with respect to a morphism $\iota : \Gamma \to \Gamma'$:
  $$\mathcal{F}o(\iota)(f) = \vec{v} \in [\![\Gamma']\!] \mapsto f(v_{\iota(1)}) \ldots (v_{\iota(n)})$$
  as in the definition of Rea 119.
- The morphism $var : \mathsf{dom} \to \mathcal{F}o$ is defined as for the Rea algebra, that is if $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ then $var(x_i)$ is the i-th projection.
- For application and abstraction we take the pointwise application and the identity, as in the definition of the Rea Σ_λ-algebra structure (definition 120). More precisely
  $$app_{\mathcal{F}o}(t, u)(x_1) \ldots (x_n) = t(x_1) \ldots (x_n)\ (u(x_1) \ldots (x_n))$$
  and
  $$abs_{\mathcal{F}o}(t) = t$$
- For the interpretation val of function symbols $f \in \Sigma$, if f us of arity n, then we define $val(f)$ to be the function that takes $t_1 \in \mathcal{T}rm_{fo} \ldots t_n \in \mathcal{T}rm_{fo}$ and returns the term $f(t_1, \ldots, t_n)$. By abuse, we will write f for this function.
- We give a structure of $\bullet$-monoid to $\mathcal{F}o$ exactly in the same manner as for Rea in definition 121.

Proof. We need to verify that $\mathcal{F}o$ is indeed a Σ_l-algebra, that the morphisms are stable by renaming, and that the algebra structure is compatible with the monoid structure. However the structure of this proof is *exactly* analogous of the proof for the Rea presheaf, with the (significant) simplification that we are working in the full function spaces, and therefore do not need to worry about functions being realized. We will therefore omit this tedious proof.

The only notable difference is the definition of the interpretation of functions, and therefore we need to verify that the interpretation of f is indeed in the interpretation of its type, which simply reduces to verifying that f is indeed monotonous. This can in turn be expressed as:

$$\forall t_1, \ldots, t_n, u_1, \ldots, u_n \in \mathcal{T}rm_{fo}\ t_1 \to^*_{\mathcal{R}} u_1, \ldots t_n \to^*_{\mathcal{R}} u_n \Rightarrow f(t_1, \ldots, t_n) \to^*_{\mathcal{R}} f(u_1, \ldots, u_n)$$

but this is true by definition of first-order reduction.

However we still need to verify that this interpretation gives rise to a premodel for $curry(\mathcal{R})$. To do this we apply lemma 64, which shows that it suffices to prove for each rule $curry(l) \to curry(r) \in curry(\mathcal{R})$, typed by context Γ that

$$(\!|\Gamma \vdash curry(l) : D|\!) \geq (\!|\Gamma \vdash curry(r) : D|\!)$$

and that

$$(\!|\Gamma \vdash (\lambda x : T.t)\ u|\!) = (\!|\Gamma \vdash t\{x \mapsto u\}|\!)$$

This second equality can be shown in the same manner as for the Rea algebra (lemma 123).

To show the first inequality, consider such a rule, and the context Γ that sends every variable in $curry(l)$ (and $curry(r)$) to the type D. It is clear that this is the most general possible context. If $\Gamma = x_1 : D, ldots, x_n : D$ then we take elements $t_1, \ldots, t_n$ of $\mathcal{T}rm_{fo}$ and variables $y_1, \ldots, y_n \in \mathcal{Y}$ in l that verify $i(y_i) = x_i$. Let σ be the substitution that sends y_i to t_i. We assert:

$$(\!|\Gamma \vdash curry(l)|\!)(t_1, \ldots, t_n) = l\sigma$$

and

$$(\!|\Gamma \vdash curry(r)|\!)(t_1, \ldots, t_n) = r\sigma$$

We show this by induction on the structure of l:

- Variable case: $l = y_i$, $curry(l) = x_i$ and $(\!|\Gamma \vdash l|\!)(t_1, \ldots, t_n) = var(x_i)(t_1, \ldots, t_n) = t_i$.
- Function case: $l = f(u_1, \ldots, u_n)$ which gives

  $$(\!|\Gamma \vdash f curry(u_1) \ldots curry(u_n)|\!)(t_1, \ldots, t_n) = f((\!|\Gamma \vdash curry(u_1)|\!)(\vec{t}), \ldots, (\!|\Gamma \vdash curry(u_n)|\!)(\vec{t}))$$

  which by induction hypothesis gives

  $$f(u_1\sigma, \ldots, u_n\sigma) = f(u_1, \ldots, u_n)\sigma$$

Then we can conclude, as $l\sigma \geq_{\mathcal{R}} r\sigma$.

We can now prove our main theorem, which states that termination of the first-order rewrite system $\mathcal{R}$ implies that of $curry(\mathcal{R}) \cup \beta$ over well-typed terms. It is of interest to note that first-order semantic labelling has been used to prove termination of $curry(\mathcal{R})$ without β-reductions (or λ-abstractions) under the condition that $\mathcal{R}$ is terminating. This was shown using almost exactly the same labelling as here, in Zantema [Zan95].

Theorem 138 (Breazu-Tannen, Gallier & Okada)
If $\mathcal{R}$ is strongly normalizing over terms in $\mathcal{T}rm^{\Sigma}_{fo}$, then so is $curry(\mathcal{R}) \cup \beta$ over $\mathcal{T}rm^{\Sigma}$

Proof. We use the main theorem of semantic labelling (theorem 84) applied to the $\mathcal{F}o$ premodel described above, taking $ar(f)$ as the number of recursive arguments for f. We then need to prove termination of $\overline{curry(\mathcal{R}) \cup \beta}$ relative to $Struct$. The positivity conditions are trivially satisfied, and all rules match on constructor terms. We take $n_f = ar(f)$.

We define the following precedence on labelled function symbols: Let $\trianglerighteq$ be the order on $\mathcal{T}rm_{fo}$ defined by reflexive transitive closure of

$$\forall i, f(t_1, \ldots, t_n) \triangleright t_i$$

If f_l is a labelled function symbol, with $l = ((\!|\Gamma \vdash t_1|\!), \ldots, (\!|\Gamma \vdash t_n|\!))$, and g_m is a labelled function symbol with $m = ((\!|\Gamma' \vdash u_1|\!), \ldots, (\!|\Gamma' \vdash u_m|\!))$, then $f_l >_{call} g_m$ if $\Gamma = \Gamma'$ and for all $\vec{v} \in [\![\Gamma]\!]$

$$f((\!|\Gamma \vdash \vec{t}|\!)(\vec{v})) \to^+_{\mathcal{R}} \circ \trianglerighteq g((\!|\Gamma \vdash \vec{u}|\!)(\vec{v}))$$

We then define the $>_{prec}$ order on labeled function symbols by

$$>_{prec} = \geq_{inst} \circ \geq_{wk} \circ >_{call}$$

Notice that if $\mathcal{R}$ is strongly normalizing on first-order terms, then $>_{call}$ is well-founded.

Well-foundedness of $>_{prec}$ can then be shown in the same way as the corresponding lemmas for a system labelled with the realizability semantics (lemmas 130 and), using the fact that $>_{call}$ respects $\geq_{\mathcal{R}}$.

As we are working with a premodel that is *not* a model, compatibility of $>_{prec}$ with $\geq_{Struct}$ is a bit trickier: in addition to the inclusions of lemma 129, we need to show the inclusions:

$$\geq_{decr} \circ >_{call} \ \subseteq >_{call} \tag{c}$$
$$\geq_{decr} \circ \geq_{wk} \ \subseteq \geq_{wk} \circ \geq_{decr} \tag{d}$$
$$\geq_{decr} \circ >_{inst} \ \subseteq >_{inst} \circ \geq_{decr} \tag{e}$$

Which allow us to show the inclusion:

$$(\geq_{decr} \cup \geq_{inst} \cup \geq_{wk}) \circ (\geq_{inst} \circ \geq_{wk} \circ >_{call}) \ \subseteq \geq_{inst} \circ \geq_{wk} \circ >_{call}$$

The decrease condition is then rather straightforward to check: if $curry(l) \to curry(r)$ is in $curry(\mathcal{R})$ and typeable in the context Γ, then let ϕ be a $\mathcal{F}o_\Gamma(\Delta)$ valuation. Set $l = f(l_1, \ldots, l_n)$. We have $\overline{curry(l)}^{\phi} = f_a \overline{curry(\vec{l})}$ with $a = ((\!|\Gamma \vdash curry(l_1)|\!)_\phi, \ldots, (\!|\Gamma \vdash curry(l_n)|\!)_\phi)$. We proceed by induction on the structure of r:

- x a variable. There is nothing to show here.
- $g(u_1, \ldots, u_m)$ We have $\overline{curry(g(u_1, \ldots, u_m))} = g_b\ \overline{curry(\vec{u})}$ with

  $$b = ((\!|\Gamma \vdash curry(u_1)|\!)_\phi, \ldots, (\!|\Gamma \vdash curry(u_m)|\!)_\phi)$$

  For any occurrence of a function symbol in u_i, we may conclude by the induction hypothesis. If $\vec{v} \in [\![\Delta]\!]$, then we define σ as the substitution that takes y such that $i(y) = x \in \mathrm{dom}(\Gamma)$ and sends it to $\phi(x)(\vec{v})$, and we have for each i, j:

  $$(\!|\Gamma \vdash curry(l_i)|\!)_\phi(\vec{v}) = l_i\sigma \quad (\!|\Gamma \vdash curry(u_j)|\!)_\phi(\vec{v}) = u_j\sigma$$

  Considering the rule $f(l_1, \ldots, l_n) \to r$ we have $f(\vec{l\sigma}) \to_{\mathcal{R}} \circ \trianglerighteq g(\vec{u}\sigma)$ for any σ, from which we conclude

  $$f_a >_{call} f_b$$

We may therefore apply theorem 124 and conclude that the labelled system is normalizing relative to $Struct$.

■

6

Related work

There has been much work on termination of higher-order rewriting. A first remark is that there is not one but many different descriptions of higher-order rewrite systems. Let us simply mention *higher-order rewriting systems* by Nipkow [Nip91], and *combinatory reduction systems* from Klop (see *e.g.* [KvOvR93]). See Terese [BKdV03] or van Raamsdonk [vR96] for an overview.

We mostly confine ourselves here to rewriting which involves bound variables. We do not attempt to give a comprehensive history of the field here, but simply try to give a description of the work closest to our results.

Jouannaud and Okada [JO91], describe a combination of left-algebraic rewriting and β-reduction, and give a termination criterion for such a combination, the *General Schema*. Drawing on work by Coquand [Coq92], this criterion was extended to an extension of the Calculus of Constructions with rewriting by Fernàndez [Fer93] and further by Blanqui *et al* [BJO97, BJO02, Bla01, Bla03] in a system called the Calculus of Algebraic Constructions. These criteria are syntactic in nature, they capture structural decrease in recursive calls, and extended to allow instantiation of higher-order variables when destructing a constructor of a higher-order data type.

The approach by rewriting is contrasted to the approach by *eliminators* which are a restricted set of function definitions which allow definition of recursive functions simply by their clever use. This simplifies termination considerations, as it is only necessary to consider the restricted set of rules. Mendler [Men87, Men91], followed by Mattes [Mat98], describes such eliminators and gives conditions under which well-typed terms in a polymorphic type system may normalize under the combination of β-reduction and the rules for these eliminators. The Calculus of Inductive Constructions, for instance, is an extension of the Calculus of Constructions with eliminators, and its correctness was shown by Paulin and Werner [PM96, Wer94]. Using eliminators to define functions can be unwieldly however, and Giménez [Gim95] describes a translation from recursive definitions satisfying a certain *guardedness condition* [Gim98] to the system with eliminators. In practice it is a variant the guardedness condition that is implemented in type systems such as the Coq interactive theorem prover [Coq08], Agda [Nor09] and Epigram [McB04] (see Amadio *et al* [ACG98] and Abel [Abe99]). The version of the CIC with fixpoint recursion under the guardedness condition system is quite close to the CAC in its description and expressivity.

The drawbacks of the syntactic approach led to the development of *type based termination*, introduced independently by Gimenez [Gim98] and Hughes, Pareto & Sabry [HPS96] for simple types and extended to polymorphism by Frade [Fra03] and to higher kinded polymorphism and more flexible inductive types by Abel [Abe06]. The present thesis attempts to give a semantic account of this technique. The type-based termination criterion given in chapter 1 is taken from

Blanqui & Roux [BR09], and is a simply-typed account of the version given by Blanqui [Bla04] for the CAC.

Van de Pol shows [vdP96, vdP93, vdPS95] that it is possible to define certain interpretations of the simply typed λ-calculus with rewrite rules into well-founded domains. These interpretations verify the following remarkable property: every rewrite step, including β-reduction, induces a strict decrease in the interpretation. Obviously this allows one to prove strong normalization. We conjecture that these interpretations are premodels in the sense of definition 58, but no labelling is required to show normalization of the rewrite system. However in practice it is quite difficult to build interpretations that verify van de Pol's conditions, it is much easier to build premodels that *satisfy β-reduction*: for all terms t, u the interpretation of $\lambda x.t\ u$ is *equal* to that of $t\{x \mapsto u\}$. The labelling approach allows separation of concerns: we build a premodel in which β-reduction is satisfied, and (relative) termination is given (on the labelled system) by our criterion (theorem 124)

If we do not restrict ourselves to left-algebraic rewriting then it can be shown that such termination models are not complete: there exist terminating rewrite systems which do not admit such a model, as shown by Kahrs [Kah95, vdP96] (see also Terese [BKdV03]). Hamana however shows [Ham07] that given a certain restriction on meta-terms appearing in the rules, which he calls *solid meta-terms*, the criterion becomes complete. Van Oostrom *et al* [AvOS10] note that many modularity results generalized from first-order rewriting become false in the higher-order case.

Our work is evidently closely related to the work of Hamana [Ham07] which builds on previous work [Ham03, Ham05] on algebraic semantics of higher-order rewriting, in which he describes a higher-order version of the semantic labelling technique and gives some applications. As previously noted, the system he describes does not preserve β-reduction and as such does not generally allow one to directly apply known termination criteria like the General Schema or the Higher Order Recursive Path Ordering. Our solution involves the use of structural rules, which turn the termination problem into a relative termination problem.

The work the most (quantitatively) related to ours is possibly that of Whalstedt [Wah07], in which he gives a general termination criterion on terms with a dependent type system and recursive definitions: if the call-relation is well-founded then well-typed terms are terminating under β-reduction and the recursive definitions. This is equivalent to the termination of the system if each recursive call is made on smaller arguments that is stable by substitution and reduction. This can be seen as the syntactic pendent to our approach: given such a well-founded order, we can construct a "syntactic model" and apply our criterion to the system labelled by this model using the well-founded order as precedence. However our approach presents some quantitative differences. Whalstedts criterion applies to a language with dependent types and type constructors with parameters, whereas we treat only rewriting within simple types. However he only treats rewriting on datatypes without higher order recursive parameters, and without matching on defined symbols. More investigation is needed to discover the relationship between these two criteria. Let us finally note that, for reasons similar to ours, Whalstedts criterion does not allow one to prove termination of example 9, as the information needed for decrease is not present in the recursive call. In a sense, all bound variables in the arguments of recursive calls are considered to be instantiable to *any* possible term. This difficulty is a fundamental aspect of higher-order termination, that we attempt to treat with our type-based method in the second part of this thesis.

Part II

A Type-Based Dependency Analysis

1

Dependency Pairs

The *dependency pair approach*, introduced by Arts and Giesel [AG00], is a framework that permits a powerful termination analysis. The fundamental idea is that it suffices to examine possible sequences of *calls*, and show that there can not be an infinite chain of calls. In this part we begin by describing the theory of so called *dynamic dependency pairs* for *first-order* systems as described in definition 135, with the intent of introducing notation and background. In particular we describe the notion of *dependency pair processor*, which embodies the fundamental philosophy of the dependency pair approach, and explain the concept of *approximated dependency graph*, *reduction pairs* and *simple projections*. The material for this chapter is taken from Giesl *et al* [GTSK05] and Hirokawa & Middeldorp [HM07a].

In the next chapter we show that it is possible to give a *type-based* approach to dependency pairs, using *refinement types* inspired by Freeman & Pfenning [FP91a], on left-algebraic systems with β-reduction. In this approach, dependency pairs are given by type information on the rules instead of pairs of terms. We show that a natural type-based dependency graph approximation can be built using these dependency pairs, and that it is possible to formulate a termination criterion based on *simple projections*. The chapter after that shows that the criterion is sound, that is that well-typed terms are strongly normalizing if the criterion is satisfied.

Size-based termination involves the use of dependent types to infer an approximation of the size of terms of inductive type. However in general the size abstraction is not sufficient to determine whether a rewrite system is strongly normalizing. Let us consider the following rewrite system:

Example 14 Let $\Sigma = \{f, g, c, d\}$ be a signature. Consider the following rewrite system $\mathcal{R}$:

$$\begin{array}{rcl} f\ (c\ x) & \to & c\ (g\ (d\ x)) \\ g\ (c\ y) & \to & f\ (c\ y) \end{array}$$

The system $\mathcal{R}$ is strongly normalizing (even on untyped terms). However this fact can not be established using the size-based criterion presented in chapter 1, simply because there is no size decrease in either rule. Termination follows from the fact that if a term t rewrites to t' by application of the second rule, then only the first rule may be applied, and then no further rules can be applied at that position, as it can not possibly match either rule. To make this reasoning precise need a way of systematically analyzing the possible sequence of calls in between rules.

1.1 The Dependency Chain Criterion

We recall the definition for first order terms.

Definition 139 Let Σ be a signature. An *arity* ar is a function from Σ to the set of natural numbers. The set of *first-order terms* for the signature Σ with arity ar is defined as the set

$$t_1, \ldots, t_n \in \mathcal{T}rm_{fo} := x \mid f(t_1, \ldots, t_n)$$

with $x \in \mathcal{X}$, $f \in \Sigma$ and $n = ar(f)$.

A first-order rewrite system $\mathcal{R}$ is a set of pairs $l \to r$ such that $l, r \in \mathcal{T}rm_{fo}$ and the set of variables in r is contained in that of l. We define head rewriting $\to^h_{\mathcal{R}}$, $\to_{\mathcal{R}}$, $\to^*_{\mathcal{R}}$ and $\to^+_{\mathcal{R}}$ in the same way as for definition 12.

We say that a symbol $f \in \Sigma$ is *defined*, if there is a rule $f\ l_1 \ldots l_n \to r \in \mathcal{R}$, we write $\mathcal{D}$ for the set of all defined symbols, if $c \notin \mathcal{D}$ we say that c is *undefined* (we do not use the common term *constructor* as it is used with another meaning in chapter 1). The *head* $hd(t)$ if a term $t \in \mathcal{T}rm_{fo}$ is the (unique) symbol f such that $t = f(t_1, \ldots, t_n)$ and is undefined if $t = x$ is a variable.

We define the *subterm ordering* $\trianglerighteq$ by the reflexive transitive closure of the relation defined by

$$\forall f \in \Sigma, 1 \leq i \leq n, f(t_1, \ldots, t_n) \trianglerighteq t_i$$

if $ar(f) = n$.

We build the set of dependency pairs, which intuitively denotes the set of possible consecutive *calls*, where a call is a subterm of a term on which it is possible to apply a rule. The fundamental insight of the dependency pair approach is the following observation: if there is a non-terminating term, then there is a *minimally non-terminating term*, such that every strict subterm is terminating. For such a term and a non-terminating rewrite sequence there is obviously at least one head rewrite step. Once this step is performed, there is some subterm of that term which is once again a minimally non-terminating term. This suggests that to prove termination of a rewrite system $\mathcal{R}$ it is interesting to concentrate on the combination of $\to_{\mathcal{R}} \circ \trianglerighteq$.

Definition 140 To each $f \in \Sigma$ we associate a new symbol $f^\sharp$, and write $\Sigma^\sharp$ for $\{f^\sharp \mid f \in \Sigma\}$, we extend an arity for Σ to an arity for $\Sigma^\sharp$ by taking $ar(f^\sharp) = ar(f)$. Given a term t in $\mathcal{T}rm_{fo}$, we define $t^\sharp$ as $f^\sharp(t_1, \ldots, t_n)$ if $t = f(t_1, \ldots, t_n)$ and as x if $t = x$.

The set $DP_{\mathcal{R}}$ of *dependency pairs* is defined as

$$DP_{\mathcal{R}} := \left\{ l^\sharp \to t^\sharp \mid l \to r \in \mathcal{R}, r \trianglerighteq t \wedge hd(t) \in \mathcal{D} \right\}$$

Thus, $DP_{\mathcal{R}}$ is a rewrite system over the signature $\Sigma \cup \Sigma^\sharp$.

Given a set of rewrite rules $\mathcal{D}$ over the signature $\Sigma \cup \Sigma^\sharp$, a $\mathcal{D}$-chain is defined by a possibly infinite sequence of rules $l_1 \to r_1, l_2 \to r_2, \ldots \in \mathcal{R}'$ such that there exists a substitution σ (of possibly infinite domain), such that

$$\forall i, r_i\sigma \to^*_{\mathcal{R}} l_{i+1}\sigma$$

We say that the *problem $\mathcal{D}$ is finite* if there are no infinite $\mathcal{D}$-chains.

Then the main theorem states:

Theorem 141 (Arts & Giesl [AG00])
Given an algebraic rewrite system $\mathcal{R}$ with a set of dependency pairs $DP_{\mathcal{R}}$, we have

$$\mathcal{SN}_{\mathcal{R}} \Leftrightarrow \text{the problem } DP_{\mathcal{R}} \text{ is finite}$$

Notice that if there are no infinite $\mathcal{D}$-chains, then in fact $\mathcal{D}$ is terminating relative to $\mathcal{R}$ in the sense of chapter 3 (definition 82), as has been noted by Zantema [Zan04a]. It is a notable fact that many practical approaches to termination consist in transforming a termination problem into a relative termination problem.

There is an analogy between the finite chain condition and the reducibility method used in chapter 5 to prove strong normalization of the termination criterion of theorem 124. Though the reducibility method is not a termination criterion *per say*, it is a proof technique which proceeds in two steps:

- If the context $C[]$ is "safe" or computable, and so is t then so is $C[t]$.
- Every function symbol f is computable: if $t_1, \ldots, t_n$ are computable terms, then so is $f(t_1, \ldots, t_n)$. This can be proven by considering all possible calls: if $f(\vec{t}) \to_{\mathcal{R}} r$ and $g(u_1, \ldots, u_m)$ is a call of r, then $g(\vec{u})$ is computable.

It is clear that the second step corresponds to the dependency pair approach, and it is in fact possible to give a proof of the main theorem using computability :

Proof. The $\Rightarrow$ direction is easy by contraposition: if there is an infinite $DP_{\mathcal{R}}$-chain, it is easy to built an infinite $\mathcal{R}$ reduction sequence.

For the $\Leftarrow$ direction: We say $t \in \mathcal{T}rm_{fo}$ is *computable* if it is in $\mathcal{SN}_{\mathcal{R}}$. We say that $f \in \Sigma$ is computable if $ar(f) = n$ and for every computable $t_1, \ldots, t_n \in \mathcal{T}rm_{fo}$, $f(t_1, \ldots, t_n)$ is computable. It is sufficient to show that every symbol f is computable to show termination: if every symbol is computable and $t \in \mathcal{T}rm_{fo}$ then by induction on t either

- $t = x$, then $t \in \mathcal{SN}_{\mathcal{R}}$
- $t = f(t_1, \ldots, t_n)$ and so by induction hypothesis each t_i is strongly normalizing, and as f is computable, so is t

We define $\mathcal{T}rm^*_{fo}$ to be the set of finite tuples of terms and $>_{dp}$ to be the order on $\Sigma \times \mathcal{T}rm^*_{fo}$ defined by the transitive closure of

$$(f, \vec{t}) >_{dp} (g, \vec{u}) \Leftrightarrow f^{\sharp}(t_1, \ldots, t_n) \to^*_{\mathcal{R}} \circ \to_{DP} g^{\sharp}(u_1, \ldots, u_m)$$

By hypothesis, $DP_{\mathcal{R}}$-chains are finite, and so $>_{dp}$ is well-founded.

To show computability of symbols, we proceed by case:

- f is undefined. Then the computability follows by definition of reduction.
- f is defined. Suppose $ar(f) = n$ and take $t_1, \ldots, t_n$ arbitrary computable terms and set $t = f(t_1, \ldots, t_n)$. We assert: it suffices to show that for every $t'_1, \ldots, t'_n$ such that for each i, $t_i \to^*_{\mathcal{R}} t'_i$, and every *head reduct* r of $f(t'_1, \ldots, t'_n)$, r is computable.

Indeed a term is normalizable if every one of its reducts is normalizable, and we can reason by well founded induction on $\vec{t}$ ordered by the product order $\to_{\mathcal{R}}^{+} \times \ldots \times \to_{\mathcal{R}}^{+}$ (well-founded on reducts of $\vec{t}$): a one step reduct of t is either a head reduct, or a reduct of the form $f(t_1, \ldots, t_i', \ldots, t_n)$, with $t_i \to_{\mathcal{R}} t_i'$, for which the assertion is true by induction hypothesis.

To show the assertion, we proceed by well-founded induction on $(f, \vec{t})$ ordered by $>_{dp}$. Suppose $t \to_{\mathcal{R}}^{*} f(t_1', \ldots, t_n') \to_{\mathcal{R}}^{h} r$. We show that r is in $\mathcal{SN}_{\mathcal{R}}$ by induction on the term structure:

- $r = x$ a variable: r is in normal form by definition.
- $r = g(u_1, \ldots, u_m)$. By the induction hypothesis, u_i are in $\mathcal{SN}$. But we also have $(f, \vec{t}) >_{dp} (g, \vec{u})$, and so $g(u_1, \ldots, u_m)$ is computable.

■

This approach to dependency pairs by computability is quite useful in the case of higher-order rewriting as noted by Kusakari & Sakai [KS07], and Blanqui *et al* [Bla06, KISB09]. The proof technique exposed above is very similar to that used to prove correctness of our type-based dependency pair criterion.

1.2 Dependency Pair Processors and the Dependency Graph

Working with the set of dependency pairs can be nice, as it offers a different rewrite system on which to try our standard termination techniques. It is a transformation technique somewhat like semantic labelling presented in chapter 3. However, unlike semantic labelling, little semantic information has been added, and in general there are as many or more rules in $DP_{\mathcal{R}}$ as in $\mathcal{R}$. This has led to the elaboration of *dependency pair processors*. Processors attempt to take a DP problem and break it into a number of smaller problems, for which finiteness would imply finiteness of the original problem. We present one of the most important DP processors, the *approximated dependency graph*.

Definition 142 Let $\mathcal{D}$ be a dependency pair problem for the rewrite system $\mathcal{R}$. The *dependency graph* DG for $\mathcal{D}$ is the (directed) graph with elements of $\mathcal{D}$ as nodes and an edge between $l_1 \to r_1$ and $l_2 \to r_2$ if there is a σ such that $r_1\sigma \to_{\mathcal{R}}^{*} l_2\sigma$ (that is if there is a chain between the pairs $l_1 \to r_1$ and $l_2 \to r_2$).

A graph $\mathcal{G}$ *approximates* DG if $DG \subseteq \mathcal{G}$, that is if the set of nodes of $\mathcal{G}$ is equal to that of DG and for each pair of nodes n_1 and n_1, if there is an edge between n_1 and n_2 in DG, then there is one in $\mathcal{G}$. A graph is *strongly connected* if there is a path from every node to every other node. A *strongly connected component* or *SCC* of $\mathcal{G}$ is a maximal strongly connected subgraph of $\mathcal{G}$.

The need for approximated dependency graphs stems from the undecidability to compute the dependency graph. From here on we only consider finite rewrite systems.

Failure of finiteness in a DP problem can be traced to cycles in a dependency graph, if this approximation is finite.

Lemma 143 For every infinite chain for a DP problem $\mathcal{D}$ and every graph $\mathcal{G}$ that is an approximation of the dependency graph, there is a strongly connected component C and an infinite path in C.

Proof. Suppose that there is an infinite $\mathcal{D}$-chain $(n_i)_{i\in\mathbb{N}}$. The chain corresponds to a path in $\mathcal{G}$, as each sequence n_i, n_{i+1} is in particular a 2-element chain, therefore a link in the dependency graph, which is approximated by $\mathcal{G}$. Observe that the graph $\mathcal{G}$ is finite, by finiteness of the set of dependency pairs, which itself follows from finiteness of $\mathcal{R}$. Now observe that there is some $N \in \mathbb{N}$ and some SCC C such that

$$\forall i \geq N, n_i \in C$$

Indeed, suppose that there are C and C' such that for all N, there exist $i_1, i_2, i_3 \geq N$ such that n_{i_1} and n_{i_3} are in C and n_{i_2} is in C'. By strong connectedness of C, this gives a path between n_{i_1} and n_{i_2} and between n_{i_2} and n_{i_1}, which contradicts maximality of SCCs.

As all SCCs are finite, there is an infinite path in C.

■

We can therefore describe a first DP processor:

Theorem 144 (Giesl, Thiemann & Shneider-Kamp [GTSK04])

Let $\mathcal{D}$ be a problem. If $\mathcal{G}$ is an approximation of the dependency graph, with SCCs $C_1, \ldots, C_n$, then

$$C_1 \text{ finite } \wedge \ldots \wedge C_n \text{ finite } \Rightarrow \mathcal{D} \text{ finite}$$

In particular, if $\mathcal{G}$ is without cycles, then $\mathcal{D}$ is finite.

To use this result, we need a simple and efficient way to compute a decent approximation to the dependency graph. One such approximation can be computed in the following manner: we take a right hand side, and consider that any term headed by a defined symbol or a variable can rewrite to any term. This is formalized by replacing these terms by fresh variables and performing unification.

Definition 145 (Standard approximated graph)

Let Σ be a signature and $\mathcal{R}$ be a rewrite system. If t is an algebraic term *on the signature* $\Sigma \cup \Sigma^\sharp$, we define $rencap(t)$ by induction:

- $rencap(f(t_1, \ldots, t_n)) = y$ with y fresh, if $f \in \mathcal{D}$.
- $rencap(g(t_1, \ldots, t_n)) = g(rencap(t_1), \ldots, rencap(t_n))$ if $g \notin \mathcal{D}$ or $g \in \Sigma^\sharp$.

The *standard approximation* to the dependency graph $DG_{\mathcal{R}}$ is the graph $\mathcal{G}_{\mathcal{R}}$ with

- As set of nodes $DP_{\mathcal{R}}$,
- An edge between $l_1 \to r_1$ and $l_2 \to r_2$ iff $rencap(r_1)$ and l_2 are unifyable.

Property 146 (Arts & Giesl [AG00])

The graph $\mathcal{G}_{\mathcal{R}}$ is an approximation of $DG_{\mathcal{R}}$.

Proof. Suppose that there is a link between $l_1 \to r_1$ and $l_2 \to r_2$ in $DG_{\mathcal{R}}$, that is that there is some substitution σ such that

$$r_1\sigma \to^*_{\mathcal{R}} l_2\sigma$$

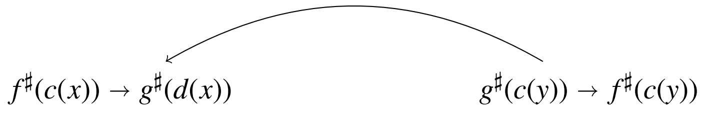
Let us show that there is a link in $\mathcal{G}_{\mathcal{R}}$: we show that there is a unifier θ of r_1 and l_2, by induction on r_1.

- $r_1 = x$: there is a trivial unifier.

- $r_1 = f(t_1, \dots, t_n)$. In this case either $f \in \mathcal{D}$ and so $rencap(r)$ is equal to a fresh variable, and we can easily find a unifier. Otherwise, $f \in \Sigma^\sharp$ or $f \in \Sigma \backslash \mathcal{D}$. In this case, as there are no rules in $\mathcal{R}$ with head equal to f, we have $l_2 = f(u_1, \dots, u_n)$, with $t_i \to^*_{\mathcal{R}} u_i$. By induction hypothesis, there are $\theta_1, \dots, \theta_n$ such that $rencap(t_i)\theta = u_i\theta_i$. Furthermore, by freshness of the variables chosen in definition of $rencap$, all variables in $rencap(t_i)$ are distinct from those in $rencap(t_j)$ if $i \neq j$. We can therefore take $\theta = \theta_1 \uplus \dots \uplus \theta_n$.

■

Notice that this allows us to prove strong normalization of (the first order analogue of) the rewrite system in example 14: the defined symbols are f and g, and the standard approximated dependency graph is:


$f^\sharp(c(x)) \to g^\sharp(d(x))$
$g^\sharp(c(y)) \to f^\sharp(c(y))$

This graph is without cycles, so by application of proposition 146 and theorem 144 and of theorem 141, we may conclude that the system is strongly normalizing.

We give another very important type of processor, which is often used in the final proof that a problem is finite. The general idea is to find a certain well founded order $\gtrsim$ on terms that is compatible with $\to^*_{\mathcal{R}}$, that is if $t \to^*_{\mathcal{R}} u$ then $t \gtrsim u$, and is also compatible with $\mathcal{D}$, and then we remove all dependency pairs which entail a strict decrease, as they may only occur a finite number of times in any infinite chain. This is difficult in practice, so we weaken the requirement: it suffices to find a *pair* of orders, called a *reduction pair* [KNT99], $\gtrsim$ and *succ*, which verify a certain compatibility property.

Definition 147 A relation R over $\mathcal{T}rm_{fo}$ is *monotonic* if for each $f \in \Sigma$, $t_1Rt'_1, \dots, t_nRt'_n$ implies $f(t_1, \dots, t_n)Rf(t'_1, \dots, t'_n)$. A relation is *stable by substitution* or simply *stable* if for each tRu and every substitution θ, $t\theta Ru\theta$. It is trivial to show that the reduction relation $\to^*_{\mathcal{R}}$ is both monotonic and stable by reduction.

Given Σ a signature and $\mathcal{R}$ a rewrite system over Σ, a *reduction pair* $(\gtrsim, \succ)$ for $\mathcal{R}$ is a pair of relations on terms over the signature $\Sigma \cup \Sigma^\sharp$ such that

- $\gtrsim$ is a monotonic and stable order.

- $\succ$ is stable and well-founded.

- $\succ$ is *compatible* with $\gtrsim$: $\gtrsim \circ \succ \subseteq \succ$ and $\succ \circ \gtrsim \subseteq \succ$.

We write $DP_{\gtrsim}$ for

$$\{l \to r \in \mathcal{D} \mid l \gtrsim r\}$$

$DP_{\succ}$ for

$$\{l \to r \in \mathcal{D} \mid l \succ r\}$$

and $\mathcal{R}_{\gtrsim}$ for

$$\{l \to r \in \mathcal{R} \mid l \gtrsim r\}$$

The processor based on reduction pairs allows us to remove strictly ordered (by $\succ$) dependency pairs from the set of dependency pairs, but only every other pair *and every rule* is weakly ordered (by $\gtrsim$).

Theorem 148 (Giesl *et al* [GTSK04])
Suppose that Σ is a signature, $\mathcal{R}$ is a rewrite system and $(\gtrsim, \succ)$ is a reduction pair for $\mathcal{R}$. If $\mathcal{R}_{\gtrsim} = \mathcal{R}$ and $DP_{\gtrsim} \cup DP_{\succ} = \mathcal{D}$ then we have

$$\mathcal{D} \text{ is finite } \Leftrightarrow DP_{\gtrsim} \backslash DP_{\succ} \text{ is finite}$$

We now have a powerful framework that allows us to prove termination of first order rewrite systems: build the approximated dependency graph, find a reduction pair, and verify that the graph $\mathcal{G}_{\mathcal{R}}$, from which we remove the nodes which admit strict decrease in $\rhd$ is without cycles.

Let us give a simpler method that allows us to remove strongly connected components from the approximated dependency graph, the *simple projection* method taken from Hirokawa and Middeldorp [HM07b]. The advantage of simple projections is that they remove the important constraint of having to verify compatibility with respect to the rules of the rewrite system $\mathcal{R}$, as is the case with the reduction pair processor.

Definition 149 (Simple projection)
Given a first-order rewrite system $\mathcal{R}$ over Σ, a *simple projection* ι is a function from $\mathcal{S}^{\sharp}$ to $\mathbb{N}$, such that $1 \le \iota(f) \le ar(f)$. We also write ι for the function that maps $f^{\sharp}(t_1, \ldots, t_n)$ to $t_{\iota(f^{\sharp})}$.

Theorem 150 (Hirokawa & Middeldorp [HM07b])
Suppose that $\mathcal{G}$ is an approximated dependency graph with SCC C, and ι is a simple projection. Suppose that for each node $l \to r$ in C, $\iota(l) \trianglerighteq \iota(r)$ and that for each cycle in C, there is at least one node $l \to r$ such that $\iota(l) \rhd \iota(r)$. Then

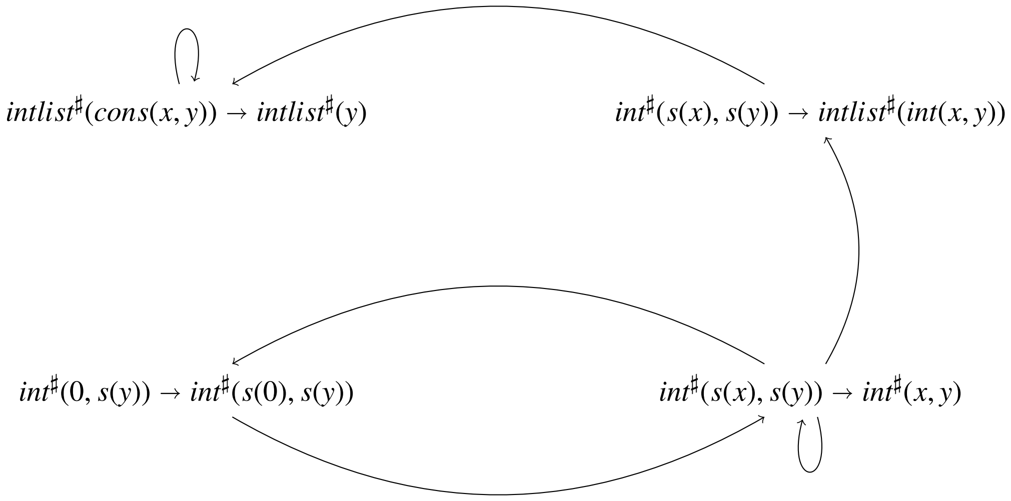
$$\mathcal{G} \backslash C \text{ is finite} \Rightarrow \mathcal{D} \text{ is finite}$$

It is of course not possible in general to have equivalence, as this is a syntactic criterion that is decidable (if $\mathcal{G}$ is finite) and therefore can not be complete. Let us give an example of the application of this method, taken again from Hirokawa & Middeldorp [HM07b].

Example 15 We consider the signature $\Sigma = \{0, S, int, intlist, nil, cons\}$ with $ar(0) = ar(nil) = 0$, $ar(intlist) = ar(S) = 1$ and $ar(int) = ar(cons) = 2$ and the following rules:

$$\begin{array}{lll}
intlist(nil) & \to & nil \\
intlist(cons(x, y)) & \to & cons(S(x), intlist(y)) \\
int(0, 0) & \to & cons(0, nil) \\
int(0, s(y)) & \to & cons(0, int(s(0), s(y))) \\
int(s(x), 0) & \to & nil \\
int(s(x), s(y)) & \to & intlist(int(x, y))
\end{array}$$

This gives the following standard approximated dependency graph:


$intlist^\sharp(cons(x, y)) \to intlist^\sharp(y)$
$int^\sharp(s(x), s(y)) \to intlist^\sharp(int(x, y))$
$int^\sharp(0, s(y)) \to int^\sharp(s(0), s(y))$
$int^\sharp(s(x), s(y)) \to int^\sharp(x, y)$

We can apply the processor described above to show termination: the SCCs are given by the full subgraphs with nodes

$$\left\{intlist^\sharp(cons(x, y)) \to intlist^\sharp(y)\right\}$$

and

$$\left\{int^\sharp(0, s(y)) \to int^\sharp(s(0), s(y)), int^\sharp(s(x), s(y)) \to int^\sharp(x, y)\right\}$$

respectively.

Taking as simple projection $\iota(intlist) = 1$ and $\iota(int) = 2$ we can easily check that for every SCC, and every dependency pair $l \to r$ in those SCCs,

$$\iota(l) \trianglerighteq \iota(r)$$

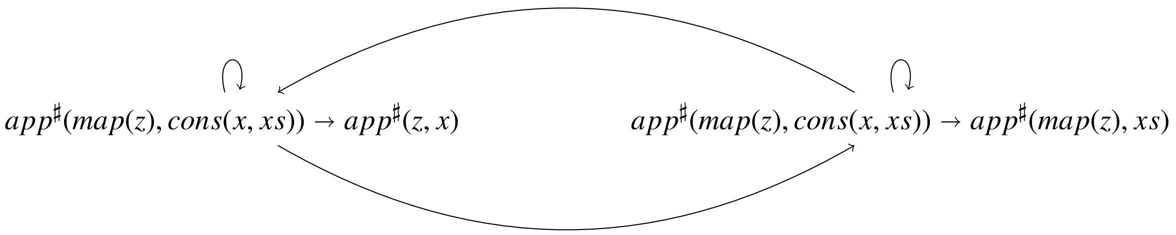
And there is a strict decrease in each cycle.

Let us make an important observation: it is possible to simulate a system with higher-order rewrite rules using first-order rewriting. The standard way to do this is to transform the left algebraic rewrite system into a first-order one, using the *defunctionalization* transformation [DN01] also called *lambda lifting* [Joh85]. By applying λ-lifting, we can give a first-order account of higher-order functions by considering systems with a special *application* function.

Example 16 We give $\Sigma = \{app, map, cons, nil\}$ as a signature with arities $ar(ap) = ar(cons) = 2$ and $ar(map) = ar(nil) = 0$. We give the rules

$$\begin{array}{lcl} app(map(z), nil) & \rightarrow & nil \\ app(map(z), cons(x, xs)) & \rightarrow & cons(app(z, x), app(map(z), xs)) \end{array}$$

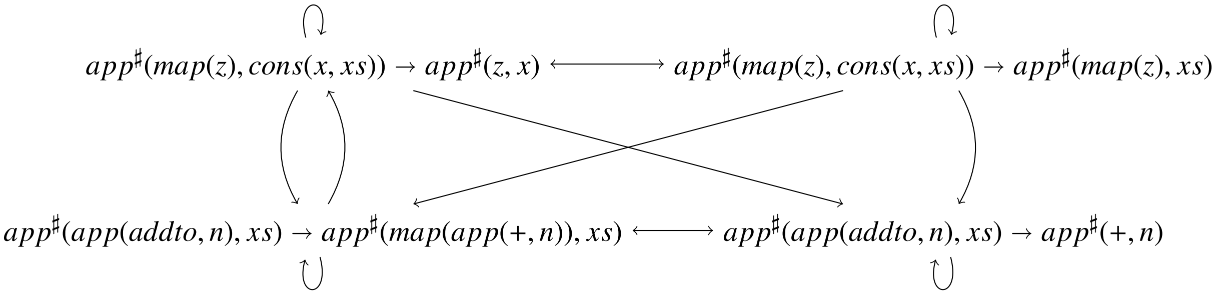
This leads to the approximated dependency graph:


$app^\sharp(map(z), cons(x, xs)) \rightarrow app^\sharp(z, x)$
$app^\sharp(map(z), cons(x, xs)) \rightarrow app^\sharp(map(z), xs)$

Which can easily be treated with the above methods. However adding the terms + and *addto* of arity 0 and following rewrite rule:

$$app(app(addto, n), xs) \rightarrow app(map(app(+, n)), xs)$$

significantly complicates the dependency problem, as the standard approximated graph contains a single SCC consisting of every possible dependency pair:


$app^\sharp(map(z), cons(x, xs)) \rightarrow app^\sharp(z, x)$
$app^\sharp(map(z), cons(x, xs)) \rightarrow app^\sharp(map(z), xs)$
$app^\sharp(app(addto, n), xs) \rightarrow app^\sharp(map(app(+, n)), xs)$
$app^\sharp(app(addto, n), xs) \rightarrow app^\sharp(+, n)$

This problem can not be treated using the simple projection method given above. Unfortunately this type of rewrite rule may naturally appear in the defunctionalized version of a higher-order rewrite system.

The dependency pair approach benefits from numerous refinements to approximations of the dependency graph, construction of complex reduction pairs, and different dependency processors. This approach leads to successful automation of the proof of termination of a large number of systems, using for example the `AProVE` software system [GSkT06] or the *TTT* system [HM07b].

However this approach suffers from several drawbacks when addressing higher-order rewrite systems. The first is that it is difficult to adapt the method to work with abstractions: for instance, the subterm of an abstracted term may have free variables not contained in the set of free variables of the left-hand side, making the definition itself of dependency pairs non-trivial.

If defunctionalization is applied, then the use of *app* as the only "real" head symbol makes the dependency graph approximation very poor in general, as illustrated by the above example.

Another drawback is the absence of types. As seen in previous chapters, types often allows us to control much of the pathological behavior of higher-order rewriting. In particular, we would like to be able to remove the dependency pairs for which the left-hand side has the form $app(x, t)$ as in the above example. In a typed setting, the intuition is that x may only be instantiated with "computable" functions, and cannot therefore lead to non-termination. A related issue is that it is difficult for these methods to deal with rewrite systems involving elements of higher-order inductive types, like the *Ord* type of example 3. It is difficult to give a higher order version of reduction orders, as is demonstrated by the relative complexity of the *higher order recursive path order*.

These drawbacks, as well as the similarity between the dependency pair approach and the reducibility method, give motivation for attempting a *type based* dependency pair approach. This is somewhat a reversal of the previous chapters: in chapter 3, we apply a technique developed in the rewriting community to deepen our understanding of a type-theoretical construction. Here we give a type-theoretical exposition of a technique originally exclusive to the rewriting community. This is not surprising, as both communities are quite related and often prove very similar results. In this case, we believe that a type theoretic approach to higher-order dependency pairs could find applications in the termination analysis of pure functional languages, and could be very easily integrated into dependently typed languages, as it is the case with termination analysis based on size-types.

2

The Type System and the Termination Criterion

In this chapter we will only consider rewrite systems where only undefined symbols may be matched upon. We proceed as follows. Given a signature and left-algebraic rewrite system, we define a class of dependent types, dubbed *refinement types*, because of their similarity with the type system presented in Freeman and Pfenning [FP91a]. The motivation is, as always, to express an abstraction of the structure of the normal forms of programs within the types. As we are working with rewrite systems that have no orthogonality or even a confluence requirement, a term may have many normal forms. The abstraction we consider describe the shape of the normal forms by analogy with pattern matching of rules. For instance if the (unique) normal form of a term t is $\mathsf{Node}\ \mathsf{Leaf}\ \mathsf{Leaf}$, then t *matches* the pattern $\mathsf{node}(\top, \top)$, and so t will be in the semantics of $\boldsymbol{B}(\mathsf{node}(\top, \top))$, with $\boldsymbol{B}$ the type of binary unlabeled trees.

These types will have a similar function as the term abstraction computed by the *rencap* function defined in chapter 1. Indeed, we may compute a dependency graph using just the type information gathered from the rewrite system. Whereas the size based criterion required strict decrease for each recursive call, with the type based dependency graph it is possible to perform a more sophisticated analysis. As in the first-order dependency pair approach, we can define an embedding order on type level terms and thus define a *simple projection* on the dependency pairs very similar to the analogous notion for dependency pairs in the first order case given in the previous chapter (definition 149). We use this order to give a termination criterion very close to that described in theorem 150, but for higher-order terms with β-reduction.

2.1 The Type System

For simplicity, in this chapter we consider only the datatype of binary unlabeled trees. The general case of first-order inductive datatypes can be constructed in the same manner, however the extension to higher-order datatypes like $\boldsymbol{Ord}$ (see example 3) is left open. We first define the set of *patterns* which will be used in the type annotations to denote values toward which terms may reduce.

Definition 151 We define the set of *type-patterns*, or just *patterns* $\mathcal{P}$:

$$p, q \in \mathcal{P} \coloneqq \alpha \mid \mathsf{leaf} \mid \mathsf{node}(p, q) \mid \top \mid \bot$$

with $\alpha \in \mathcal{V}$ a set of *pattern variables.*

The patterns can label the base type of *binary trees* $\boldsymbol{B}$. The set of simple types is the same as in preceding chapters, except we allow *explicit* quantification over pattern variables. The $\top$ pattern or *wildcard* matches every possible value. It may be used in much the same way as ∞ is used in chapter 1 to allow more flexibility in typing. The $\bot$ pattern does not match any value. It allows us to type only neutral terms.

Definition 152 We define the set of types $\mathcal{T}$:

$$T, U \in \mathcal{T} := \boldsymbol{B}(p) \mid T \to U \mid \forall \alpha . T$$

Explicit quantification is a departure from the presentation of the size based criterion presented in chapter 1, in which type variables were implicitly universally quantified. Whereas the implicit presentation is more convenient for the model construction of chapter 4, the explicit presentation is more convenient in implementations, as it allows a simple, *syntax directed* type inferrence.

The explicit version with quantification at any position in types is in this case slightly more expressive than the implicit version, as we can express for example the type

$$(\forall \alpha . \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)) \to \boldsymbol{B}(\text{leaf})$$

Which is inexpressible in the implicit version. The difference is analogous to ML polymorphism versus system F polymorphism, see for example Milner [Mil78]. As in this case, the explicit pattern abstractions and applications are sufficient to allow decidability of type inferrence in our system. Note that type inferrence is likely to be undecidable in a system with implicit quantification of types which may appear at any position in a type, as is the case for system F as proven by Wells [Wel94].

Definition 153 We define the set of terms as:

$$t, u \in \mathcal{T}rm := x \mid f \mid t\ u \mid t\ p \mid \lambda x : T.t \mid \Lambda \alpha . t \mid \mathsf{Node} \mid \mathsf{Leaf}$$

with $x \in \mathcal{X}$ a set of term variables, $f \in \Sigma$ is a set of *function symbols* and $\alpha \in \mathcal{V}$.

We give explicit annotations for pattern abstraction and application in relation to the above remark.

Definition 154 We suppose given a *type assignment* $\tau : \Sigma \to \mathcal{T}$, such that for each $f \in \Sigma$, $\tau_f = \forall \alpha_1, \ldots, \alpha_k . A_1 \to \ldots \to A_k \to T_f$ with

- α_i pairwise distinct.
- $A_i = \boldsymbol{B}(\alpha_i)$
- $\forall 1 \le i \le k, \alpha_i$ appears *positively* in T_f.

In this case k is called the number of *recursive arguments*.

The positivity condition is quite similar to the one used in the usual formulation of type-based termination, see for instance Abel [Abe06] for an in depth analysis. This requirement is related to the *cumulativity* of types, which means that the interpretation of a type contains the interpretation of all its subtypes. Note that the positivity condition for defined function symbols

$$\frac{}{\Gamma, x : T, \Delta \vdash x : T}\ \textbf{ax}$$

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda x : T.t : T \to U}\ \textbf{t-lam}$$

$$\alpha \notin \mathcal{FV}(\Gamma)\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \uplambda\alpha.t : \forall\alpha.T}\ \textbf{p-lam}$$

$$\frac{}{\Gamma \vdash \mathrm{Leaf} : \boldsymbol{B}(\mathrm{leaf})}\ \textbf{leaf-intro}$$

$$\frac{}{\Gamma \vdash \mathrm{Node} : \forall\alpha\beta.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \to \boldsymbol{B}(\mathrm{node}(\alpha,\beta))}\ \textbf{node-intro}$$

$$\frac{\Gamma \vdash t : T \to U \qquad \Gamma \vdash u : T}{\Gamma \vdash t\ u : U}\ \textbf{t-app}$$

$$\frac{\Gamma \vdash t : \forall\alpha.T}{\Gamma \vdash t\ p : T\{\alpha \mapsto p\}}\ \textbf{p-app}$$

$$\frac{}{\Gamma \vdash f : \tau_f}\ \textbf{symb}$$

Figure 2.1: Typing Rules

is also guarenteed by the system given in chapter 1, as a consequence of being in *elimination form* (definition 36). The typing rules are also similar to the ones for type-based termination.

The typing rules of our system are given by the typing rules in figure 2.1. To these rules we add the subtyping rule:

$$\frac{\Gamma \vdash t : T \quad T \leq U}{\Gamma \vdash t : U}\textbf{sub}$$

where the subtyping relation is defined in figure 2.2, and which uses the *subpattern* relation $\ll$ on patterns.

Notice the absence of rules allowing us to give the type $\boldsymbol{B}(\bot)$. Intuitively if a term is of type $\boldsymbol{B}(p)$, then all of its normal forms that are values match p. In consequence, a term of type $\boldsymbol{B}(\bot)$ (and that is strongly normalizing) may not reduce to a value. In the same way, a term of type $\boldsymbol{B}(\mathrm{leaf})$ may only have Leaf as a normal form that is a value. We will be able to give a precise interpretation to this intuition with the termination semantics given in the next chapter.

We now prove that type checking is *decidable*, by exhibiting a *syntax directed type inferrence system* which types the same terms as our type system. By syntax directed, we mean that for a given term at most one rule of inferrence can be applied, and whether this rule can be applied is determined by the syntactic structure of the term.

First note that subtyping is decidable: this is a simple consequence of the subtyping rules being syntax directed.

Lemma 155 The subtyping relation is transitive: if T, U, V are types, then

$$T \leq U \wedge U \leq V \Rightarrow T \leq V$$

$$\frac{}{p \ll \top} \qquad \frac{}{\bot \ll p}$$

$$\frac{}{\alpha \ll \alpha} \qquad \frac{}{\mathrm{leaf} \ll \mathrm{leaf}}$$

$$\frac{p_1 \ll q_1 \qquad p_2 \ll q_2}{\mathrm{node}(p_1, p_2) \ll \mathrm{node}(q_1, q_2)}$$

$$\frac{p \ll q}{\boldsymbol{B}(p) \leq \boldsymbol{B}(q)}$$

$$\frac{T_2 \leq T_1 \qquad U_1 \leq U_2}{T_1 \to U_1 \leq T_2 \leq U_2}$$

$$\frac{T \leq U}{\forall\alpha.T \leq \forall\alpha.U}$$

Figure 2.2: Rules for subtyping

Proof. First we prove transitivity for patterns: for p, q, r

$$p \ll q \ll r \Rightarrow p \ll r$$

we proceed by induction on the structure of p:

- $p = \bot$. Then we have $\bot \ll r$.
- $p = \top$. In this case we can only have $q = \top$ and $r = \top$.
- $p = \mathrm{leaf}$. In this case have $q = \top$, in which case $r = \top$ and we are done, or $q = \mathrm{leaf}$ and then $r = \mathrm{leaf}$ or $r = \top$.
- $p = \alpha$. In this case $q = \alpha$ or $q = \top$ and we can conclude as above.
- $p = \mathrm{node}(p_1, p_2)$. In this case either $q = \top$, and we proceed as above, or $q = \mathrm{node}(q_1, q_2)$ and $p_i \ll q_i$ for $i = 1, 2$. In this case either $r = \top$, and we are done, or $r = \mathrm{node}(r_1, r_2)$ and then by induction hypothesis $p_1 \ll r_1$ and $p_2 \ll r_2$ and therefore $p \ll r$.

Now we show transitivity of subtyping by induction on the structure of T:

- $T = \boldsymbol{B}(p)$. In this case we necessarily have $U = \boldsymbol{B}(q)$ and $V = \boldsymbol{B}(r)$ and we may conclude using the transitivity of the subpattern relation.
- $T = T_1 \to T_2$. In this case we have $U = U_1 \to U_2$ with $U_1 \leq T_1$ and $T_2 \leq U_2$, and $V = V_1 \to V_2$ with $V_1 \leq U_1$ and $U_2 \leq V_2$. Induction hypothesis gives $V_1 \leq T_1$ and $T_2 \leq V_2$ and so $T \leq V$.

■

Definition 156 (type synthesis)

Given a context Γ, a term t and a type T, we say that t *synthesizes* T in context Γ, if $\Gamma \vdash t \uparrow T$ can be derived using the rules of figure 2.3.

$$\frac{}{\Gamma, x{:}T, \Delta \vdash x \uparrow T}\ \textbf{ax}$$

$$\frac{\Gamma, x{:}T \vdash t \uparrow U}{\Gamma \vdash \lambda x{:}T.t \uparrow T \to U}\ \textbf{t-lam}$$

$$\alpha \notin \mathcal{FV}(\Gamma)\ \frac{\Gamma \vdash t \uparrow T}{\Gamma \vdash \Lambda\alpha.t \uparrow \forall\alpha.T}\ \textbf{p-lam}$$

$$\frac{}{\Gamma \vdash \mathsf{Leaf} \uparrow \boldsymbol{B}(\mathrm{leaf})}\ \textbf{leaf-intro}$$

$$\frac{}{\Gamma \vdash \mathsf{Node} \uparrow \forall\alpha\beta.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \to \boldsymbol{B}(\mathrm{node}(\alpha,\beta))}\ \textbf{node-intro}$$

$$\frac{\Gamma \vdash t \uparrow T \to U \qquad \Gamma \vdash u \uparrow T' \qquad T' \leq T}{\Gamma \vdash t\ u \uparrow U}\ \textbf{t-app}$$

$$\frac{\Gamma \vdash t \uparrow \forall\alpha.T}{\Gamma \vdash t\ p \uparrow T\{\alpha \mapsto p\}}\ \textbf{p-app}$$

$$\frac{}{\Gamma \vdash f \uparrow \tau_f}\ \textbf{symb}$$

Figure 2.3: Type Synthesis Rules

Type synthesis is decidable, as the rules are syntax directed. It suffices to show that type synthesis allows us to conduct type checking, that is prove the equivalence between $\Gamma \vdash t{:}T$ and $\Gamma \vdash t \uparrow T$.

Theorem 157 (Correctness and completeness of type synthesis)
Suppose Γ is a context, t is a term and T is a type. Then type synthesis is contained in type checking:

$$\Gamma \vdash t \uparrow T \Rightarrow \Gamma \vdash t{:}T$$

Conversely, if t is typeable of type T in Γ, then t synthesizes a type U with $U \leq T$:

$$\Gamma \vdash t{:}T \Rightarrow \Gamma \vdash t \uparrow U \ \wedge\ U \leq T$$

Proof. First suppose that $\Gamma \vdash t \uparrow T$. We prove by induction on the derivation that $\Gamma \vdash t{:}T$. The only interesting case is

t-app: Take $t = t_1\ t_2$ with $\Gamma \vdash t_1 \uparrow T \to U$, $\Gamma \vdash t_2 \uparrow T'$ and $T' \leq T$. By induction hypothesis, $\Gamma \vdash t_1{:}T \to U$ and $\Gamma \vdash t_2{:}T'$. By application of the **sub** rule we conclude that $\Gamma \vdash t_2{:}T$ and we can apply the **app** rule to derive

$$\Gamma \vdash t_1\ t_2{:}U$$

Now suppose that $\Gamma \vdash t{:}T$ we prove by induction on the derivation that there is U such that $\Gamma \vdash t \uparrow U$ with $U \leq T$. We only treat the interesting cases.

- **sub**: we have $\Gamma \vdash t : T'$ with $T' \leq T$. By induction hypothesis there exists U such that

  $$\Gamma \vdash t \uparrow U \wedge U \leq T'$$

  By transitivity of the subtyping relation, $U \leq T$.

- **t-app**: in this case $t = t_1\ t_2$ with $\Gamma \vdash t_1 : T \to U$ and $\Gamma \vdash t_2 : T$. By induction hypothesis, $\Gamma \vdash t_1 \uparrow C$, $\Gamma \vdash t_2 \uparrow A'$ with $C \leq T \to U$ and $A' \leq T$. In this case we must have $C = A \to B$ with $T \leq A$ and $B \leq U$. By transitivity of the subtype relation, $A' \leq A$ and we can apply **t-app** to obtain

  $$\Gamma \vdash t_1\ t_2 \uparrow B$$

  and we have $B \leq U$.

■

2.2 Minimal Typing and the Dependency Graph

Now that we have a type system, we need to build the dependency analysis by examining the rewrite rules. As in chapter 1, we will need to constrain both the shape and type of terms that may appear in the left-hand sides.

Definition 158 Now suppose given a set $\mathcal{R}$ of rewrite rules. A *constructor* is either Leaf or Node, and a *constructor term* is a term with only constructors and variables, with no variable in an application position. We suppose given a rewrite system $\mathcal{R}$ such that each rule $\rho \in \mathcal{R}$ is of the form $f\ p_1 \ldots p_k\ l_1 \ldots l_k \to r$ with k the number of recursive arguments, l_i constructor terms and p_j patterns. We furthermore suppose that for each such rule there is a context Γ and a type T such that:

$$\Gamma \vdash_{\mathrm{min}} f\ p_1 \ldots p_k\ l_1 \ldots l_k : T$$

and $\Gamma \vdash r : T$, with $\vdash_{\mathrm{min}}$ defined in figure 2.4. Suppose in addition that every function symbol $g \in r$ is *fully applied to its pattern arguments*, that is if $\tau_g = \forall \alpha_1 \ldots \alpha_k.T$ then for each occurrence of g in r there are patterns $p_1, \ldots, p_k \in \mathcal{P}$ such that $g\ p_1 \ldots p_k$ appears at that position.

Minimal typing here is essentially identical to minimal typing as defined in chapter 1 (definition 39), and serves a similar function, that is to force the types to adequately denote the semantics of the terms.

We can then define the higher-order analogue of dependency pairs, which use the type information instead of the term information. The type information for the left hand side is deduced from the type inference, so we shall need unicity of the inferred type.

Lemma 159 Suppose that $\Gamma \vdash_{\mathrm{min}} l : T$ with l a constructor term. Then T is unique modulo variable renaming.

Proof. The proof is an easy induction, given that all rules are syntax directed. ■

Note that, in the typed framework, a dependency pair is not formally a (higher-order) rewrite rule.

$$\frac{}{\Gamma, x : \boldsymbol{B}(\alpha), \Gamma' \vdash_{\text{min}} x : \boldsymbol{B}(\alpha)}\ \alpha \notin \Gamma, \Gamma'$$

$$\frac{}{\Gamma \vdash_{\text{min}} \text{Leaf} : \boldsymbol{B}(\text{leaf})}$$

$$\frac{\Gamma \vdash_{\text{min}} l_1 : \boldsymbol{B}(p_1) \qquad \Gamma \vdash_{\text{min}} l_2 : \boldsymbol{B}(p_2)}{\Gamma \vdash_{\text{min}} \text{Node}\ \ p_1\ p_2\ l_1\ l_2 : \boldsymbol{B}(\text{node}(p_1, p_2))}$$

$$\frac{\Gamma \vdash_{\text{min}} l_1 : \boldsymbol{B}(p_1) \qquad \dots \qquad \Gamma \vdash_{\text{min}} l_k : \boldsymbol{B}(p_k)}{\Gamma \vdash_{\text{min}} f\ p_1 \dots p_k\ l_1 \dots l_k : T_f \phi}\ \vec{\alpha} \notin \Gamma$$

with $\tau_f = \forall \alpha_1 \dots \alpha_k . A_1 \to \dots \to A_k \to T_f$ and $\phi(\alpha_i) = p_i$ for $1 \leq i \leq k$.

Figure 2.4: Minimal Typing Rules

Definition 160 (Type dependency pairs)
Let $\rho := f\ \vec{p}\ \vec{l} \to r$ be a rule in $\mathcal{R}$, with Γ such that $\Gamma \vdash_{min} f\ \vec{p}\ \vec{l} : T$, and $\Gamma \vdash r : T$. The set of *type dependency pairs* $DP_{\mathcal{T}}(\rho)$ is the set

$$\{f^\sharp(p_1, \dots, p_k) \to g^\sharp(q_1, \dots, q_l) \mid \forall i, \Gamma \vdash_{\text{min}} l_i : \boldsymbol{B}(p_i) \land g\ q_1 \dots q_l \text{ appears in } r\}$$

The set $DP_{\mathcal{T}}(\mathcal{R})$ is defined as the union of all $DP_{\mathcal{T}}(\rho)$, for $\rho \in \mathcal{R}$, where we suppose that all variables are disjoint between dependency pairs.

The set of higher-order dependency pairs defined above should already be seen as an abstraction of the dependency pair notion defined in the previous chapter (definition 140). Indeed, thanks to subtyping, there may be some information loss in the types, if for instance the $\top$ pattern is used. As an example, if f, g and h all have the type $\forall \alpha . \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\top)$, consider the rule

$$f\ \alpha\ x \to g\ \top\ (h\ \alpha\ x)$$

The dependency pairs we obtain are

$$f^\sharp(\alpha) \to g^\sharp(\top), \quad f^\sharp(\alpha) \to h^\sharp(\alpha)$$

In the first dependency pair, the information that g is called on the argument $h\ x$ is lost. However one may observe that in the case of the simple approximation to the dependency graph, the fact that h is a defined symbol gives $rencap(g^\sharp\ (h\ x)) = g^\sharp\ y$ with y some fresh variable, which effectively gives us the same information as our typed dependency pair. This approach can therefore be seen as a type based manner to study the standard approximated dependency graph. Note that in the case where h is given a more precise type, like $\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\text{leaf})$, which is the case if every normal form of $h\ t$ is either neutral or Leaf, we have a more precise approximation than the standard approximated dependency graph.

Definition 161 Let p and q be patterns. We say that p and q are *pattern-unifyable*, and write $p \bowtie q$, if p' and q' are unifyable, where p' and q' are the patterns p and q in which each occurrence of $\top$ and each occurrence of a variable is replaced by some *fresh* variable. An alternative defintion for $\bowtie$ is the smallest relation that verifies:

- $p \bowtie \top$

- $p \bowtie \alpha$
- $p_1 \bowtie q_1 \wedge p_2 \bowtie q_2 \Rightarrow \mathrm{node}(p_1, p_2) \bowtie \mathrm{node}(q_1, q_2)$
- $\mathrm{leaf} \bowtie \mathrm{leaf}$
- $\bot \bowtie \bot$
- $p \bowtie q \Rightarrow q \bowtie p$

for any p, q patterns and α any pattern variable.

We then write $f^\sharp(p_1, \ldots, p_k) \bowtie g^\sharp(q_1, \ldots, q_l)$ if $f^\sharp = g^\sharp$, $k = l$ and $p_i \bowtie q_i$ for each $1 \leq i \leq k$.

The *standard typed dependency graph* $\mathcal{G}_\mathcal{R}$ is defined as the graph with

- As set of nodes the set $DP_\mathcal{T}(\mathcal{R})$.
- There is an edge between the dependency pairs $t_1 \to u_1$ and $t_2 \to u_2$ if

$$u_1 \bowtie t_2$$

We have in this definition an adequate higher-order notion of standard approximated dependency graph.

Definition 162 If p and q do not contain any occurence of $\top$, we define the *embeddeding preorder* $p \rhd q$ by the following rules

- $p_i \unrhd q \Rightarrow \mathrm{node}(p_1, p_2) \rhd q$ for $i = 1, 2$
- $p_1 \rhd q_1 \wedge p_2 \unrhd q_2 \Rightarrow \mathrm{node}(p_1, p_2) \rhd \mathrm{node}(q_1, q_2)$
- $p_1 \unrhd q_1 \wedge p_2 \rhd q_2 \Rightarrow \mathrm{node}(p_1, p_2) \rhd \mathrm{node}(q_1, q_2)$

with $\unrhd$ as the reflexive closure of $\rhd$.

2.3 Erased Terms and the Main Theorem

We shall need to revise our definition of reduction in order to state (and prove) the normalization theorem. The problem with our current definition of rewriting arises when trying to match on patterns. Take the rule

$$f\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\ \ x\ y) \to \mathrm{Leaf}$$

In the presence of this rule, we wish to have, for instance, the reduction

$$f\ \top\ (\mathrm{Node}\ (g\ x)\ (h\ x)) \to \mathrm{Leaf}$$

However, there is no substitution θ such that $\mathrm{node}(\alpha,\beta)\theta = \top$. There are two ways to deal with this. Either we take the subpattern order into account when performing matching, that is match on terms *modulo* $\ll$, or we do away with the pattern arguments when performing reduction. We adopt the second solution, as it is used in practice when dealing with languages with dependent type annotations (see for example McKinna [McK06]). Symmetrically, we erase pattern abstractions as well.

Definition 163 We define the set of *erased terms* $\mathcal{T}rm^{|\cdot|}$ as:

$$t, u \in \mathcal{T}rm^{|\cdot|} := x \mid f \mid \lambda x.t \mid t\ u \mid \mathrm{Leaf} \mid \mathrm{Node}$$

where $x \in \mathcal{X}$ and $f \in \mathcal{F}$.

Given a term $t \in \mathcal{T}rm$, we define the *erasure* $|t| \in \mathcal{T}rm^{|\cdot|}$ of t as:

$$\begin{array}{lcl} |x| & = & x \\ |f| & = & f \\ |\lambda x\!:\!T.t| & = & \lambda x.|t| \\ |\Lambda\alpha.t| & = & |t| \\ |t\ u| & = & |t|\ |u| \\ |t\ p| & = & |t| \\ |\,\mathrm{Leaf}\,| & = & \mathrm{Leaf} \\ |\,\mathrm{Node}\,| & = & \mathrm{Node} \end{array}$$

An erased term can intuitively be thought of as the compiled form of a well typed term.

Definition 164 An erased term t *head rewrites* to a term u if there is some rule $l \to r \in \mathcal{R}$ and some substitution σ from $\mathcal{X}$ to terms in $\mathcal{T}rm^{|\cdot|}$ such that

$$|l|\sigma = t \wedge |r|\sigma = u$$

We then define $\to^*_{\mathcal{R}\cup\beta}$ and $\to^+_{\mathcal{R}\cup\beta}$ as usual.

We can now express our termination criterion. The idea is quite close to the dependency pair processors exposed in the theorems 144 and 148 from chapter 1, using the simple projection criterion.

Theorem 165 (Type-based dependency pair criterion)
Let $\mathcal{G}$ be the typed dependency graph for $\mathcal{R}$ and suppose that for every SCC $\mathcal{G}_1, \ldots, \mathcal{G}_n$, there is a *simple projection* $\iota^1, \ldots, \iota^n$ which to each function symbol $f \in \Sigma$ associates an integer $1 \le \iota^i_f \le k$ with k the number of recursive arguments. Suppose that for each $1 \le i \le n$ and each rule $f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_n)$ in $\mathcal{G}_i$, we have $p_{\iota^i_f} \trianglerighteq q_{\iota^i_g}$. Finally suppose that for each cycle in $\mathcal{G}_i$, there is some rule $f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_n)$ such that

$$p_{\iota^i_f} \rhd q_{\iota^i_g}$$

then for every Γ, t, T such that $\Gamma \vdash t : T$,

$$|t| \in \mathcal{SN}_{\mathcal{R}}$$

The next chapter is devoted to the proof of this theorem.

We give a representative example of our criterion.

Example 17 Consider the following rewrite system:

- The signature is given by $\{\mathrm{app}, a, g, h\}$

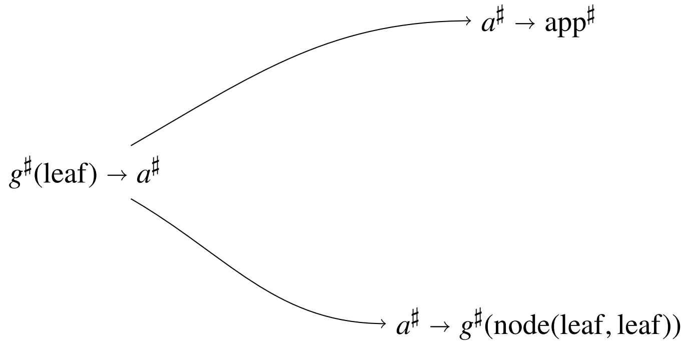

$a^\sharp \to \mathsf{app}^\sharp$
$g^\sharp(\mathrm{leaf}) \to a^\sharp$
$a^\sharp \to g^\sharp(\mathrm{node}(\mathrm{leaf}, \mathrm{leaf}))$

Figure 2.5: Dependency graph of example 17

- The type assignment τ is defined by

$$\begin{array}{lcl}
\tau_{\mathsf{app}} & = & \forall\alpha\beta.(\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta)) \to \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \\
\tau_a & = & \boldsymbol{B}(\mathrm{node}(\mathrm{leaf}, \mathrm{leaf})) \\
\tau_g & = & \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\mathrm{node}(\mathrm{leaf}, \mathrm{leaf})) \\
\tau_h & = & \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\mathrm{leaf})
\end{array}$$

where app and a have 0 recursive arguments, and g and h both have 1.

- And the rules are given by

$$\begin{array}{lcl}
\mathsf{app} & \to & \lambda\alpha\beta.\lambda x{:}\,\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta).\lambda y{:}\,\boldsymbol{B}(\alpha).x\ y \\
a & \to & \mathsf{app}\ \ \mathrm{node}(leaf, leaf)\ \ \mathrm{leaf}\ (g\ \ \mathrm{node}(\mathrm{leaf}, \mathrm{leaf}))\ (\mathrm{Node}\,\mathrm{leaf}\,\mathrm{leaf}\ \ \mathrm{Leaf}\ \ \mathrm{Leaf}) \\
g\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\,\alpha\ \beta\ x\ y) & \to & \mathrm{Leaf} \\
g\ \ \mathrm{leaf}\ \ \mathrm{Leaf} & \to & a
\end{array}$$

Or in a more readable form with the type and pattern arguments omited:

$$\begin{array}{lcl}
\mathsf{app} & \to & \lambda x.\lambda y.x\ y \\
a & \to & \mathsf{app}\ \ g\ (\mathrm{Node}\ \ \mathrm{Leaf}\ \ \mathrm{Leaf}) \\
g(\mathrm{Node}\ \ x\ y) & \to & \mathrm{Leaf} \\
g\ \ \mathrm{Leaf} & \to & a
\end{array}$$

It is possible to verify that the criterion can be applied and that in consequence, according to theorem 165, all well typed terms are strongly normalizing under $\mathcal{R} \cup \beta$. Indeed, we may easily check that each of these rules is minimally typed in the context $x{:}\,\boldsymbol{B}(\alpha), y{:}\,\boldsymbol{B}(\beta)$, and furthermore, the dependency graph in figure 2.5 has no cycles.

One may object that if we inline the definition of app and perform β-reduction on the right-hand sides of rules we obtain a rewrite system that can be treated with more conventional methods, such as those performed by the `AProVe` tool [GTSK05] (on terms without abstraction, and without β-reduction). However this operation can be very costly if performed automatically and is, in its most naïve form, ineffective for even slightly more complex higher-order programs such as *map*, which performs pattern matching and for which we would need to instantiate certain variables. By resorting to typing, we allow termination to be proven using only “local” considerations, as the information encoding the semantics of app is contained in its type.

However it becomes necessary, if one desires a fully automated termination check on an unannotated system, to somehow infer the type of defined constants, and possibly perform an analysis quite similar in effect to the one proposed above. We believe that to this end one may apply known type inference technology, such as the one described in Chin and Khoo [CK01], to compute these annotated types. In conclusion, what used to be a termination problem becomes a type inference problem, and may benefit from the knowledge and techniques of this new community, as well as facilitate integration of these techniques into type-theoretic based proof assistants like Coq [Coq08].

Let us examine a second, slightly more complex example, in which there is "real" recursion.

Example 18 Let $\mathcal{R}$ be the rewrite system defined by

$$\begin{array}{rcl} f\ (\mathrm{Node}\ \ x\ y) & \to & g\ (i\ (\mathrm{Node}\ \ x\ y)) \\ g\ (\mathrm{Node}\ \ x\ y) & \to & f\ (i\ x) \\ g\ \ \mathrm{Leaf} & \to & f\ (h\ \ \mathrm{Leaf}) \\ i\ (\mathrm{Node}\ \ x\ y) & \to & \mathrm{Node}\ (i\ x)\ (i\ y) \\ i\ \ \mathrm{Leaf} & \to & \mathrm{Leaf} \\ h\ (\mathrm{Node}\ \ x\ y) & \to & h\ x \end{array}$$

again with the type arguments omited, and with types $f, g\colon \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\top)$, $h\colon \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\bot)$ and $i\colon \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)$. Every equation can by typed in the context $\Gamma = x\colon \boldsymbol{B}(\alpha), y\colon \boldsymbol{B}(\beta)$. The system with full type annotations is given by:

$$\begin{array}{rcl} f\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\ \ \alpha\ \beta\ x\ y) & \to & g\ \ \mathrm{node}(\alpha,\beta)\ (i\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\ \ \alpha\ \beta\ x\ y)) \\ g\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\ \ \alpha\ \beta\ x\ y) & \to & f\ \alpha\ (i\ \alpha\ x) \\ g\ \ \mathrm{leaf}\ \ \mathrm{Leaf} & \to & f\ \bot\ (h\ \ \mathrm{leaf}\ \ \mathrm{Leaf}) \\ i\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\ \ \alpha\ \beta\ x\ y) & \to & \mathrm{Node}\ \ \mathrm{node}(\alpha,\beta)\ (i\ \alpha\ x)\ (i\ \beta\ y) \\ i\ \ \mathrm{leaf}\ \ \mathrm{Leaf} & \to & \mathrm{Leaf} \\ h\ \ \mathrm{node}(\alpha,\beta)\ (\mathrm{Node}\ \ \alpha\ \beta\ x\ y) & \to & h\ \alpha\ x \end{array}$$

The dependency graph is given in figure 2.3 with as dotted edges those that do not contribute to cycles, and has as SCCs the full subgraphs of $\mathcal{G}_{\mathcal{R}}$ with nodes

$$\left\{i^\sharp(\mathrm{node}(\alpha,\beta)) \to i^\sharp(\alpha),\ \ i^\sharp(\mathrm{node}(\alpha,\beta)) \to i^\sharp(\beta)\right\}$$

$$\left\{f^\sharp(\mathrm{node}(\alpha,\beta)) \to g^\sharp(\mathrm{node}(\alpha,\beta),\ \ g^\sharp(\mathrm{node}(\alpha,\beta) \to f^\sharp(\alpha)\right\}$$

$$\left\{h^\sharp(\mathrm{node}(\alpha,\beta)) \to h^\sharp(\alpha)\right\}$$

respectively. Taking $\iota_s = 1$ for every SCC and every symbol $s \in \Sigma$, it is easy to show that every SCC respects the decrease criterion on cycles. For example, in the cycle

$$f^\sharp(\mathrm{node}(\alpha,\beta)) \to g^\sharp(\mathrm{node}(\alpha,\beta)) \leftrightarrows g^\sharp(\mathrm{node}(\alpha,\beta)) \to f^\sharp(\alpha)$$

we have $\mathrm{node}(\alpha,\beta) \trianglerighteq \mathrm{node}(\alpha,\beta)$ and $\mathrm{node}(\alpha,\beta) \triangleright \alpha$, so the cycle is weakly decreasing with at least one strict decrease.

We may then again apply the correctness theorem to conclude that the erasure of all well-typed terms are strongly normalizing with respect to $\mathcal{R} \cup \beta$.

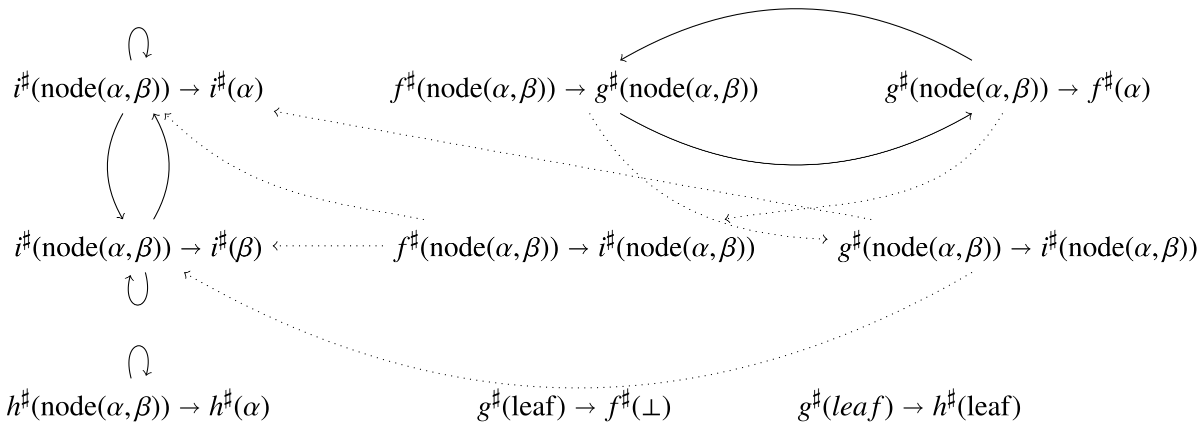

$i^\sharp(\text{node}(\alpha,\beta)) \to i^\sharp(\alpha)$
$f^\sharp(\text{node}(\alpha,\beta)) \to g^\sharp(\text{node}(\alpha,\beta))$
$g^\sharp(\text{node}(\alpha,\beta)) \to f^\sharp(\alpha)$
$i^\sharp(\text{node}(\alpha,\beta)) \to i^\sharp(\beta)$
$f^\sharp(\text{node}(\alpha,\beta)) \to i^\sharp(\text{node}(\alpha,\beta))$
$g^\sharp(\text{node}(\alpha,\beta)) \to i^\sharp(\text{node}(\alpha,\beta))$
$h^\sharp(\text{node}(\alpha,\beta)) \to h^\sharp(\alpha)$
$g^\sharp(\text{leaf}) \to f^\sharp(\bot)$
$g^\sharp(leaf) \to h^\sharp(\text{leaf})$

Figure 2.6: The dependency graph for example 18

3

Correctness

The proof of the termination theorem uses computability predicates. As mentioned before, the absence of control, and particularly the lack of orthogonality makes giving accurate semantics somewhat difficult. We draw inspiration from the termination semantics of Berger [Ber05], which uses sets of values to denote terms. As is standard in computability proofs, each type will be interpreted as a set of strongly normalizing (erased) terms. Suppose a term t reduces to the normal forms Leaf and Node Leaf Leaf. In that case t is in the candidate that contains all terms that reduce to Leaf or Node Leaf Leaf, or are hereditarily neutral. If t is the erasure of a term of type $\boldsymbol{B}(\alpha)$ for some pattern variable α, the interpretation $[\![\boldsymbol{B}(\alpha)]\!]$ must depend on some valuation of the free variable α. If we valuate α by some closed pattern p and interpret $[\![\boldsymbol{B}(\alpha)]\!]$ by the set of terms whose normal forms are neutral or match p, then the only possible choice for p is $\top$. Indeed if α is interpreted by p then both Leaf and Node Leaf Leaf must match p which is only the case if $p = \top$.

Clearly this does not give us the most precise possible semantics for t, as the set of terms which match $\top$ also includes terms such as u = Node (Node x y) Leaf. We need more precise semantics if we are to capture the information needed for the dependency analysis: if we take the constructor term l = Node *Leaf* x, then a reduct of t does match l, but this can never happen for u. To give sufficiently precise semantics to terms, we therefore need to interpret pattern variables with *sets of closed patterns*. In this case we will interpret α by the set $\{\text{leaf}, \text{node}(\text{leaf}, \text{leaf})\}$ to capture sufficiently precise semantics for t.

Let us briefly sketch the termination proof. We first define a *term-matching* relation, that allows us to relate normal forms and closed type-patterns. We use this relation to define the interpretation of base types $\boldsymbol{B}(p)$ as the type of terms t such that the normal forms of t term-match p. We then proceed as usual to define the interpretation $[\![T]\!]_\theta$ of a type T with respect to a valuation of pattern variables θ. We show that this interpretation satisfies the Girard conditions and that it is a correct interpretation with respect to all the typing rules, except the **symb** rule.

Then we must show the correctness of the interpretation with respect to the **symb** rule, that is $f \in [\![\tau_f]\!]$ for every defined symbol. In order to do this, we need to relate the possible reductions in the operational semantics to the dependency graph $\mathcal{G}_\mathcal{R}$. For a given rule $\rho = f\vec{l} \to r$, if $l_i : \boldsymbol{B}(p_i)$, and if there is a call in r to the function g of type $\boldsymbol{B}(q_1) \to \ldots \to \boldsymbol{B}(q_m) \to T_g$, then we show that for every instance $\vec{t}$ of $\vec{l}$ there is a *minimal* type-valuation ψ such that for each i, $t_i \in [\![\boldsymbol{B}(p)]\!]_\psi$. Then we show for each j, u_j is in $[\![\boldsymbol{B}(q_i)]\!]_\psi$, and if $p_i \rhd q_j$ (resp. $\trianglerighteq$) then there is a strict decrease (resp. large decrease) between t_i and u_j in some well-founded order.

Finally if the u_j reduce to some u'_j, and this leads to the application of another rule $\rho' = g\vec{l'} \to$

r', then there is an edge between the node $f^\sharp(\vec{p}) \to g^\sharp(\vec{q})$ and the nodes that correspond to the rule ρ'. Then it only remains to show that this does not break the decrease in the aforementioned well-founded order, and we will be able to prove $f \in [\![\tau_f]\!]$ by induction on this order.

3.1 The Type Interpretation and Conditional Correctness

We define the interpretation of types, and prove that they satisfy the Girard conditions. We then show that correctness of the defined function symbols implies correctness of the semantics, as in the proof of theorem 124.

Definition 166 A *value* is a term $v \in \mathcal{T}rm^{|\cdot|}$ of the form:

- $\lambda x.t$
- $\mathrm{Node}\ \ t\ u$
- Leaf

For any $t \in \mathcal{T}rm^{|\cdot|}$ we say v is a *value of* t if $t \to^*_{\mathcal{R}\cup\beta} v$ and v is a value.

A term is *neutral* if it is not a value, and is *hereditarily neutral* if it has no values.

We introduce *term-matching* as the mechanism for relating terms and type-patterns. It resembles ordinary pattern matching, with the difference that no substitution needs to be computed, so it is just a relation. In the semantics we only consider closed patterns, as the variables in patterns themselves only represent potential pattern instantiations.

Definition 167 Let $\mathcal{P}_c$ be the set of *closed* patterns, and $\mathcal{NF}$ the set of *$\mathcal{R}\beta$-normal forms* in $\mathcal{T}rm^{|\cdot|}$. The *term matching relation* $\ll \subseteq \mathcal{NF} \times \mathcal{P}_c$ is defined in the following way:

- $v \ll \top$
- $v \ll p$ if v is neutral.
- $v \ll \mathrm{node}(p, q)$ if $v = \mathrm{Node}\ \ v_1\ v_2$ with $v_1 \ll p \wedge v_2 \ll q$.
- $v \ll \mathrm{leaf}$ if $v = \mathrm{Leaf}$.

A *pattern valuation*, or *valuation* if the context is clear, is a *partial* function with finite support from pattern variables $\mathcal{V}$ to *non-empty* sets of *closed* patterns. If p is a pattern, θ is a pattern valuation and $\mathcal{FV}(p) \subseteq \mathrm{dom}(\theta)$ then $p\theta$ is the set defined inductively by:

- $\alpha\theta = \theta(\alpha)$
- $\mathrm{leaf}\,\theta = \{\mathrm{leaf}\}$
- $\top\theta = \{\top\}$
- $\bot\theta = \{\bot\}$
- $\mathrm{node}(p_1, p_2)\theta = \{\mathrm{node}(q_1, q_2) \mid q_1 \in p_1\theta \wedge q_2 \in p_2\theta\}$

We may write $p\theta = \{p \mid \alpha_1 \leftarrow \theta(\alpha_1), \ldots, \alpha_n \leftarrow \theta(\alpha_n)\}$ where the α_i are the variables of p, by analogy with *list comprehension* notation (as in Berger [Ber05]). If $\alpha \notin \mathsf{dom}(\theta)$ and P is a set of closed patterns, we write θ^α_P for the valuation that sends $\beta \in \mathsf{dom}(\theta)$ to $\theta(\beta)$ and α to P. Notice that $p\theta$ is a set of *closed* patterns.

Finally if P is a set of closed patterns and t is a term in $\mathcal{SN}$, we write $t \downarrow\!\ll P$ if for every *normal form* v of t:

$$\exists p \in P,\ v \ll p$$

We define $\mathcal{B}$ to be the smallest set that verifies:

$$\mathcal{B} = \{t \in \mathcal{SN} \mid \forall v \text{ a value of } t, v = \text{Leaf } \vee v = \text{Node } t_1\ t_2 \wedge t_1, t_2 \in \mathcal{B}\}$$

That this set exists can be proven using the Tarski fixed-point theorem exactly in the same manner as in the proof of theorem 124.

The *type interpretation* $[\![\_]\!]_\_$ is a function that to each $T \in \mathcal{T}$ and each valuation θ such that $\mathcal{FV}(T) \subseteq \mathsf{dom}(\theta)$ associates a set $[\![T]\!]_\theta \subseteq \mathcal{SN}_{\mathcal{R}\cup\beta}$. We define it by induction on the structure of T:

- $[\![\boldsymbol{B}(p)]\!]_\theta = \{t \in \mathcal{B} \mid t \downarrow\!\ll p\theta\}$
- $[\![T \to U]\!]_\theta = \{t \in \mathcal{SN} \mid \forall u \in [\![T]\!]_\theta, t\ u \in [\![U]\!]_\theta\}$
- $[\![\forall\alpha.T]\!]_\theta = \{t \in \mathcal{SN} \mid \forall P, t \in [\![T]\!]_{\theta^\alpha_P}\}$

Where P is a non-empty set of closed patterns.

We proceed as in the proof of theorem 3, and show that each type interpretation is a reducibility candidate.

Lemma 168 (Girard conditions 21)
If $T \in \mathcal{T}$ is a type, then for every valuation θ,

$$[\![T]\!]_\theta \text{ satisfies the Girard conditions}$$

Proof. We proceed by induction on the structure of T.

- $T = \boldsymbol{B}(p)$
  - Strong normalization: by definition of $\mathcal{B}$.
  - Stability by reduction. Suppose that $t \in [\![\boldsymbol{B}(p)]\!]_\theta$. If $t \to^* u$, then the set of normal forms of u is contained in the set of normal forms of t.
  - Sheaf condition. Suppose that t is neutral and that each one step reduct of t is in $[\![\boldsymbol{B}(p)]\!]_\theta$. Now either t is in normal form, and then $t \downarrow\!\ll p\theta$ (as it is non empty), or for every one step reduct u of t, $u \downarrow\!\ll p\theta$. But in this case every normal form of t is the normal form of some $t \to u$, and thus $t \downarrow\!\ll p\theta$.
- $T = T_1 \to T_2$ This is shown in the same way as for lemma 21.
- $T = \forall\alpha.U$

- Strong normalization: by definition.
- Stability by reduction: Let $t \in [\![\forall\alpha.U]\!]_\theta$. We have for every set P of closed terms, $t \in [\![U]\!]_{\theta^\alpha_P}$. By induction, every reduct u of t is also in $[\![U]\!]_{\theta^\alpha_P}$. As P was chosen arbitrarily, u is also in $[\![\forall\alpha.U]\!]_\theta$.
- Sheaf condition. Let t be neutral and suppose that one step reducts of t are in $[\![\forall\alpha.U]\!]_\theta$. Take an arbitrary P. Every reduct of t is in $[\![U]\!]_{\theta^\alpha_P}$. By induction hypothesis, t is in $[\![U]\!]_{\theta^\alpha_P}$, from which we may conclude.

■

Now we give the conditional correctness theorem, which states that if the function symbols belong to the interpretation of their types, then so does every well-typed term, which is exactly the approach taken with theorem 124.

Definition 169 Let θ be a valuation, σ a substitution from term variables to erased terms, and Γ a context. We say that (θ,σ) *validates* Γ, and we write $\sigma \models_\theta \Gamma$, if the set of free pattern variables in Γ is contained in $\mathsf{dom}(\theta)$, and if for every $x \in \mathsf{dom}(\Gamma)$

$$\sigma(x) \in [\![\Gamma(x)]\!]_\theta$$

Likewise, we write $\sigma \models_\theta t : T$ if $\mathcal{FV}(t) \subseteq \mathsf{dom}(\sigma)$, $\mathcal{FV}(T) \subseteq \mathsf{dom}(\theta)$ and

$$|t|\sigma \in [\![T]\!]_\theta$$

Theorem 170 (Relative correctness of the semantics)
Suppose that for each $f \in \Sigma$ and each valuation θ,

$$f \in [\![\tau_f]\!]_\theta$$

then for every context Γ, term t and type T, if $\Gamma \vdash t : T$

$$\forall(\theta,\sigma),\ \sigma \models_\theta \Gamma \Rightarrow \sigma \models_\theta t : T$$

We need the *substitution lemma* for types:

Lemma 171 For every patterns q, p and valuation θ, if α is not in the domain of θ, then

$$p\{\alpha \mapsto q\}\theta = p\theta^\alpha_{q\theta}$$

Proof. We proceed by induction on the structure of p:

- $p = \alpha$: trivial.
- $p = \beta \neq \alpha$: We have $p\{\alpha \mapsto q\} = \beta$ and therefore $p\{\alpha \mapsto q\}\theta = \theta(\beta) = \theta^\alpha_{q\theta}(\beta)$.
- $p = \mathsf{leaf}, \top, \bot$: trivial.
- $p = \mathsf{node}(p_1,p_2)$: We have $p\{\alpha \mapsto q\}\theta = \mathsf{node}(p_1\{\alpha \mapsto q\}, p_2\{\alpha \mapsto q\})\theta$. But this last term is equal to
$$\{\mathsf{node}(q_1,q_2) \mid q_i \in p_i\{\alpha \mapsto q\}\theta, i = 1,2\}$$
which by induction is equal to
$$\{\mathsf{node}(q_1,q_2) \mid q_i \in p_i\theta^\alpha_{q\theta}, i = 1,2\}$$
which allows us to conclude.

■

Lemma 172 (Substitution lemma)
Let T be a type and θ a pattern valuation. If α does not appear in the domain of θ then:

$$[\![T\{\alpha \mapsto p\}]\!]_{\theta} = [\![T]\!]_{\theta^{\alpha}_{p\theta}}$$

Proof. We proceed by induction on the type.

- Atomic case:
$$[\![\boldsymbol{B}(q)\{\alpha \mapsto p\}]\!]_{\theta} = \{t \in \mathcal{B} \mid t \downarrow\ll q\{\alpha \mapsto p\}\theta\}$$
But by lemma 171, $q\{\alpha \mapsto p\}\theta = p\theta^{\alpha}_{p\theta}$, from which we can conclude.
- Arrow case: straightforward from induction hypothesis.
- case $\forall\beta.T$. We may suppose by the Barendregt convention that β is distinct from α, not in the domain of θ and distinct from all variables in p. We then have:
$$[\![(\forall\beta.T)\{\alpha \mapsto p\}]\!]_{\theta} = \{t \in \mathcal{SN} \mid \forall Q, t \in [\![T\{\alpha \mapsto p\}]\!]_{\theta^{\beta}_{Q}}\}$$
Let $\theta' = \theta^{\beta}_{Q}$. We may apply the induction hypothesis, which gives:
$$[\![T\{\alpha \mapsto p\}]\!]_{\theta'} = [\![T]\!]_{\theta'^{\alpha}_{p\theta'}}$$
And as β does not appear in p:
$$[\![T]\!]_{\theta'^{\alpha}_{p\theta'}} = [\![T]\!]_{\theta^{\alpha}_{p\theta}{}^{\beta}_{Q}}$$
But we have:
$$\{t \in \mathcal{SN} \mid \forall Q, t \in [\![T]\!]_{\theta^{\alpha}_{p\theta}{}^{\beta}_{Q}}\} = [\![\forall\beta.T]\!]_{\theta^{\alpha}_{p\theta}}$$
Which concludes the argument.

■

We may easily generalize this result to:

Corollary 173 Let T be a type. If ϕ is a substitution, and θ is a valuation such that the variables of T do not appear in the domain of θ, then:

$$[\![T\phi]\!]_{\theta} = [\![T]\!]_{\theta\circ\phi}$$

Where $\theta \circ \phi$ is the valuation defined by $\theta \circ \phi(\alpha) = \phi(\alpha)\theta$.

Another useful lemma states that type interpretations only depend on the value of the pattern substitutions in the free variables of the type.

Lemma 174 Let T be some type and θ, θ' be two closed pattern substitutions. If $\theta(\alpha) = \theta'(\alpha)$ for every $\alpha \in \mathcal{FV}(T)$, then $[\![T]\!]_{\theta} = [\![T]\!]_{\theta'}$.

Proof. Straightforward induction on T.

The next lemmas proceed to show correctness of the interpretation with respect to subtyping.

Definition 175 Let P and Q be sets of closed patterns. We write $P \ll Q$ if for each $p \in P$, there is a $q \in Q$ such that $p \ll q$.

The definition of $\ll$ for sets of closed patterns is the one we need for this correctness lemma. Intuitively the set of terms t such that $t \downarrow\lll P$ is included in the set of terms u such that $u \downarrow\lll Q$, as we may "transport" a witness p of $t \downarrow\lll P$ into Q.

Lemma 176 Let θ be a pattern valuation. If $p \ll q$, then $p\theta \ll q\theta$

Proof. Induction on the derivation of $p \ll q$. The only non-trivial case is $\mathsf{node}(p_1, p_2) \ll \mathsf{node}(q_1, q_2)$ with $p_i \ll q_i$ for $i = 1, 2$. In that case, if $r \in \mathsf{node}(p_1, p_2)\theta$, we have $r = \mathsf{node}(r_1, r_2)$ with $r_i \in p_i\theta$ for $i = 1, 2$. By induction hypothesis, there is r'_1, r'_2 in $\mathsf{node}(q_1, q_2)\theta$ such that $r_i \ll r'_i$ for each i. Then we take $\mathsf{node}(r'_1, r'_2) \in \mathsf{node}(q_1, q_2)\theta$ to conclude.

■

Lemma 177 (Correctness of subtyping)
Suppose $T \leq U$. Then for all θ, $[\![T]\!]_\theta \subseteq [\![U]\!]_\theta$

Proof. We proceed by induction on all the possible cases for the judgement $T \leq U$.

- $p \ll q$: We first show that for all terms t, and every non-empty set of closed patterns P and Q, if $P \ll Q$, then $t \downarrow\lll P \Rightarrow t \downarrow\lll Q$. This follows from the following fact: if v is in normal form and $r \ll s$, then

  $$v \lll r \Rightarrow v \lll s$$

  To show this we proceed by induction on the $\ll$ judgement. The first three cases are easy. In the fourth case, $v \lll \mathsf{node}(r_1, r_2)$ which by definition implies that $v = \mathsf{Node}\ v_1\ v_2$, with $v_1 \lll r_1$ and $v_2 \lll r_2$. We can then conclude by the induction hypothesis.

  Now using lemma 176, we have, if $p \ll q$, $t \downarrow\lll p\theta \Rightarrow t \downarrow\lll q\theta$.

  Now let $t \in [\![\boldsymbol{B}(p)]\!]_\theta$, we have by definition $t \downarrow\lll p\theta$, and by the previous remark, $t \downarrow\lll q\theta$ which implies $t \in [\![\boldsymbol{B}(q)]\!]_\theta$.

- Suppose $T_2 \leq T_1$ and $U_1 \leq U_2$. Let t be in $[\![T_1 \to U_2]\!]_\theta$, we show that it is in $[\![T_2 \to U_2]\!]_\theta$. Let u be in $[\![T_2]\!]_\theta$. By the induction hypothesis, $u \in [\![T_1]\!]_\theta$, therefore (by definition of $[\![T_1 \to U_1]\!]_\theta$), $t\ u$ is in $[\![U_1]\!]_\theta$, which by another application of the induction hypothesis, is included in $[\![U_2]\!]_\theta$. From this we can conclude that t is in $[\![T_2 \to U_2]\!]_\theta$.

- Let t be a term in $[\![\forall\alpha.T]\!]_\theta$ and P be some arbitrary set of closed patterns, and suppose that α is a variable not appearing in the domain of θ. We then have

  $$t \in [\![T]\!]_{\theta^\alpha_P}$$

  Since $\forall\alpha.T \leq \forall\alpha.U$, we have $T \leq U$. The induction hypothesis gives:

  $$[\![T]\!]_{\theta'} \subseteq [\![U]\!]_{\theta'}$$

  for all valuations θ'. Take θ' to be θ^α_P. We have:

  $$[\![T]\!]_{\theta^\alpha_P} \subseteq [\![U]\!]_{\theta^\alpha_P}$$

  From this we can deduce $t \in [\![U]\!]_{\theta^\alpha_P}$ and conclude.

■

We shall also need the fact that given T and a valuation θ, then $[\![T]\!]_\theta$ is included in $[\![T]\!]_{\theta'}$ if θ' is a *weakening* of θ on the variables in *positive position* in T.

Lemma 178 Let T be a type and θ, θ' two closed pattern substitutions. If for every variable α such that every free occurence of α in T is in a *positive position*, $\theta(\alpha) \ll \theta'(\alpha)$, and $\theta(\beta) = \theta'(\beta)$ for every other variable, then

$$[\![T]\!]_\theta \subseteq [\![T]\!]_{\theta'}$$

Conversely if $\theta(\alpha) \ll \theta'(\alpha)$ for every free variable α in a *negative* position, then

$$[\![T]\!]_{\theta'} \subseteq [\![T]\!]_\theta$$

Proof. First notice that if p is a pattern, then $p\theta \ll p\theta'$, by a simple induction on p. We prove both propositions simultaneously by induction on T:

- $T = \boldsymbol{B}(p)$. All variables of p appear positively in T. Then by the above remark, $p\theta \ll p\theta'$, and therefore $[\![\boldsymbol{B}(p)]\!]_\theta \subseteq [\![\boldsymbol{B}(p)]\!]_{\theta'}$.
- $T = T_1 \to T_2$. We treat the positive case. We have by induction hypothesis $[\![T_1]\!]_{\theta'} \subseteq [\![T_1]\!]_\theta$, as all variable of T_1 that appear positively in T appear negatively in T_1, and $[\![T_2]\!]_\theta \subseteq [\![T_2]\!]_{\theta'}$. Therefore, by definition of $[\![T_1 \to T_2]\!]_\phi$, we have:

  $$[\![T_1 \to T_2]\!]_\theta \subseteq [\![T_1 \to T_2]\!]_{\theta'}$$

  The negative case is treated in the same fashion.
- $T = \forall\alpha.U$: straightforward induction.

■

We note that these properties are exactly analogous to properties of the type interpretation usually used to prove correctness for size-based termination, see *e.g.* Blanqui [Bla04].

We can now prove the correctness of the interpretation relative to that of the function symbols (theorem 170).

Proof. We proceed by induction on the typing derivation.

- ax: by definition of $\sigma \models_\theta \Gamma$.
- t-lam: identical to the case in theorem 25.
- p-lam: by induction hypothesis, for all σ', θ' such that $\sigma' \models_{\theta'} \Gamma, |t|\sigma'$ is in $[\![T]\!]_{\theta'}$. Let σ, θ be some such valuations and P be a set of closed patterns. As $|\lambda\alpha.t|\sigma = |t|\sigma$, we need to show that $|t|\sigma \in [\![T]\!]_{\theta^\alpha_P}$

  Observe that if α does not appear in Γ, then $\sigma \models_\theta \Gamma$ implies $\sigma \models_{\theta^\alpha_P} \Gamma$, by virtue of lemma 174. We may therefore conclude that $|t|\sigma$ is in $[\![T]\!]_{\theta^\alpha_P}$.
- leaf-intro: Clear by definition of $[\![\boldsymbol{B}(\mathrm{leaf})]\!]_\theta$
- node-intro: let t, u be terms in $[\![\boldsymbol{B}(\alpha)]\!]_\theta$ and $[\![\boldsymbol{B}(\beta)]\!]_\theta$, respectively. The normal forms of Node $t\ u$ are of the form Node $t'\ u'$, with t' and u' normal forms of t and u, respectively. Therefore, to check if Node $t\ u \downarrow\ll \mathrm{node}(\theta(\alpha), \theta(\beta))$, it suffices to check $t \downarrow\ll \theta(\alpha)$ and $u \downarrow\ll \theta(\beta)$, both of which are true by hypothesis.
- t-app: straightforward by the induction hypothesis.

- p-app: by hypothesis, $|t|\sigma \in \llbracket \forall x.T \rrbracket_\theta$, this gives by definition $|t|\sigma \in \llbracket T \rrbracket_{\theta^x_{p\theta}}$, and by the substitution lemma (lemma 172), $|t|\sigma \in \llbracket T\{x \mapsto p\} \rrbracket_\theta$, therefore

$$|t\ \ p|\sigma \in \llbracket T\{x \mapsto p\} \rrbracket_\theta$$

- symb: By hypothesis.
- sub: By application of the correctness of subtyping (lemma 177), and the induction hypothesis.

■

3.2 Higher Order Dependency Chains

Now it remains to show that each function symbol is computable. By analogy with the first order dependency pair framework, we need to build an order on terms that is in relation to the approximated dependency graph. Then sequences of decreasing terms will be the analogue of *chains*, and we will show that there can be no infinite decreasing sequences. For this we wish to build a well-founded order on sequences of calls, which correspond to the dependency chains in the first-order case (definition 140). The definition in the first-order case can not be sufficient here, as the subterms of function calls do not contain sufficient information to determine termination (as in example 9). We therefore need to use the information given by our type system.

However it is somewhat subtle to build this order in practice: indeed, a natural candidate for such an order is (the transitive closure of) the relation defined by $(f, \vec{t}) > (g, \vec{u})$ if and only if

$$\exists \theta, \phi, f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_m) \in \mathcal{G},\ \forall i, j,\ t_i \in \llbracket \boldsymbol{B}(p_i) \rrbracket_\theta \wedge u_j \in \llbracket \boldsymbol{B}(q_j) \rrbracket_\phi$$

This would allow us to easily build the relation between the graph and the order, and show that each call induces a decrease in this order. Sadly, this order may not be well founded even in the event that the termination criterion is satisfied. Consider for example the rule $f\ \mathsf{node}(\alpha,\beta)\ (\mathsf{Node}\ \ x\ y) \to f\ \alpha\ x$, typeable in the context $\Gamma = x : \boldsymbol{B}(\alpha), y : \boldsymbol{B}(\beta)$. Given the above definition, we have $(f, t) > (f, u)$ provided that there are closed p and q such that $t \ll p$ and $u \ll q$. But then we may take $p = q = \top$ and if $t = z$ and $u = z$ with z a variable, then $(f, z) > (f, z)$. The rewrite system does satisfy the criterion, as $\mathsf{node}(\alpha,\beta) \rhd \alpha$, but the order is not well founded.

One possible solution is to restrict the reduction to call-by-value on closed terms, where a reduction in $\mathcal{R}$ can occur only if the arguments to the defined function are values in normal form (although β-reduction can occur at any moment). However we strive for more generality.

The solution we adopt here is to take, instead of just a particular instance of the pattern variables, the *most general* possible instance, and noticing that a rewrite step may occur only if a term matches a left-hand side.

Definition 179 Take the set $\mathcal{P}_{min}$ of *minimal patterns* to be the subset of $\mathcal{P}$ defined by:

$$p, q \in \mathcal{P}_{min} := \alpha \mid \mathsf{leaf} \mid \mathsf{node}(p, q)$$

Let v be a term in normal form. We inductively define the *pattern form* $pat(v)$ of v inductively:

- $pat(v) = \bot$ if t is neutral.

- $pat(\text{Leaf}) = \text{leaf}$
- $pat(\text{Node}\ \ v\ w) = \text{node}(pat(v), pat(w))$
- $pat(v)$ is undefined otherwise.

Note that if v is a normal form in $\mathcal{B}$, then $pat(v)$ is defined. If t is a term in $\mathcal{B}$, then $pats(t)$ is the set

$$pats(t) = \{pat(v) \mid v \text{ a normal form of } t\}$$

We define the partial *type matching* function $\mathsf{match}_{\mathcal{P}}$ that takes terms $t_1, \dots, t_n$ in $\mathcal{B}$, and minimal patterns $p_1, \dots, p_n$ in $\mathcal{P}_{min}$ and returns a pattern valuation:

- if $p_1 = \alpha_1, \dots, p_n = \alpha_n$ and $t_i = t_j$ whenever $\alpha_i = \alpha_j$, then
$$\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})(\alpha_i) = pats(t_i)$$
- if $p_i = \text{node}(q_1, q_2)$ and $t_i = \text{Node}\ \ u_1\ u_2$ then
$$\mathsf{match}_{\mathcal{P}}(\vec{t}; \vec{p}) = \mathsf{match}_{\mathcal{P}}(t_1, \dots, t_{i-1}, u_1, u_2, t_{i+1}, \dots, t_n\ ;\ p_1, \dots, p_{i-1}, q_1, q_2, p_{i+1}, \dots, p_n)$$
- if $p_i = \text{leaf}$ and $t_i = \text{Leaf}$ then
$$\mathsf{match}_{\mathcal{P}}(\vec{p}, \vec{t}) = \mathsf{match}_{\mathcal{P}}(t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n\ ;\ p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)$$
- $\mathsf{match}_{\mathcal{P}}$ is undefined in other cases.

The type matching can be seen as a way of giving the most precise possible valuation for terms that match some left-hand side of a rule. Notice that for each $f^\sharp(\vec{p}) \to g^\sharp(\vec{q}) \in \mathcal{G}$, each p_i is in $\mathcal{P}_{min}$. Indeed, an examination of the minimal typing rules show that only minimal patterns may appear in types.

Note also the reassuring fact that if $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p}) = \theta$, then for each i, $t_i \downarrow\ll p_i\theta$, by a simple induction.

Definition 180 A *link* is a tuple $(n, \vec{t}, \vec{u})$ such that

- $\vec{t}, \vec{u} \in \mathcal{B}$
- $n = f^\sharp(\vec{p}) \to g^\sharp(\vec{q}) \in \mathcal{G}$
- $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p}) = \theta$ is defined, and
$$\forall j, u_j \downarrow\ll q_j\theta'$$
For some extension θ' of θ such that $\mathcal{FV}(\vec{q}) \subseteq \mathsf{dom}(\theta')$.

A *chain* is an eventually infinite sequence $c_1, c_2, \dots$ of links such that if $c_i = (n_i, \vec{t_i}, \vec{u_i})$, then for each i,

$$\vec{u_i} \to^* \vec{t}_{i+1}$$

and if $n_i = f_i^\sharp(\vec{p}) \to g_i^\sharp(\vec{q})$ then $g_i = f_{i+1}$.

Notice that if $\mathcal{FV}(\vec{q}) \subseteq \mathcal{FV}(\vec{p})$ then we may take $\theta' = \theta$ in the definition of chains.

We first need to show a correspondence between the chains and the graph, that is:

Lemma 181 (Correctness of the approximated graph)
For each chain $c_1, c_2, \ldots$ such that $c_i = (n_i, \vec{t_i}, \vec{u_i})$, there is a path $n_1 \to n_2 \to \ldots$ in $\mathcal{G}_{\mathcal{R}}$.

Proof. It suffices to show that if $c_1 = (n_1, \vec{t}, \vec{u}), c_2 = (n_2, \vec{u}', \vec{v})$ is a chain, then there is an edge between $n_1 = f^\sharp(\vec{p}) \to g^\sharp(\vec{q})$ and $n_2 = g^\sharp(\vec{r}) \to h^\sharp(\vec{s})$. First note that the variables of $\vec{q}$ and $\vec{r}$ may be considered distinct. Notice that by definition of a link, there is a θ such that for each i, $u_i \downarrow\ll q_i\theta$, and furthermore $\mathsf{match}_{\mathcal{P}}(r_i, u'_i)$ is defined. As $u_i \to^* u'_i$, all normal forms of u'_i are also normal forms of u_i. We need to prove $\forall i,\ q_i \bowtie r_i$. We proceed by induction on $\mathsf{match}_{\mathcal{P}}(r_i, u'_i)$.

- r_i is a variable. We can conclude immediately by the definition of $\bowtie$, as a fresh variable can unify with any pattern.

- $u'_i = \mathsf{Leaf}$ and $r_i = \mathsf{leaf}$. In this case Leaf is a normal form of u_i, so there is some $q' \in q_i\theta$ such that $\mathsf{Leaf} \ll q'$. By examining the possible cases for q_i it follows that q_i is either leaf, $\top$ or some variable α. In each of those cases we can conclude that $q_i \bowtie \mathsf{leaf}$.

- $u'_i = \mathsf{Node}\ \ u'^1_i\ \ u'^2_i$ and $r_i = \mathsf{node}(r^1_i, r^2_i)$. Now let us examine q_i. We may exclude the cases $q_i = \mathsf{leaf}$ and $q_i = \bot$, as every normal form of u'_i is a normal form of u_i and is of the form $\mathsf{Node}\ \ v\ v'$. In the case $q_i = \alpha$ or $q_i = \top$ we may easily conclude. The only remaining case is $q_i = \mathsf{node}(q^1_i, q^2_i)$. From the induction hypothesis we get $q^1_i \bowtie r^1_i$ and $q^2_i \bowtie r^2_i$, which imply $q_i \bowtie r_i$

■

If the conditions of the termination theorem are satisfied, there are no infinite chains, in the same way as for the first order dependency pair approach.

Theorem 182 (Finiteness of chains)
Suppose that the conditions of the termination theorem of chapter 2 (theorem 165) are satisfied. Then there are no infinite chains.

We need to define and establish the well foundedness of the embedding order on terms.

Definition 183 We mutually define the strict and large *embedding preorder* on erased terms in normal form $\rhd$ and $\unrhd$ by:

- $t_1 \unrhd u \Rightarrow \mathsf{Node}\ \ t_1\ \ t_2 \rhd u$

- $t_2 \unrhd u \Rightarrow \mathsf{Node}\ \ t_1\ \ t_2 \rhd u$

- $t_1 \rhd u_1 \land t_2 \unrhd u_2 \Rightarrow \mathsf{Node}\ \ t_1\ \ t_2 \rhd \mathsf{Node}\ \ u_1\ \ u_2$

- $t_1 \unrhd u_1 \land t_2 \rhd u_2 \Rightarrow \mathsf{Node}\ \ t_1\ \ t_2 \rhd \mathsf{Node}\ \ u_1\ \ u_2$

- $\mathsf{Leaf} \unrhd \mathsf{Leaf}$

- $t \unrhd u$ if t and u are neutral.

- $t \rhd u \Rightarrow t \unrhd u$

Note that the preorder is *not* an order: for instance, $x \trianglerighteq y$ and $y \trianglerighteq x$. It can be seen as an order on the set in which all neutral terms are identified.

Lemma 184 The preorder $\triangleright$ is well-founded.

Proof. Given a term in normal form t, define $\mathsf{size}(t)$ inductively:

- $\mathsf{size}(\mathsf{Node}\ t_1\ t_2) = \mathsf{size}(t_1) + \mathsf{size}(t_2) + 1$
- $\mathsf{size}(t) = 0$ otherwise.

It is then easy to verify by mutual induction that if $t \triangleright u$, $\mathsf{size}(t) > \mathsf{size}(u)$ and if $t \trianglerighteq u$ then $\mathsf{size}(t) \geq \mathsf{size}(u)$. Well foundedness of the order on naturals yields the desired conclusion.

■

To show that there are no infinite chains, we will exploit the fact that if $c = (n, \vec{t}, \vec{u})$ is a link, that is decreasing in the embedding order on patterns, then there is a decrease in the normal forms from t to u.

To show this, we must prove that pattern-matching does indeed *completely* capture the "pattern semantics" of a term in $\mathcal{B}$.

Lemma 185 Suppose $\vec{t}$ are terms in $\mathcal{B}$ and $\vec{p}$ are minimal patterns. If $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined and equal to θ, then for each $q \in p_i\theta$, there is a normal form v of t_i such that $pat(v) = q$.

Proof. We proceed by induction on the definition of $\mathsf{match}_{\mathcal{P}}$:

- $p_i = \alpha_i$. In this case (as $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined) By definition $\alpha\theta$ is equal to $\{pat(v) \mid v \text{ is a normal form of } t_i\}$.
- $p_i = \mathsf{leaf}$. In this case $t_i = \mathsf{Leaf}$ and therefore we can take $v = \mathsf{Leaf}$.
- $p_i = \mathsf{node}(p_i^1, p_i^2)$. In this case, $t_i = \mathsf{Node}\ t_i^1\ t_i^2$. By the induction hypothesis, for any $q_1 \in p_i^1\theta$ and $q_2 \in p_i^2\theta$ there are normal forms v_1 and v_2 of t_i^1 and t_i^2 such that $q_j = pat(v_j)$ for $j = 1, 2$. It is easy to observe that $\mathsf{Node}\ v_1\ v_2$ is a normal form of t_i, and that $q = \mathsf{node}(q_1, q_2)$ is an element of $p_i\theta$, and $pat(\mathsf{Node}\ v_1\ v_2) = q$ allows us to conclude.

■

To prove that there are no infinite chains, we need to relate the decrease of the patterns to the decrease of the normal forms of the terms that appear in chains.

Lemma 186 (Correctness of the embedding order)
Suppose that p and q are closed patterns such that $p \triangleright q$ (respectively $p \trianglerighteq q$), and v_1, v_2 normal forms such that $pat(v_1) = p$ and $v_2 \ll q$. Then $v_1 \triangleright v_2$ (respectively $v_1 \trianglerighteq v_2$).

Proof. We prove both properties simultaneously by induction on the derivation of $p \triangleright q$:

- $p = \mathsf{node}(p_1, p_2)$ and $p_1 \trianglerighteq q$. We have $v_1 = \mathsf{Node}\ u_1\ u_2$ with $pat(u_1) = p_1$. By induction hypothesis $u_1 \trianglerighteq v_2$, and therefore $\mathsf{Node}\ u_1\ u_2 \triangleright v_2$.
- $p = \mathsf{node}(p_1, p_2), q = \mathsf{node}(q_1, q_2)$ with $p_1 \triangleright q_1$ and $p_2 \trianglerighteq q_2$. In that case $v_1 = \mathsf{Node}\ v_1^1\ v_1^2$ and $v_2 = \mathsf{Node}\ v_2^1\ v_2^2$. The induction hypothesis gives $v_1^1 \triangleright v_2^1$ and $v_1^2 \trianglerighteq v_2^2$, from which we may conclude.

- The symmetrical cases are treated in the same manner.
- $p = \mathsf{leaf}$ and $q = \mathsf{leaf}$. In this case, $v_1 = v_2 = \mathsf{Leaf}$, and $v_1 \trianglerighteq v_2$.
- $p = \bot$ and $q = \bot$. In this case both v_1 and v_2 are neutral, which gives $v_1 \trianglerighteq v_2$.

Lemma 187 (Chain decrease)
Let $c = (n, \vec{t}, \vec{u})$ be some link such that $n = f^\sharp(\vec{p}) \to g^\sharp(\vec{q})$. Suppose that there is i such that $p_i \rhd q_i$, (respectively $p_i \trianglerighteq q_i$). Then if v is a normal form of u_i, there exists some normal form v' of t_i such that $v' \rhd v$, (respectively $v' \trianglerighteq v$).

Proof. Let $\theta = \mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$, which is guaranteed to exist by hypothesis. First notice that for every $\alpha \in \mathcal{FV}(\vec{p})$, $\theta(\alpha)$ does *not* contain $\top$, by definition of $\mathsf{match}_{\mathcal{P}}$.

We treat the $\rhd$ case first. Suppose that v is a normal form of u_i. By definition, we have $u_i \downarrow\!\ll q_i\theta$, which means by definition that there is some $r \in q_i\theta$ such that $v \ll r$. Since $p_i \rhd q_i$, this implies that there is some $r' \in p_i\theta$ such that $r' \rhd r$. We have by lemma 185 that there exists some v' a normal form of t_i such that $pat(v') = r'$, which allows us to conclude using the previous lemma (lemma 186).

■

We finally have all the tools to give the proof of well foundedness of chains.

(Proof of theorem 182.
By contradiction, let $c_1, c_2, \ldots$ be an infinite chain, such that for each i, $c_i = (n_i, \vec{t_i}, \vec{u_i})$. By lemma 181, $n_1, n_2, \ldots$ is an infinite path in $\mathcal{G}$. By finiteness of $\mathcal{G}$, there is some SCC $\mathcal{G}'$ and some natural number k such that $n_k, n_{k+1}, \ldots$ is contained in $\mathcal{G}'$. By hypothesis, if $n_i = f_i^\sharp(\vec{p^i}) \to g_i^\sharp(\vec{q^i})$, there is an index j such that for each i, $p^i_j \trianglerighteq q^i_j$. Furthermore, again by hypothesis, there are an infinite number of indexes i such that $p^i_j \rhd q^i_j$. Let $V_i = \{v \mid v \text{ is a normal form of } t^i_j\}$ and $U_i = \{v \mid v \text{ is a normal form of } u^i_j\}$. We apply lemma 187 to show that for each $v'_i \in U_i$ there exists $v_i \in V_i$ such that $v_i \rhd v'_i$ for these indexes and $v_i \trianglerighteq v'_i$ for the others.

We wish to show that there is an infinite chain $v_1, v_2, \ldots$ such that $v_i \trianglerighteq v_{i+1}$ for each i and $v_i \rhd v_{i+1}$ for an infinite number of indexes i, contradicting well-foundedness of $\rhd$ (lemma 184).

To do this we first notice that $V_{i+1} \subseteq U_i$, as $\vec{u_i} \to^* \vec{t_{i+1}}$. Then we build the following graph:

- We have a node at the top, connected to every element of V_k.
- We have an edge between $a \in V_i$ and b in U_i if $a \trianglerighteq b$ or $a \rhd b$.
- We have an edge between $a \in U_i$ and $b \in V_{i+1}$ if $a = b$.

Notice first that every V_i, U_i is finite, as the rewrite system is finite (each strongly normalizing term therefore has a finite number of normal forms). We wish to apply *König's lemma* [Kön26] which states: every infinite connected graph with finite degree has an infinite simple path (a path without repeated nodes). It is easy to see that the graph is of finite degree: every V_i and U_i is finite and vertices are exclusively bewtween the V_i and U_i. We can verify that the graph is infinite, as no V_i or U_i is empty (the t_i and u_i are strongly normalizing and therefore have at least one normal form). Finally the graph is connected, as there is an edge between an element of U_i and an element of V_i for each i. This give us the existence of an infinite path in the tree, which concludes the proof.

■

We could alternatively use well-foundedness of the *multiset ordering* [DM79] to prove the non-existence of infinite chains. Note however that well-foundedness of the multiset order is also shown using König's lemma.

3.3 Correctness of Defined Function Symbols

To prove that the function symbols are in the interpretation of their type, we shall (obviously) need to consider the rewrite rules. In particular, we need to relate the minimal typing used to derive the types of left hand sides and pattern matching, in order to prove that our notion of chain is the correct one.

Lemma 188 (Correctness of type-matching)
Suppose that Γ is a context, that $l_1, \ldots, l_k$ are constructor terms and that $\Gamma \vdash_{min} l_1 : \boldsymbol{B}(p_1), \ldots, \Gamma \vdash_{min} l_k : \boldsymbol{B}(p_k)$. Suppose that $t_1, \ldots, t_k$ *match* $l_1, \ldots, l_k$. Then $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined.

Proof. We proceed by induction on the structures of l_i (matching the cases of the $\mathsf{match}_{\mathcal{P}}$ judgement)

- $l_1 = x_1, \ldots, l_n = x_n$. In this case, the only applicable case for $\vdash_{min}$ is the variable case. If $x_i = x_j$, then $t_i = t_j$. Furthermore $p_i = \alpha_i$ for some variable α_i and again, $\alpha_i = \alpha_j$ if and only if $x_i = x_j$, by linearity of α_i and α_j in Γ. Therefore if $\alpha_i = \alpha_j$, then $t_i = t_j$, and $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined.

- $l_i = \mathsf{Leaf}$. In this case the only applicable rule is the leaf rule, and $p_i = \mathsf{leaf}$ and $t_i = \mathsf{Leaf}$. By induction $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined.

- $l_i = \mathsf{Node}\ l_i^1\ l_i^2$. In this case we apply the node rule, and we have $p_i = \mathsf{node}(p_i^1, p_i^2)$. Again, we have $t_i = \mathsf{Node}\ t_i^1\ t_i^2$, and we may conclude by the induction hypothesis.

■

Our reason for defining pattern matching is to provide the "closest" possible pattern semantics for a term. In fact we have the following result, which states that any valuation θ such that t is in $[\![\boldsymbol{B}(\alpha)]\!]_\theta$ can be "factored through" $\mathsf{match}(t, p)$:

Lemma 189 Suppose that $\vec{t}$ is a tuple of strongly normalizing terms, that $\vec{\alpha}$ is a tuple of pattern variables, and θ' is a valuation that verifies:

$$\forall i, t_i \in [\![\boldsymbol{B}(\alpha_i)]\!]_{\theta'}$$

Suppose in addition that $\vec{p}$ are minimal patterns such that $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined and equal to θ. Let ϕ be the substitution that sends α_i to p_i. Then

$$\forall i, \theta \circ \phi(\alpha_i) \ll \theta'(\alpha_i)$$

Proof. We proceed by induction on the judgment $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$.

- $p_i = \beta_i$ for each p_i, and therefore $\phi(\alpha_i) = \beta_i$. In that case, $\theta \circ \phi(\alpha_i) = \{pat(v) \mid v \text{ normal form of } t_i\}$. Furthermore, $t_i \downarrow\ll \theta'(\alpha_i)$. Take some v a normal form of t_i. We have some $q \in \theta'(\alpha_i)$ such that $v \ll q$. We then verify that $pat(v) \ll q$, which implies $\theta \circ \phi(\alpha_i) \ll \theta'(\alpha_i)$

- $p_i = \mathsf{leaf}$. In this case, $\theta' \circ \phi(\alpha_i) = \mathsf{leaf}$. By $t_i \downarrow\ll \theta'(\alpha_i)$ and $t_i = \mathsf{Leaf}$, we have that $\theta'(\alpha_i)$ contains leaf or $\top$, and in each case we can conclude.
- $p_i = \mathsf{node}(p_i^1, p_i^2)$. In this case, $t_i = \mathsf{Node}\ t_1^1\ t_i^2$, and

$$\theta \circ \phi(\alpha_i) = \{\mathsf{node}(r_1, r_2) \mid r_1 \in p_i^1\theta \wedge r_1 \in p_i^2\theta\}$$

  By $t_i \downarrow\ll \theta'(\alpha_i)$ we have for each normal form v of t_i some q in $\theta'(\alpha_i)$ such that $v \ll q$. In addition v is of the form $\mathsf{Node}\ v_1\ v_2$, where v_1 is a normal form of t_i^1 and v_2 is a normal form of t_i^2. From this we get that either $q = \top$, in which case we are done, or $q = \mathsf{node}(q_1, q_2)$ with $v_1 \ll q_1$ and $v_2 \ll q_2$. In this case we apply the induction hypothesis to deduce that there is some $r_1 \in p_i^1\theta$ and $r_2 \in p_i^2\theta$ such that $r_1 \ll q_1$ and $r_2 \ll q_2$, and thus $\mathsf{node}(r_1, r_2) \ll \mathsf{node}(q_1, q_2)$.

■

Before proving the computability of the defined function symbols, we have to consider a last obstacle: the recursive calls are made to functions applied to all their pattern arguments, but not necessarily applied to their arguments themselves. To remedy this, we stratify the interpretation of the type of function symbols, and prove that if some conditions are satisfied, then the stratified interpretation is equal to the ordinary interpretation.

Definition 190 We define the following order $>_{dp}$ on pairs $(f, \vec{t})$ with $f \in \Sigma$ and $\vec{t}$ a tuple of terms:

$$(f, \vec{t}) >_{dp} (g, \vec{u}) \Leftrightarrow \exists \vec{v}, n = f^\sharp(\vec{p}) \to g^\sharp(\vec{q}),\ \vec{t} \to^* \vec{v} \wedge (n, \vec{v}, \vec{u}) \text{ is a link}$$

That is, if $\vec{t}$ reduces to $\vec{v}$ such that there is a link between $\vec{v}$ and $\vec{u}$, and where the associated node corresponds to a call from f to g.

Lemma 191 If the conditions of theorem 165 are satisfied then the order $>_{dp}$ is well-founded.

Proof. Any infinite decreasing sequence $(f_1, \vec{t_1}) > (f_1, \vec{t_2}) > \dots$ gives rise to an infinite chain, which is not possible by theorem 182.

We have enough to prove the main theorem, that is correctness of defined symbols.

Theorem 192 (Correctness of defined symbols)
Suppose that the conditions of theorem 165 are satisfied. Then for each $f \in \Sigma$ and each valuation θ, $f \in [\![\tau_f]\!]$.

Proof. Suppose that $\tau_f = \forall\vec{\alpha}.\boldsymbol{B}(\alpha_1) \to \dots \to \boldsymbol{B}(\alpha_k) \to T_f$. Take θ a valuation and $t_1, \dots, t_n$ in $[\![\boldsymbol{B}(\alpha_1)]\!]_\theta, \dots, [\![\boldsymbol{B}(\alpha_k)]\!]_\theta$. We need to show that

$$f\ t_1 \dots t_n \in [\![T_f]\!]_\theta$$

Note that each t_i is strongly normalizing. As $t = f\ \vec{t}$ is neutral, it suffices to show that every reduct of t is in $[\![T_f]\!]_\theta$.

We proceed as in the proof of correctness of the dependency pair criterion (theorem 141): it suffices to show that $u \in [\![T_f]\!]_\theta$ where u is a *head reduct* of $f\ t'_1 \dots t'_n$ and $t_i \to^* t'_i$. We show this by well-founded induction on $\vec{t}$ ordered by strict reduction: the reducts u of $f\ t_1 \dots t_n$ are either

- reducts of a head reduct t' of t: then $t' \in [\![T_f]\!]_\theta$ by hypothesis, and then $u \in [\![T_f]\!]_\theta$ by stability of reducibility candidates by reduction.

- reducts of $f\ t_1 \ldots t'_i \ldots t_n$ where $t_i \to^+ t'_i$. By hypothesis, every head reduct of $f\ u_1 \ldots u_n$ with $t_1 \to^* u_1 \ldots t'_i \to^* u_i \ldots t_n \to^* u_n$ is computable. By induction hypothesis this means that $f\ t_1 \ldots t'_i \ldots t_n$ is computable and therefore so is u.

So in this case every reduct of t is computable, and therefore so is t.

So now take $t_1 \to^* t'_1 \ldots t_n \to^* t'_n$, and suppose that $f\ t'_1 \ldots t'_n$ head rewrites to t'. We prove that t' is computable by well founded induction on $(f, \vec{t})$ ordered by $>_{dp}$. There is some rule $l \to r \in \mathcal{R}$, and some substitution σ such that $|l|\sigma = t$, and $|r|\sigma = t'$. By hypothesis that there is some context Γ and some derivation $\Gamma \vdash_{min} l_i : \boldsymbol{B}(p_i)$ for each i, and a derivation $\Gamma \vdash r : T_f\phi$, with ϕ the substitution that sends α_i to p_i.

By lemma 188, $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p}) = \psi$ is defined. We therefore have $t_i \in [\![\boldsymbol{B}(p_i)]\!]_\psi$ for each i, which gives $t_i \in [\![\boldsymbol{B}(\alpha_i)]\!]_{\psi\circ\phi}$ by the substitution lemma. By lemma 189, $\psi \circ \phi \ll \theta$. We may then apply the positivity condition of τ_f using lemma 178 to deduce that $[\![T_f]\!]_{\psi\circ\phi} \subseteq [\![T_f]\!]_\theta$. Therefore it suffices to show that t' is in $[\![T_f]\!]_{\psi\circ\phi}$, which is equal to $[\![T_f\phi]\!]_\psi$ by the substitution lemma. By hypothesis, $\Gamma \vdash r : T_f\phi$, so we would like to apply the correctness theorem 170 to show that $t' = |r|\sigma \in [\![T_f\phi]\!]_\psi$. The correctness theorem itself can not be applied, as it takes as hypothesis the correctness of function symbols, which we are trying to prove. But we will proceed in the same manner, making essential use of the well-founded induction hypothesis.

Let us first show that for each $x \in \mathsf{dom}(\sigma)$, $\sigma(x) \in [\![\Gamma(x)]\!]_\psi$. Suppose that i is such that x appears in l_i. We proceed by induction on the derivation of $\Gamma \vdash_{min} l_i$.

- **var**. In this case $l_i = x$. Then $\Gamma(x)$ is neccesarily equal to $\boldsymbol{B}(\gamma)$, and $\phi(\alpha_i) = p_i = \gamma$. Furthermore by definition of $\mathsf{match}_{\mathcal{P}}$, $\psi(\gamma) = pats(t_i)$ and so $\sigma(x) = t_i \in [\![\boldsymbol{B}(\gamma)]\!]_\psi$.

- **leaf**. In this case $l_i = \mathsf{Leaf}$. We have nothing to show here.

- **node**. In this case $l_i = \mathsf{Node}\ \ l^1\ l^2$. Simple application of the induction hypothesis.

Now we prove by induction on the derivation of $\Gamma \vdash r : T$ that $|r|\sigma \in [\![T]\!]_\psi$. We can exactly mimic the proof of theorem 170, except for the **symb** case. In this case, there is a g such that $r = g\ \vec{q}$, and if $\tau_g = \forall\vec{\beta}.\boldsymbol{B}(\beta_1) \to \ldots \to \boldsymbol{B}(\beta_m) \to T_g$, we need to show that, for some extension ψ' of ψ, $g \in [\![\boldsymbol{B}(q_1) \to \ldots \to \boldsymbol{B}(q_m) \to T_g]\!]_{\psi'}$. Recall the induction hypothesis on $(f, \vec{t})$, which states that for every θ, if $(f, \vec{t}) >_{dp} (g, \vec{u})$, then $g\vec{u} \in [\![T_g]\!]_\theta$. Now take θ to be $\psi' \circ \zeta$ where ζ is the substitution that sends β_i to q_i. It suffices to show that if for $i = 1, \ldots, m$ $u_i \in [\![\boldsymbol{B}(\beta_i)]\!]_{\psi'\circ\zeta}$, then $(f, \vec{t}) >_{dp} (g, \vec{u})$. For this we need to show that there exists $n \in \mathcal{G}$ such that:

- $n = f^\sharp(\vec{r}) \to g^\sharp(\vec{s})$

- $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{r}) = \theta$

- There is an extension θ' of θ such that

$$\vec{u} \downarrow\!\ll \sigma\theta'$$

We just take n to be the node that corresponds to the call site of $g\ \vec{q}$. In this case, $\vec{r} = \vec{p}$ and $\vec{s} = \vec{q}$. By definition, $\mathsf{match}_{\mathcal{P}}(\vec{t}, \vec{p})$ is defined and equal to ψ. Then ψ' is an extension of ψ and as $u_i \in [\![\boldsymbol{B}(\beta_i)]\!]_{\psi'\circ\zeta} = [\![\boldsymbol{B}(q_i)]\!]_{\psi'}$, we have $u_i \downarrow\!\ll q_i\psi'$.

■

Corollary 193 Every well-typed term is in the interpretation of its type, that is

$$\forall \Gamma, t, T \;\; \Gamma \vdash t : T \Rightarrow |t| \in [\![T]\!]$$

Where $[\![T]\!] = [\![T]\!]_\theta$ where θ is the valuation that sends every variable to the set $\{\top\}$.

Proof. In fact it does not matter which θ we choose: let θ be any valuation. Given a variable x and a type T, by lemma 168, $x \in [\![T]\!]_\theta$, as x is neutral and in normal form. Given $\Gamma \vdash t : T$, we can therefore take the substitution σ that sends every variable $x \in \mathsf{dom}(\Gamma)$ to itself. In that case $\sigma(x) \in [\![\Gamma(x)]\!]_\theta$ by the above remark, and by the combination of theorem 170 and theorem 192, $|t|\sigma \in [\![T]\!]_\theta$. But in this case $|t|\sigma = |t|$.

■

We obtain theorem 165 as a corollary: every well typed term is in the interpretation of its type, but this interpretation only contains strongly normalizing terms by lemma 168.

4

Related work

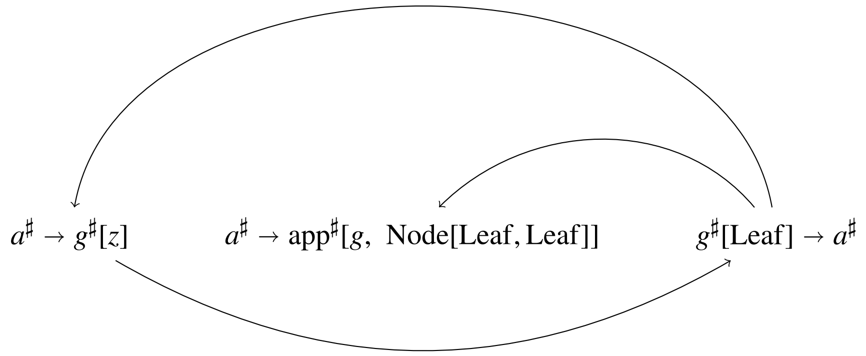
Our work on type-based dependency pairs follows the philosophy of size-based termination: we aim to define a type system that captures the semantics of the phenomena under consideration, and give a criterion that states that under certain conditions involving the typing of the rules, every well-typed terms are strongly normalizing. We have already discussed size-types in the previous part. We discuss other approaches to the extension of dependency pairs to higher-order rewriting.

These approaches can be loosely divided into two categories: the *dynamic* dependency pairs, which may include pairs of the form $f[\vec{t}] \to x[\vec{u}]$, as exposed in chapter 1 and *static* dependency pairs in which such rules can not appear. Our type based approach is of the latter flavor, as is usually the case with systems based on *computability*: the termination of $(x\ \vec{t})\sigma$ for a computable σ is trivial if every t_i is computable as well. The static approaches generally do not enjoy completeness: there are some rewrite systems which are terminating but for which there are infinite chains.

Work by Kusakari & Sakai [SK05] allows for the treatment of *applicative* rewrite systems, in which abstractions may not appear. It is however possible to take a system with abstractions and apply *defunctionalization* [DN01, Joh85] in order to obtain a system of the same expressivity, though termination of untyped terms of defunctionalized systems is not strictly equivalent to that of the original system in general. They also treat rules in which a variable may be applied in the left-hand side of a rule, which we forbid in our system. As for the other criteria described in this section, they allow matching on defined symbols, a feature which is absent from our approach (in these aspects, we are closer to functional languages in the tradition of ML [MTH91] or Haskell [HPJW+92]). The defunctionalized rewrite system for the example with app (example 17) is given by

$$\begin{array}{lcl}
\mathrm{app}[x,y] & \to & x[y] \\
f & \to & \mathrm{app}[g, \mathrm{Node}[\mathrm{Leaf}, \mathrm{Leaf}]] \\
g[\mathrm{Node}[x,y]] & \to & \mathrm{Leaf} \\
g[\mathrm{Leaf}] & \to & f
\end{array}$$

Which leads to the following graph for the static dependency pair framework (called *SC dependency pairs* in [KS07]):


$a^\sharp \to g^\sharp[z]$
$a^\sharp \to \mathrm{app}^\sharp[g,\ \mathrm{Node}[\mathrm{Leaf},\mathrm{Leaf}]]$
$g^\sharp[\mathrm{Leaf}] \to a^\sharp$

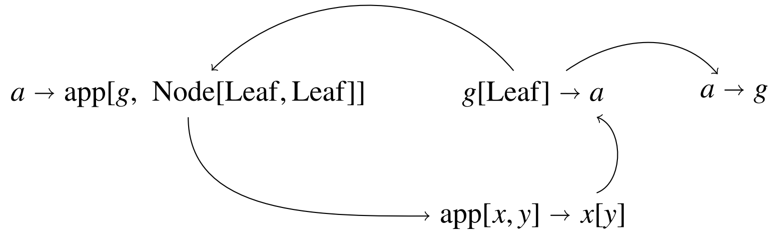
Unfortunately, this graph does have a cycle, and so the criterion can not be applied. Now it is possible to apply a certain number of transformation rules, for example *supercompilation* [Tur85] to try and find a semantically equivalent rewrite system for which the approximated dependency graph does not have any unresolvable cycles. In this case we would simply reduce the right hand side of the rule $a \to \mathrm{app}[g, \mathrm{Node}[\mathrm{Leaf}, \mathrm{Leaf}]]$ and turn it into the rule

$$a \to g[\mathrm{Node}[\mathrm{Leaf}, \mathrm{Leaf}]]$$

Which leads to an approximated dependency graph without cycles. However this analysis is in general much more complex than the simple *rencap* analysis described in chapter 1 (definition 145).

The static dependency pair frameworks described in this section all suffer from this problem. Blanqui *et al* [KISB09, Bla06] extend the approach to HRSes which allow abstraction in right-hand sides and left-hand sides under certain restrictions, and define the notion of *argument filterings* and *usable rules* for such systems.

Dynamic dependency pairs are considered by Kusakari *et al* [SK05, KS07], Aoto and Yamada [AY05] for applicative systems.Giesl, Thiemann and Schneider-Kamp [GTSK05], and Hirokawa, Middeldorp and Zankl *et al* [HMZ08] show that it is possible to adapt the dependency pair approach for first-order rewrite systems in order to treat applicative higher-order rewrite systems effectively. Given the above λ-free system, the approximated dynamic dependency graph is:


$a \to \mathrm{app}[g,\ \mathrm{Node}[\mathrm{Leaf},\mathrm{Leaf}]]$
$g[\mathrm{Leaf}] \to a$
$a \to g$
$\mathrm{app}[x, y] \to x[y]$

To prove that there are no infinite dependency chains, each of these approaches must therefore face the non-trivial task of treating the cycle in this graph.

Every criterion mentioned in this paper is capable of treating the example 18 given in chapter

2, as it is the applicative form of a relatively simple first-order rewrite system:

$$\begin{array}{rcl}
f(\mathrm{Node}(x,y)) & \rightarrow & g(i(\mathrm{Node}(x,y))) \\
g(\mathrm{Node}(x,y)) & \rightarrow & f(i(x)) \\
g(\mathrm{Leaf}) & \rightarrow & f(h(\mathrm{Leaf})) \\
i(\mathrm{Node}(x,y)) & \rightarrow & \mathrm{Node}(i(x),i(y)) \\
i(\mathrm{Leaf}) & \rightarrow & \mathrm{Leaf} \\
h(\mathrm{Node}(x,y)) & \rightarrow & h(x)
\end{array}$$

However termination can not be shown using simple projections, as

- for any term t, $i(t)$ is *not* a subterm of t.
- semantic information on the function h needs to be given.

Our type system allows us to express the simple facts that $(i\ t)$ *behaves* like the term t with respect to term structure and that $(h\ t)$ may *never* reduce to a constructor. this allows a simple syntactic analysis at type level to determine termination. Again, it could be argued that a non-trivial semantic analysis needs to be performed to be able to infer the types of the i and g functions. In a sense we displace a termination problem to a type inference problem, though of course many techniques are similar in both fields.

Conclusion

Higher-order labelling and size-based termination

We have shown that the termination of a version of size-based termination by type-level annotations could be captured by the combination of:

- A higher-order version of semantic labelling.
- A realizability-model, which takes terms of base type to a set-theoretic representation of inductive types, and terms of function type to *realized* set-theoretical functions.
- A termination criterion, that allows us to show relative termination of the labelled system.

This approach leads to a general semantic termination criterion: first find a model that captures the semantic properties of the system, and apply the termination criterion to the labelled system. To apply the precedence criterion, we need to find a compatible precedence. We show that this is not always trivial to do, and in the case of the size semantics and the syntactic model of theorem 137, we show that a certain combinatory lemma allows us to show the well-foundedness of the precedence, which is built by combination of a "natural" order, and an order which ensures compatibility. We believe that this lemma can be applied in a systematic way, and it would be nice to find a direct combinatory characterization of our termination, which we conjecture to be close to that of Whalstedt [Wah07].

While the categorical semantics have not yet, to our knowledge, been extended to treat dependent or polymorphic types, it seems that such developments are not far away. At any rate, our approach can be instantiated to a particular class of models, and as such can be straightforwardly generalized to more powerful type theories. In particular we believe that it is possible to give the fundamental theorem of semantic labelling for the system F [Gir72, Rey74] with rewriting, in which we restrict the models to realizability models in the sense of Λ-*sets* from Altenkirch [Alt93]. As our termination criterion is based on a simplification of the General Schema applied to rewriting modulo equivalence rules [Bla03], it can readily be generalized to system F or even the Calculus of Algebraic Constructions. This would give a general account of the idea of "separation of concerns", which aims to separate the computability arguments used to handle β-reductions and the combinatory or semantic argument used to handle the left-algebraic rules. In this sense our result is a *modularity* result.

Another natural question is the expressivity of this criterion. As noted in example 9, the criterion is not complete. Another example that can not be handled by the labelling approach is the system consisting of the rule:

$$f\ (S\ \ x) \to (\lambda y.y\ x)f$$

In both these situations, the β-expansion "hides" information from the argument on which the recursive call is performed. One advantage of the traditional presentation of type-based termination, which is absent in our framework, is that such information can be encoded in the types. This is the motivation of the type-based dependency pair approach described in chapter 2. One may wonder therefore, if there is a restriction of higher-order rewriting with left-algebraic rules for which the labelling criterion is complete. We have shown in section 5.3 of chapter 5 that this is the case for the combination of a (simply sorted) first order rewrite system and currified rewriting with β-reduction. We furthermore conjecture that this is the case for all rewrite system for which the right-hand sides of rules satisfy the following constraints:

- All function symbols are fully applied.

- No bound variables appear in the arguments of defined symbols.

Note however, that the semantic-labelling approach is capable of handling strictly more powerful systems, as the recursors for Brouwer ordinals (example 3) do not satisfy this condition, yet can be shown to be terminating using size-based termination.

It is also conceivable to replace our precedence criterion with a more powerful version, for example an adaptation of the *Higher Order Recursive Path Ordering* of Jouannaud & Rubio (see *e.g.* [BJR08]) to the case of relative normalization. Though it is unlikely that we obtain a theoretically more powerful criterion, it may be the case that the combination of semantic labelling and HORPO leads to a criterion that is easier to apply on specific examples, as a premodel which leads to a labelled system that does not pass the precedence criterion may possibly be treated by HORPO. This occurs frequently in the first-order case, see *e.g.* Zantema [BKdV03] (page 248-249 example 6.5.42.).

We believe that our restriction to orthogonal rewrite systems in the criterion of chapter 1 can be raised, either by taking the more general constraint of *confluence*, or by giving a more permissive model in which terms may be interpreted by *sets of atoms* in the semantic world.

Finally we wish to generalize these results to richer notions of rewriting. Hamana presents his criterion for CRSs, which include a simple form of matching on terms with bound variables. The absence of this constraint has simplified the presentation of our labelling framework, but it is likely that we may generalize the correctness result to the richer notion.

Type based dependency pairs

We have described a type system which allows us to build a *type-level* dependency graph for constructor-based left-algebraic rewrite systems, and we show that if the *simple projection* criterion of Hirokawa & Middeldorp holds for this graph, then the system is terminating.

Much can be said about possible generalizations of the type-based dependency pair criterion. It is clear that the system may be generalized to any first-order datatype, like the Peano natural numbers or (monomorphic) lists. It is less clear that this criterion is able to treat higher-order datatypes like the Brouwer ordinals of example 3. However we believe that these types can be treated, using simply a suitable modification of the proof of well-foundedness of $\rhd$ on normal forms (lemma 184). In this case however, considering the set of reducts of terms of base type is not sufficient. It is also necessary to consider reducts of functional arguments applied to terms. Then the finite sets considered in the proof of well-foundedness of chains (theorem 182) become infinite, and we may not easily apply König's lemma *as is.*

Our criterion only treats left-algebraic rules in which matching may not occur on defined symbols. In addition, it is difficult to give an interesting return type for most functions as the shape of the output is not generally *uniform* in the input. As an example consider the function defined by the rules (given without the type annotations):

$$\begin{array}{lll} f\ \mathrm{Leaf} & \rightarrow & \mathrm{Node\ Leaf\ Leaf} \\ f\ \mathrm{Node}\ x\ y & \rightarrow & \mathrm{Leaf} \end{array}$$

In our framework, the only possible type we can give to this function is $f : \forall\alpha.\boldsymbol{B}(\alpha) \rightarrow \boldsymbol{B}(\top)$. Indeed for every annotation $p(\alpha)$ such that $f : \forall\alpha.\boldsymbol{B}(\alpha) \rightarrow \boldsymbol{B}(p(\alpha))$ is valid, we have $\mathrm{node}(\mathrm{leaf}, \mathrm{leaf}) \ll p(\mathrm{leaf})$ and $\mathrm{leaf} \ll p(\mathrm{node})$, but this is the case only for $p(\alpha) = \top$. A possibility is to introduce a *type-level* function $\hat{f}$, with the conversion rules $\hat{f}(\mathrm{leaf}) \simeq \mathrm{node}(\mathrm{leaf}, \mathrm{leaf})$ and $\hat{f}(\mathrm{node}(p, q)) \simeq \mathrm{leaf}$. We then carry such a conversion to the types, and give to f the type

$$\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\hat{f}(\alpha))$$

This approach gives much more power to the type system, and allows extending the framework to matching upon defined symbols. Such rules would make computing the dependency graph more difficult of course, but it is possible to adopt the approximation given in definition 145 using the *cap* function. A more subtle problem is then giving a suitable generalization of the $\trianglerighteq$ order on type annotations that allows us to generalize the termination theorem (theorem 165). We conjecture that, if the conversion rule $\simeq$ is oriented left-to-right, and considered as a first-order rewrite system, then a simplification order such as $>_{rpo}$ may suffice. This order would then have to weakly orient every conversion rule and use $>_{rpo}$ instead of $\triangleright$ to treat every cycle in the graph. This leads to a "type level first order system" and an interesting research question is the following: is proving termination of a higher-order rewrite system using refinement types and type level conversion equivalent to applying defunctionalization to the system and applying first-order termination techniques, such as described in Giesl *et al* [GTSK05]?

If we do not have conversion at the type level, we can still add expressivity to our framework by adding *union types*. To do this we authorize atomic types to be of the form $\boldsymbol{B}(p_1)\cup\ldots\cup\boldsymbol{B}(p_n)$. This extension is very natural as our semantics already account for instantiation of type variables by sets of patterns. This is less liberal than the unions considered in Freeman and Pfenning [FP91b], who may consider unions of arbitrary types at the cost of a considerable loss of simplicity in type-checking. With such an extension, we would be able to give the following type to the function f defined above:

$$f : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\mathrm{leaf}) \cup \boldsymbol{B}(\mathrm{node}(\mathrm{leaf}, \mathrm{leaf}))$$

We believe that the semantics defined in chapter 3 is sufficient to prove termination of such an extended system, though the semantics of type-level variable substitution is considerably more complex.

As for our semantic labelling approach, we believe that the type-based dependency pair approach can be generalized to systems with polymorphic or dependent types, such as the Calculus of Inductive Constructions, as has already been done for type-based termination [Bla04, Abe04, BGP06].

Our framework uses explicit type information in the abstraction of terms, and marks pattern abstractions and applications, in order to make the type inference *syntax directed*. In this way it is possible to give an alternative type system in which subtyping is pushed into the application rule and types are *synthesized* rather than checked. However, the user does not wish to give all this explicit type information. It is likely that the type information can be recovered from a system without the annotations on types and explicit abstraction and application, by using a method similar to Blanqui [Bla05a]. Indeed, carrying out the synthesis described above without these explicit annotations results in a number of *constraints* on patterns (much in the same way as for Damas-Milner type inference [DM82], see also Pierce [Pie02]). We conjecture that finding solutions to these constraints is decidable.

A further possible development, which would make the criterion fully automatic, is the inference of the type annotations for defined functions themselves. Note that it is always possible to give a "trivial" annotation to functions, namely by taking the simple type of the function, and to each atomic type in *positive* position, associating the atomic type $\boldsymbol{B}(\top)$ and to each atomic type in *negative* position, associating $\boldsymbol{B}(\alpha)$ for some fresh variable α, and then quantifying over all variables. For instance to a symbol f of simple type $B \to ((B \to B) \to B) \to B$ we would associate:

$$f : \forall \alpha\beta\gamma.\boldsymbol{B}(\alpha) \to ((\boldsymbol{B}(\beta) \to \boldsymbol{B}(\top)) \to \boldsymbol{B}(\gamma)) \to \boldsymbol{B}(\top)$$

This easily ensures well-typedness of the rules, but much information is lost, though one should note that the criterion does not become trivial. In particular, the dependency graph of example 17 *does* contain a cycle if such an approximation is used. The inference of more informative annotations seems to be a non-trivial problem. The closest to our case seems to be the work of Barthe, Gregoire and Pastawski [BGP09] in which they describe such an inference system for a size-type system for the Calculus of Inductive Constructions, and prove a form of completeness of their inference. In their framework, the inputs on which the size of the output depends must be marked. Let us mention also work by Chin and Khoo [CK01], in which they use a form of abstract interpretation refinement to compute size annotations for functions in the language of Presburger arithmetic. Let us mention in passing work on program transformation by *deforestation* (see *e.g.* Hamilton [Ham06]) which appears to apply semantic techniques very similar to those necessary for inference of annotations. Finally, work by Rondon *et al* [RKJ08, KRJ10], dubbed *liquid types* describes inference for different variants of refinement types, and may very well apply to our case with minor variations.

Our criterion is completely orthogonal to the *size change principle* described by Lee, Jones and Ben-Amram [LJBA01]. Indeed the SCP operates simply on the control graph of the rewrite system (or the program), showing, in analogy with the theorem on finiteness of chains (theorem 182), that a certain sequence of calls may not occur an infinite number of times. We therefore conjecture that this analysis could easily be integrated into our criterion. This is comforted by the fact that the SCP has been successfully integrated into more complex analyses on sequences of calls, see for example Codish *et al* [CFGSK10] and Hyvernat [Hyv10].

Other research directions

It would not be difficult to build a premodel analogous to that of chapter 4, which sends terms of base type to finite sets of patterns. However it is not possible to use semantic labelling to prove the correctness of the normalization theorem for the type level dependency pairs because, as we have noted above, semantic labelling does not capture the "non-local data flow" which is captured by the type systems used in the traditional size-based termination or our refinement type framework. One may wonder how to combine the powerful semantics of the premodel approach with the data-flow analysis provided by typing.

A natural approach is, given a presheaf algebra M for our terms, to refine the interpretation $M_T(\Gamma)$ in order to be able to express *constraints* in addition to *guarantees*: Given a term t typed by T in the context Γ, $(\!|\Gamma \vdash t : T|\!) = o$ expresses the *guarantee* that the semantics of t is o. However we wish to express the fact that certain function applications may never occur: a function is *constrained* to only be applied to certain arguments. This is (one of the) fundamental insights of type-based termination. If we restrict ourselves to models in which the sheaf $M_{A\to B}(\Gamma)$ is a certain set of *functions* from $[\![A]\!]$ to $[\![B]\!]$ (as is the case for our size-premodel), then constraints can be ensured by restricting the domain of these functions to a subset $\{v \mid v \leq k\} \subseteq [\![A]\!]$. If we wish to apply semantic labelling to prove termination, then the correctness of the interpretations of subterms must then be shown incrementally, in a similar manner to the derivation of a size-type. We therefore conjecture the existence of a *semantic typing system* for termination, which may very well be complete for left-algebraic simply typed and positive rewrite systems. This can certainly be seen as the most ambitious open research direction for this thesis. Berger [Ber05], following Coquand and Spiwak [CS06], describes a domain which allows interpreting λ-terms, and in which the interpretation is different than $\perp$ if and only if the term is strongly normalizing under rewrite rules combined with β-reduction, which may provide a clue as to how to construct such a semantics-based type system.

Bibliography

[Abe99] Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 1956 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1999.

[Abe04] Andreas Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.

[Abe06] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[ACG98] Roberto Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. In M. Nivat, editor, *Proceedings of FOSSACS'98*, volume 1378, pages 48–62, 1998.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

[Alt93] Thorsten Altenkirch. Proving strong normalization of cc by modifying realizability semantics. In Barendregt and Nipkow [BN94], pages 3–18.

[Alt94] Thorsten Altenkirch. *Constructions, inductive types and strong normalisation.* PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1994.

[AvOS10] Claus Appel, Vincent van Oostrom, and Jakob Grue Simonsen. Higher-order (non-)modularity. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 17–32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[AY05] Takahito Aoto and Toshiyuki Yamada. Dependency pairs for simply typed term rewriting. In Giesl [Gie05], pages 120–134.

[Bar84] Henk Barendregt. *Lambda Calculus: its Syntax and Semantics.* North Holland, 1984.

[Ber05] Ulrich Berger. Continuous semantics for strong normalization. In S.B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE 2005: New Computational Paradigms*, volume 3526 of *Lecture Notes in Computer Science*, pages 23–34. Springer-Verlag, 2005.

[BFG97] Franco Barbanera, Maribel Fernández, and Hermann Geuvers. Modularity of strong normalisation and confluence in the algebraic λ-cube. *Journal of Functionnal Programming*, 7(6):613–660, November 1997.

[BFG$^+$04] Gilles Barthe, Maria João Frade, Eduardo Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[BGP06] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Type-based termination of recursive definitions in the calculus of inductive constructions. In *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, November 2006. To appear.

[BGP09] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for typed-based termination in a polymorphic setting. In *TLCA*, pages 71–85, 2009.

[BJO97] Frédéric Blanqui, Jean-Pierre Jouannaud, and M. Okada. Abstract data type systems. *Theoretical Computer Science*, pages 349–391, 1997.

[BJO99] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The algebraic calculus of constructions. In P. Narendran and M. Rusinowitch, editors, *Proceedings of RTA'99*, volume 1631, pages 301–316, 1999.

[BJO02] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive-data-type Systems. *Theoretical Computer Science*, 272:41–68, 2002.

[BJR08] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering: the end of a quest. In *7th EACSL Annual Conference on Computer Science Logic - CSL'08*, volume 5213 of *LNCS*, Bertinoro Italie, 2008.

[BKdV03] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[Bla01] Frédéric Blanqui. *Théorie des Types et Récriture*. PhD thesis, Université Paris XI, Orsay, France, 2001. Available in english as "Type theory and Rewriting".

[Bla03] Frédéric Blanqui. Rewriting modulo in deduction modulo. In Robert Nieuwenhuis, editor, *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2003.

[Bla04] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proc. of the 15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, 2004.

[Bla05a] Frédéric Blanqui. Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. In *Proceedings of the 19th International Conference on Computer Science Logic*, Lecture Notes in Computer Science 3634, 2005.

[Bla05b] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[Bla06] Frédéric Blanqui. Higher-order dependency pairs. In *Proceedings of the 8th International Workshop on Termination*, 2006.

[BN94] Henk Barendregt and Tobias Nipkow, editors. *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*. Springer, 1994.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BR06] Frédéric Blanqui and Colin Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 4246, 2006.

[BR09] Frédéric Blanqui and Cody Roux. On the relation between sized-types based termination and semantic labelling. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2009.

[CFGSK10] Michael Codish, Carsten Fuhs, Jürgen Giesl, and Peter Schneider-Kamp. Lazy abstraction for size-change termination. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2010.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CHS80] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *To H.B. Curry: essays on combinatory logic, lambda calculus, and formalism.* Academic Press, 1980.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:414–432, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[CL03] René Cori and Daniel Lascar. *Logique mathématique, tome 2 : Fonctions récursives, théorème de Gödel, théorie des ensembles.* Dunod, January 2003.

[CM98] Guy Cousineau and Michel Mauny. *The functional approach to programming.* Cambridge University Press, 1998.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal proceedings of Logical Frameworks'92*, pages 66–79, 1992.

[Coq93] Thierry Coquand. Infinite objects in type theory. In Barendregt and Nipkow [BN94], pages 62–78.

[Coq08] Coq Development Team. *The Coq Reference Manual, Version 8.2.* INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`.

[CS06] Thierry Coquand and Arnaud Spiwack. A proof of strong normalisation using domain theory. In *LICS*, pages 307–316. IEEE Computer Society, 2006.

[Dav85] Martin Davis. *Computability and Unsolvability (Mcgraw-Hill Series in Information Processing and Computers.)*. Dover Publications, December 1985.

[dB91] Nicolaas Govert de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91:189–204, April 1991.

[dB95] Nicolaas Govert de Bruijn. On the roles of types in mathematics. In de Groote, Philippe, editor, *The Curry-Howard isomorphism*, volume 8 of *Cahiers du centre de logique*, pages 27–54. Universite catholique de Louvain, 1995.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal models and semantics. Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[DK98] Henk Doornbos and Burghard Von Karger. On the union of well-founded relations, 1998.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22:465–476, August 1979.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of POPL'82*, pages 207–212. ACM Press, 1982.

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *proceedings of PPDP*, pages 162–174. ACM, 2001.

[Fax02] Karl-Filip Faxén. A static semantics for haskell. *J. Funct. Program.*, 12:295–357, July 2002.

[FBHL73] Abraham A. Fraenkel, Yehoshua Bar-Hillel, and Azriel Lévy. *Foundations of set theory*. Studies in logic and the foundations of mathematics. North-Holland, 1973.

[Fer93] Maribel Fernández. *Modèles de calcul multiparadigmes fondés sur la réécriture.* PhD thesis, Université Paris-Sud Orsay, 1993.

[FP91a] Tim Freeman and Frank Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, 1991.

[FP91b] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of PLDI'91*, pages 268–277. ACM Press, 1991.

[FPT99] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *LICS*, pages 193–202, 1999.

[Fra03] Maria João Frade. *Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi.* PhD thesis, Universidade do Minho, 2003.

[Gab07] Murdoch J. Gabbay. Fresh logic: proof-theory and semantics for fm and nominal techniques. *Journal of Applied Logic*, 5(2):356 – 387, 2007. Logic-Based Agent Verification.

[Gal90] Jean Gallier. On Girard's "candidats de réducibilité". In P. Odifreddi, editor, *Logic and Computer Science*, pages 123–203. Academic Press, 1990.

[GAO02] Jürgen Giesl, Thomas Arts, and Enno Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *J. Symb. Comput.*, 34:21–58, July 2002.

[GBT89] Jean Gallier and Val Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *Proc. of ICALP*, volume 372 of *LNCS*, pages 137–150. Springer-Verlag, 1989.

[Ges90] Alfons Geser. *Relative Termination.* PhD thesis, University of Passau, Germany, 1990.

[Gie05] Jürgen Giesl, editor. *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science.* Springer, 2005.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin / Heidelberg, 1995.

[Gim96] Eduardo Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants.* PhD thesis, Ecole Normale Superieure de Lyon, 1996.

[Gim98] Eduardo Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443, pages 397–408, 1998.

[Gir72] Jean-Yves Girard. *Interpretation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur.* PhD thesis, Université Paris VII, 1972.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Number 7 in Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[GSkT06] Jürgen Giesl, Peter Schneider-kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *In Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006*, pages 281–286. Springer-Verlag, 2006.

[GTSK04] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2004.

[GTSK05] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *proceedings of the 5th FROCOS conference*, pages 216–231. Springer, 2005.

[Ham98] Makoto Hamana. *Semantics for Interactive Higher-order Functional-logic Programming.* PhD thesis, University of Tsukuba, Japan, 1998.

[Ham03] Makoto Hamana. Term rewriting with variable binding: an initial algebra approach. In *PPDP*, pages 148–159. ACM, 2003.

[Ham05] Makoto Hamana. Universal algebra for termination of higher-order rewriting. In Giesl [Gie05], pages 135–149.

[Ham06] Geoff W. Hamilton. Higher order deforestation. *Fundam. Inform.*, 69(1-2):39–61, 2006.

[Ham07] Makoto Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2007.

[Har15] Fritz Hartogs. Über das problem der wohlordnung. *Mathematische Annalen*, 76:438–443, 1915. 10.1007/BF01458215.

[HM06] Nao Hirokawa and Aart Middeldorp. Predictive labeling. In Frank Pfenning, editor, *Term Rewriting and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 313–327. Springer Berlin / Heidelberg, 2006.

[HM07a] Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool: Techniques and features. *Inf. Comput.*, 205(4):474–511, 2007.

[HM07b] Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool: Techniques and features. *Inf. Comput.*, 205:474–511, April 2007.

[HMZ08] Nao Hirokawa, Aart Middeldorp, and Harald Zankl. Uncurrying for termination. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 667–681. Springer, 2008.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 204–, Washington, DC, USA, 1999. IEEE Computer Society.

[HPJW$^+$92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Arvind, Boutel Jon Fairbairn, Joseph Fasel, Marìa Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the functional programming language Haskell, version 1.2. *Special Issue of SIGPLAN Notices*, 27, 1992.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of POPL'96*, pages 410–423. ACM Press, 1996.

[Hyv10] Pierre Hyvernat. The size-change termination principle for constructor based languages. http://www.lama.univ-savoie.fr/ hyvernat/research.php, 2010.

[Jec06] Thomas Jech. *Set Theory.* Monographs in Mathematics. Springer, 2006.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. A computational model for executable higher-order algebraic specification languages. In *Proceedings of the sixth annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 350–361, 1991.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985.

[Kah95] Stefan Kahrs. Towards a domain theory for termination proofs. In Jieh Hsiang, editor, *RTA*, volume 914 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 1995.

[KISB09] Keiichirou Kusakari, Yasuo Isogai, Masahiko Sakai, and Frédéric Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE TRANSACTIONS on Information and Systems Vol.E92-D No.10*, pages 2007–2015, 2009.

[Kle45] Stephen Cole Kleene. On the interpretation of intuitionistic number theory. *J. Symb. Log.*, 10(4):109–124, 1945.

[KNT99] Keiichirou Kusakari, Masaki Nakamura, and Yoshihito Toyama. Argument filtering transformation. In Gopalan Nadathur, editor, *PPDP*, volume 1702 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 1999.

[Kön26] Dénes König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.

[KRJ10] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. Dsolve: Safety verification via liquid types. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 123–126. Springer, 2010.

[KS07] Keiichirou Kusakari and Masahiko Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *Appl. Algebra Eng. Commun. Comput.*, 18(5):407–431, 2007.

[KvOvR93] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993.

[KZ06] Adam Koprowski and Hans Zantema. Automation of recursive path ordering for infinite labelled rewrite systems. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2006.

[Lan71] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, Berlin, 1971.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL'01*, pages 81–92. ACM Press, 2001.

[Mat98] Ralph Mattes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-Universität München, 1998.

[McB04] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[McK06] James McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

[Men87] Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, USA, 1987.

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, 1991.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MM04] Conor McBride and James McKinna. Functional pearl: i am not a number–i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA, 2004. ACM.

[MN70] Zohar Manna and Stephen Ness. On the termination of Markov algorithms. In *Proceedings of the 3rd Hawaii International Conference on System Sciences*, 1970.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. Technical report, Duke University, Durham, NC, USA, 1987.

[MTH91] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[MW98] Paul-André Melliès and Benjamin Werner. A generic proof of strong normalisation for pure type systems. In E. Giménez and C. Paulin-Mohring, editors, *Proceedings of TYPES'96*, volume 1512, 1998.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In *LICS*, pages 342–349. IEEE Computer Society, 1991.

[Nor09] Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *TLDI*, pages 1–2. ACM, 2009.

[Oka89] Mitsuhiro Okada. Strong normalizability for the combined system of the typed λ-calculus and an arbitrary convergent term rewrite system. In *Proc. of ISSAC*, pages 357–363. ACM Press, 1989.

[OMG00] Hitoshi Ohsaki, Aart Middeldorp, and Jürgen Giesl. Equational termination by semantic labelling. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 457–471, London, UK, 2000. Springer-Verlag.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165 – 193, 2003. Theoretical Aspects of Computer Software (TACS 2001).

[Plo75] Gordon Plotkin. Call by name, call by value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PM96] Christine Paulin-Mohring. *Définitions Inductives en Theorie des Types d'Ordre Superieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.

[Pos47] Emil L. Post. Recursive unsolvability of a problem of thue. *J. Symb. Log.*, 12(1):1–11, 1947.

[Pre29] Mojzesz Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101. Casopis Prague, 1929.

[Rey74] John Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Proceedings of the Programming Symposium*, volume 19, pages 408–425, 1974.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

[SK05] Masahiko Sakai and Keiichirou Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.

[Ste78] Guy L. Steele, Jr. Rabbit: A compiler for scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.

[Tur36] Alan M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[Tur85] Valentin F. Turchin. Program transformation by supercompilation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 1985.

[vdP93] Jaco van de Pol. Termination proofs for higher-order rewrite systems. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *HOA*, volume 816 of *Lecture Notes in Computer Science*, pages 305–325. Springer, 1993.

[vdP96] Jaco van de Pol. *Termination of Higher-Order Rewrite Systems*. PhD thesis, Utrecht University, 1996.

[vdPS95] Jaco van de Pol and Helmut Schwichtenberg. Strict functionals for termination proofs. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 1995.

[Ven00] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types.* PhD thesis, University of Tartu, Estonia, 2000.

[vO94] Vincent van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, 1994. Available at `http://www.cs.vu.nl/~oostrom`.

[vO02] Jaap van Oosten. Realizability: a historical essay. *Mathematical. Structures in Comp. Sci.*, 12:239–263, June 2002.

[vR96] Femke van Raamsdonk. *Confluence and normalisation for higher-order rewriting.* PhD thesis, Vrije Universiteit, 1996.

[Wah07] David Wahlstedt. *Dependent Type Theory with Parametrized First-Order Data Types and Well-Founded Recursion.* PhD thesis, Chalmers University of Technology, 2007.

[Wel94] John B. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of LICS'94*, pages 176–185. IEEE Computer Society Press, 1994.

[Wer94] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.

[Xi01] Hongwei Xi. Dependent Types for Program Termination Verification. In *Proceedings of LICS'01*, pages 231–242. IEEE Computer Society Press, 2001.

[Zan95] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

[Zan04a] Hans Zantema. Relative termination in term rewriting. In Michael Codish and Aart Middeldorp, editors, *Proceedings of the 7th International Workshop on Termination*, pages 51–55. RWTH Aachen, 2004.

[Zan04b] Hans Zantema. Torpa: Termination of rewriting proved automatically. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 95–104. Springer, 2004.

[Zsi10] Julianna Zsido. *Typed Abstract Syntax.* PhD thesis, Université de Nice Sophia-Antipolis, 2010.

Résumé

Ce manuscrit présente une réflexion sur la terminaison des systèmes de réécriture d'ordres supérieurs. Nous nous concentrons sur une méthode particulière, la terminaison à base de tailles. La terminaison à base de tailles utilise le typage pour donner une approximation syntaxique à la taille d'un élément du langage.

Notre contribution est double: premièrement, nous permettons d'aborder de manière structurée le problème de la correction des approches à base de taille. Pour ce faire, nous montrons qu'elle peut être traitée par une version de la méthode des annotations sémantiques. Cette dernière utilise des annotations sur les termes calculés à partir de la sémantique des sous-termes dans un certain prémodèle équationnel. Nous montrons la correction de notre approche par annotations sémantiques, ainsi que du critère qui permet de traiter le système annoté, et nous construisons un prémodèle pour le système qui correspond intuitivement à la sémantique du système de réécriture. Nous montrons alors que le système annoté passe le critère de terminaison.

D'un autre coté nous modifions l'approche classique de la terminaison a base de tailles et montrons que le système modifié permet une analyse fine du flot de contrôle dans un langage d'ordre supérieur. Ceci nous permet de construire un graphe, dit graphe de dépendance approximé, et nous pouvons montrer qu'un critère syntaxique sur ce graphe suffit à montrer la terminaison de tout terme bien typé.

Mots clés: Réécriture, ordre supérieur, typage, normalisation, tailles, annotations sémantiques, prémodèle, paires de dépendance.

Abstract

The present manuscript is a reflection on termination of higher-order rewrite systems. We concentrate our efforts on a particular approach, size-based termination. This method uses typing to give a syntactic approximation to the size of an element of the language.

Our contribution is twofold: first we give a structured approach to proving the correctness of size-based termination. To do this, we show that it is possible to apply a certain version of semantic labelling. This technique uses annotations on terms computed using the semantics of subterms in a certain equational premodel. We show correctness of our labelling framework and of the criterion that allows us to prove termination of the labelled system, and we build a premodel of the rewrite system that intuitively corresponds to the rewrite system. We show that the system labelled using these semantics passes the termination criterion.

Furthermore we show that a modification of the classical size-types approach allows us to perform a fine control-flow analysis in a higher-order language. This allows us to build an approximated dependency graph, and show that if a certain syntactic criterion is satisfied by the graph, then all well-typed terms are terminating.

Keywords: Rewriting, higher-order, typing, termination, size, semantic labelling, premodel, dependency pairs.