Le Coq' Art (V8)

Yves Bertot, Pierre Castéran

26 janvier 2011

Table des matières

*Les annotations de niveau : *, **, et *** apparaissant dans cette table des matières sont expliquées page 32.*

Chapitre 1

Préface

Lorsque Don Knuth entreprit sa grande œuvre de poser les fondements de l'informatique dans un traité sur la programmation, il n'intitula pas l'ouvrage "The Science of Computer Programming", mais "The Art of Computer Programming". En effet, il aura fallu plus de 30 ans de recherches supplémentaires pour véritablement élaborer une discipline rigoureuse de la programmation, l'algorithmique. De manière similaire, les fondements rigoureux de la conception de preuves formelles sont encore en cours d'élaboration ; bien que les concepts principaux de la théorie de la démonstration remontent aux travaux de Gentzen, Gödel et Herbrand dans les années 30, et que Turing lui-même s'intéressa très tôt à l'automatisation de la construction de preuves mathématiques, ce n'est qu'au milieu des années 60 que les premières expériences d'automatisation de logique de premier ordre, par énumération systématique du domaine de Herbrand, virent le jour. 40 ans plus tard, l'assistant de preuves Coq est l'aboutissement d'une longue lignée de recherche en logique computationnelle, et d'une certaine manière représente l'état de l'art en la matière, mais son utilisation effective est elle-même un art difficile à acquérir et à perfectionner. C'est pourquoi l'ouvrage d'Yves Bertot et Pierre Castéran est un guide extrêmement précieux, car il permet à la fois aux néophytes de s'initier à l'art de Coq, et aux utilisateurs chevronnés de devenir de véritables experts et de développer les preuves mathématiques requises pour la certification d'applications en vraie grandeur.

Un petit historique du système Coq peut aider à situer ce logiciel et les objets mathématiques qu'il met en œuvre. La genèse des idées sous-jacentes peut également guider dans la compréhension des mécanismes avec lesquels l'utilisateur du système doit interagir, le choix des facettes à utiliser pour une modélisation, les options à envisager en cas de difficultés.

Gérard Huet commença à travailler en démonstration automatique en 1970, en implémentant en LISP un système SAM de preuves en logique de premier ordre permettant de traiter des axiomatisations équationnelles. L'état de l'art à l'époque était de traduire les propositions logiques en matrices et/ou (listes de clauses), les quantifications étant remplacées par des fonctions de Skolem, et les étapes de déduction étant réduites à l'utilisation d'un principe d'appariement de formules atomiques complémentaires modulo instantiation (résolution avec unificateur principal). Les égalités donnaient lieu à des réécritures uni-directionnelles, modulo unification également. L'ordre des réécritures était déterminé de façon relativement ad hoc, et il n'y avait donc pas d'assurance de convergence, ni de complétude. Les démonstrateurs étaient des boîtes noires qui engendraient des milliers de conséquences logiques illisibles, à cause de la normalisation initiale des formules. Le mode standard d'utilisation était de rentrer sa conjecture et d'attendre que la mémoire de l'ordinateur soit saturée. Seulement dans des cas exceptionnellement triviaux une réponse était obtenue. Cet état de l'art catastrophique n'était d'ailleurs pas reconnu comme tel, car il était plus ou moins analysé comme étant inéluctable, à cause notamment des théorèmes d'incomplétude. Néanmoins, les études de complexité allaient bientôt montrer que même dans le domaine décidable, et même dans la forme logique la plus simple (calcul propositionnel), la démonstration automatique était vouée à se heurter au mur de l'explosion combinatoire.

Une percée décisive fut néanmoins accomplie dans les années 70 avec la mise au point, à partir de l'article fondateur de Knuth et Bendix, d'une méthodologie systématique de réécriture guidée par des ordres de terminaison. C'est ainsi que le logiciel KB, réalisé en LISP en 1980 par Jean-Marie Hullot et Gérard Huet, permit d'automatiser de manière naturelle des procédures de décision ou semi-décision dans des structures algébriques. À cette époque, de grands progrès avaient été effectués dans le domaine des preuves par récurrence, notamment avec le système NQTHM/ACL de Boyer et Moore. Une autre avancée avait été la généralisation du principe de résolution à la logique d'ordre supérieure, par la conception d'un algorithme d'unification en théorie des types simples par Gérard Huet en 1972, cohérent avec une problématique générale d'unification dans une théorie équationnelle développée en parallèle par Gordon Plotkin.

À la même époque, des logiciens (Dana Scott) et informaticiens théoriciens (Gordon Plotkin, Gilles Kahn, Gérard Berry) mettaient au point une théorie logique des fonctions calculables (les domaines de calcul) munies d'une axiomatisation effective (récurrence computationnelle) permettant de représenter la sémantique des langages de programmation. On pouvait ainsi espérer aborder de manière rigoureuse le problème de la mise au point de logiciels fiables par des méthodes formelles. La validité d'un programme vis-à-vis de ses spécifications logiques pouvait s'exprimer par un théorème dans une théorie mathématique exprimant les structures de données et de contrôle utilisées par l'algorithme. Ces idées furent mises en œuvre notamment par l'équipe de Robin Milner à l'Université d'Edimbourg qui réalisa le système LCF vers 1980. L'originalité principale de ce système résidait dans la notion de tactique de preuve programmable dans un méta-langage (ML). Les formules n'étaient pas réduites à des clauses indéchiffrables, et l'utilisateur pouvait donc utiliser son intuition et sa connaissance du domaine axiomatisé pour guider le système dans des démonstrations qui mélangeaient des étapes automatiques (par combinaison des tactiques prédéfinies et programmation de tactiques spécifiques dans le méta-langage), avec des étapes manuelles compréhensibles.

Une autre voie d'investigation était ouverte par le philosophe Per Martin-Löf, à partir du fondement constructif des mathématiques initié par l'intuitionnisme de Brouwer et poursuivi notamment par la présentation constructive de l'Analyse par Bishop. Sa Théorie des Types Intuitionniste, élaborée au début des années 80, donnait un cadre élégant et général à l'axiomatisation constructive de structures mathématiques, pouvant servir de fondements à la programmation fonctionnelle. Cette voie fut poursuivie sérieusement par Bob Constable à l'Université Cornell, qui mit en chantier un programme NuPRL de conception de logiciels à partir de preuves formelles, et par l'équipe de "Programming methodology" dirigée par Bengt Nordström à l'Université Chalmers de Göteborg.

Toutes ces recherches s'appuyaient sur la notation du λ-calcul, originellement conçu par le logicien Alonzo Church dans sa version pure comme langage de description de fonctionnelles récursives, et dans sa version typée comme langage de prédicats d'ordre supérieur (théorie des types simples), proposé comme langage de méta-description de mathématiques plus simple que le système utilisé autre-fois par Whitehead et Russell dans "Principia Mathematica". Mais le λ-calcul pouvait également servir de support à la représentation des preuves en déduction naturelle, donnant lieu à la fameuse "correspondance de Curry-Howard" qui exprime l'isomorphisme entre des structures de preuves et des structures fonctionnelles. Cette dualité du λ-calcul était d'ailleurs mise en œuvre dans le système Automath de représentation des Mathématiques conçu par Niklaus de Bruijn à Eindhoven dans les années 70. Dans ce système les types des λ-expressions n'étaient plus de simples stratifications fonctionnelles, mais étaient eux-mêmes des λ-expressions exprimant la possible dépendance du type résultat d'une fonctionnelle sur la valeur de son argument – en analogie avec l'extension du calcul propositionnel en calcul de prédicats prenant en argument des termes représentant des éléments du domaine support.

Le λ-calcul était d'ailleurs le principal outil en théorie de la démonstration. En 1970, Jean-Yves Girard donna la preuve de cohérence de l'Analyse à partir d'une preuve de terminaison d'un λ-calcul polymorphe appelé système F, généralisable en un calcul F_ω de fonctionnelles polymorphes, permettant l'expression d'une classe d'algorithmes transcendant les hiérarchies ordinales traditionnelles. Ce système allait d'ailleurs être redécouvert en 1974 par John Reynolds comme langage de programmation générique, généralisant le polymorphisme restreint présent dans le langage ML.

Au début des années 80 il y a avait donc une grande effervescence de recherches aux frontières de la logique et de l'informatique, sous la bannière de la Théorie des Types. En 1982 Gérard Huet démarrait le projet Formel au laboratoire de Rocquencourt de l'INRIA, en collaboration avec Guy Cousineau et Pierre-Louis Curien du Laboratoire d'Informatique de l'Ecole Normale Supérieure. Cette équipe se donnait pour objectif la conception et la réalisation d'un système de preuves capitalisant sur les idées du système LCF, et notamment lui empruntant le langage ML, non seulement aux fins de servir de méta-langage à un système de tactiques de preuves, mais aussi comme langage d'implémentation de l'ensemble de l'assistant de preuves. Cet effort de recherche et de développement en programmation fonctionnelle allait donner naissance au fil des années à la famille des langages Caml, dont est issu Objectif Caml, support d'implémentation de l'Assistant de Preuves Coq.

En 1984 Thierry Coquand et Gérard Huet présentèrent à la Conférence Internationale sur les Types organisée à Sophia-Antipolis par Gilles Kahn une synthèse des types dépendants et du polymorphisme, permettant d'accommoder à la fois la théorie constructive de Martin-Löf et le polymorphisme de F_ω au sein d'une généralisation du système Automath appelée *Calcul des Constructions*. Thierry Coquand donna dans sa thèse l'analyse méta-théorique du λ-calcul sous-jacent. En prouvant la terminaison du calcul il donna ainsi la preuve de cohérence du système logique. Ce calcul fut adopté comme support logique du système de preuves mis en chantier au projet Formel, et Gérard Huet proposa un premier vérificateur du calcul (CoC) à partir de sa *Constructive Engine*, utilisant les indices de de Bruijn pour l'opération de substitution. Un premier prototype en Caml de ce vérificateur de types permettait de présenter les premiers développements mathématiques au congrès Eurocal en Avril 1985.

Le premier étage de ce qui allait devenir le système Coq était ainsi constitué : un vérificateur de types d'une λ-expression représentant un terme de preuve du système logique ou le corps de la définition d'un objet mathématique. Ce cœur de l'assistant de preuves était complètement découplé des mécanismes de synthèse des termes soumis à vérification – l'interprète de la machine constructive est un programme déterministe. Thierry Coquand réalisa une machine de synthèse de preuves dans un calcul de séquents permettant la construction d'un terme-preuve par raffinements progressifs, programmable avec un jeu de tactiques inspiré du système LCF. Ce deuxième étage allait bientôt être complété par Christine Mohring, avec une première implémentation d'un algorithme de recherche de preuves à la PROLOG, la fameuse tactique Auto. Le système Coq était né dans ses grandes lignes, puisqu'aujourd'hui encore le noyau de Coq re-vérifie le terme de preuves synthétisé par les tactiques mises en œuvre par l'utilisateur. Cette architecture permet d'ailleurs de simplifier la machine de recherche de preuves, qui fait l'économie de certaines contraintes de stratification du système de types.

Très vite l'équipe Formel envisagea l'utilisation du Calcul des Constructions pour la synthèse de programmes certifiés, dans l'esprit du système NuPRL, et en profitant de la puissance du polymorphisme, qui permet d'exprimer par un type du système F une structure algébrique telle que les entiers, systématisant une méthode due à Böhm et Berarducci. Christine Mohring se consacra à cette problématique, et implémenta une tactique complexe de synthèse de principes de récurrence en CoC qui lui permit de présenter une méthode de développement d'algorithmes certifiés au congrès Logic in Computer Science (LICS) de Juin 1986. Mais en poussant cette étude dans sa thèse, elle réalisa que les codages "imprédicatifs" des structures de données n'étaient pas fidèles à la tradition où les structures d'un type donné sont réduites aux termes obtenus en composant les constructeurs de type. Les codages du λ-calcul introduisaient des termes parasites et ne permettaient pas d'exprimer les bons principes de récurrence. Cet échec relatif permit en fait à Christine Mohring, en collaboration avec Thierry Coquand, de concevoir en 1988 une extension du formalisme en un Calcul des Constructions Inductives (CIC), doté lui des bonnes propriétés pour l'axiomatisation des algorithmes sur des structures de données inductives.

L'équipe Formel a toujours eu le souci d'équilibrer les recherches fondamentales avec l'expérimentation avec des maquettes destinées à fournir des preuves de faisabilité, des logiciels prototypes permettant le développement de preuves en vraie grandeur, et des systèmes plus conséquents, distribués sur le modèle du logiciel libre, avec une bibliothèque maintenue, une documentation, et un souci de compatibilité entre versions successives. Le prototype *CoC* interne au projet devint ainsi le logiciel *Coq*, diffusé auprès d'une communauté d'utilisateurs communiquant à travers un forum électronique. Pour autant, les recherches en amont n'étaient pas négligées, et par exemple Gilles Dowek développait une théorie systématique de l'unification et de la recherche de preuves en théorie des types destinée à fournir un fondement aux versions suivantes de Coq.

En 1989 Coq était diffusé, dans sa version 4.10, avec un premier système d'extraction de programmes fonctionnels Caml à partir de preuves dû à Benjamin

Werner, un jeu de tactiques permettant un certain degré d'automatisation des preuves, et une petite bibliothèque de développements mathématiques et informatiques. Une nouvelle époque commençait. Thierry Coquand partit enseigner à Göteborg, Christine Paulin-Mohring partit à l'Ecole Normale Supérieure de Lyon, et le projet Coq continua les recherches et le développement du système avec une équipe bilocalisée entre Lyon et Rocquencourt, alors que le projet Cristal continuait les recherches en programmation fonctionnelle autour du langage Caml. À Rocquencourt, Chet Murthy, qui venait de finir sa thèse dans l'équipe NuPRL sur l'interprétation constructive des preuves de logique classique, apporta sa contribution au développement d'une architecture plus complexe de la version 5.8 de Coq. Un effort Européen conséquent s'organisa autour du Basic Research Action "Logical Frameworks", suivi 3 ans plus tard de sa continuation par le BRA "Types". Plusieurs équipes unirent leurs efforts de recherche autour de la conception d'assistants de preuve bénéficiant d'une émulation stimulante : Coq bien sûr, mais aussi LEGO, développé par Randy Pollack à Edimbourg, Isabelle, développé par Larry Paulson à Cambridge puis Tobias Nipkow à Munich, Alf, développé par l'équipe de Göteborg, etc.

En 1991 Coq V5.6 offrait un langage uniforme de description mathématique (le "vernaculaire" Gallina), des types inductifs primitifs, l'extraction de programmes Caml à partir de preuves, et une interface graphique. Coq était maintenant un système opérationnel, permettant de démarrer des collaborations industrielles fructueuses, notamment avec le CNET et Dassault-Aviation. Cette première génération d'utilisateurs extérieurs motiva le développement d'un Tutorial et d'un Manuel de Référence, même si l'Art de Coq restait encore une affaire largement hermétique aux non-initiés. Car Coq restait avant tout un véhicule de recherche et d'expérimentation. À Sophia-Antipolis, Yves Bertot reconvertissait l'effort Centaur de manipulation de structures en un interface CT-Coq permettant d'expérimenter la conception interactive de preuves par une méthodologie originale de "preuves par pointage", où l'utilisateur met en œuvre un jeu de tactiques par le truchement de la désignation d'hypothèses pertinentes avec la souris d'un poste de travail. À Lyon, Catherine Parent montra dans sa thèse comment inverser la problématique d'extraction de programmes à partir de preuves en une problématique de programmes décorés par des invariants, utilisés comme squelettes de leur propre preuve de correction. À Bordeaux, Pierre Castéran montra comment cette technologie permettait l'élaboration de bibliothèques certifiées d'algorithmes définis par des sémantiques de continuations. À Lyon, Eduardo Giménez montra dans sa thèse comment étendre la problématique des types inductifs définissant des structures héréditairement finies en une problématique de types co-inductifs, permettant d'axiomatiser des structures potentiellement infinies, et par suite de faire des preuves de protocoles travaillant sur des flots de données, ouvrant la voie à d'importantes applications aux télécommunications.

À Rocquencourt, Samuel Boutin montra dans sa thèse comment implémenter en Coq des raisonnements par réflexion, permettant notamment d'automatiser des étapes fastidieuses par réécriture algébrique. Sa tactique *Ring* permet ainsi de simplifier les expressions polynomiales et donc de rendre implicites les manipulations algébriques usuelles sur les expressions arithmétiques. D'autres procédures de décision permirent d'augmenter considérablement le champ de l'automatisation des preuves en Coq : *Omega* dans le domaine de l'arithmétique de Presburger (Pierre Crégut au CNET-Lannion), *Tauto* et *Intuition* dans le domaine propositionnel (César Muñoz à Rocquencourt), *Linear* pour le calcul des prédicats sans contraction (Jean-Christophe Filliâtre à Lyon). Amokrane Saïbi montra comment une notion de sous-typage avec héritage et coercions implicites permettait de faire des développements modulaires en algèbre universelle, et notamment de représenter élégamment en Coq les notions principales de la Théorie des Catégories.

En Novembre 1996 Coq V6.1 intégrait dans sa version distribuée toutes les avancées théoriques mentionnées, mais aussi nombre d'améliorations technologiques essentielles pour l'amélioration de ses performances, notamment une machine de réduction efficace due à Bruno Barras et inspirée de l'évaluateur du langage Caml ainsi que des tactiques avancées de manipulation de définitions inductives dûes à Cristina Cornes. Un traducteur de preuves en langue naturelle (anglais et français) dû à Yann Coscoy permettait de représenter sous une forme lisible les termes de preuve calculés par les tactiques – avantage considérable par rapport à des systèmes concurrents ne donnant pas accès à une démonstration explicite permettant l'audit de certifications formelles.

Dans le domaine de la certification de programmes, J.C. Filliâtre montra en 1999 dans sa thèse comment implémenter en Coq les raisonnements sur des programmes impératifs, en renouvellant la problématique des assertions de Floyd-Hoare-Dijkstra sur des programmes impératifs considérés comme notation pour les expressions fonctionnelles issues de leur sémantique dénotationnelle. L'architecture à deux niveaux de Coq était confortée par la certification par Bruno Barras du vérificateur de CoC extrait automatiquement de la formalisation en Coq de la métathéorie du Calcul des Constructions – tour de force technique, mais aussi avancée considérable dans la fiabilité des méthodes formelles. Inspiré par le système de modules d'Objectif Caml, Judicaël Courant jetait dans sa thèse les fondements d'un langage modulaire de développements mathématiques, permettant d'envisager la réutilisation de bibliothèques et le développement de logiciels certifiés à large échelle.

La création en 1999 de la Société Trusted Logic, spécialisée dans la certification de systèmes à base de cartes à puces, et utilisant des technologies issues des équipes Caml et Coq, conforta les chercheurs dans la pertinence de leur démarche. De nombreux projets applicatifs se mirent en place.

Le système Coq était alors complètement remis en chantier dans une version 7 reposant sur un noyau fonctionnel, les maîtres d'œuvre étant Jean-Christophe Filliâtre, Hugo Herbelin et Bruno Barras. Un nouveau langage de tactiques était conçu par David Delahaye, offrant à l'utilisateur un langage de haut niveau pour programmer des stratégies de preuve sophistiquées. Micaela Mayero attaquait l'axiomatisation des réels, dans la perspective de certification d'algorithmes numériques. Yves Bertot de son côté remettait en chantier les idées de CTCoq dans un interface graphique sophistiqué *PCoq* développé en Java.

En 2002, quatre ans après la thèse de Judicaël Courant, Jacek Chrząszcz réussissait à intégrer dans Coq un système de modules et foncteurs analogue à celui de Caml. S'intégrant parfaitement dans l'environnement de développement de théories, cette extension améliore considérablement la généricité des bibliothèques. Pierre Letouzey proposait un nouvel algorithme d'extraction de programmes Caml à partir de preuves prenant en compte l'ensemble du langage de Coq y compris les modules.

Du côté des applications, Coq est devenu assez robuste pour servir de langage de bas niveau à des outils spécialisés dans des activités de preuves de programmes. C'est le cas de la plateforme CALIFE de modélisation et preuve d'automates temporisés, de l'outil *Why* de preuve de programmes impératifs ou de l'outil *Krakatoa* pour la certification d'applets Java développé dans le cadre du projet européen VERIFICARD. Ces outils tirent parti du langage de Coq pour établir les propriétés des modèles et lorsque la complexité des propositions à établir dépasse les possibilités des outils automatiques.

Un succès industriel majeur vient de récompenser la société Trusted Logic, à l'issue de 3 ans d'efforts de modélisation formelle de l'ensemble de l'environnement d'exécution du langage Java Card, conformément à la cible de sécurité "Sun Java Card System Protection Profile". Cette méthodologie de sécurité vient en effet d'obtenir le niveau de certification EAL7 (le plus élevé). Ce développement formel a nécessité l'écriture de 121000 lignes de Coq réparties en 278 modules.

Coq est également utilisé pour le développement de bibliothèques de théorèmes mathématiques avancés sous des formes constructives ou classiques. Ce domaine d'application a requis de restreindre le langage logique utilisé par défaut par Coq afin de rester compatible avec certains axiomes naturels au mathématicien.

Fin 2003, à l'occasion d'une réforme majeure de la syntaxe du langage, la première version 8.0 de Coq est distribuée – c'est cette version que Coq'Art utilise.

Il suffit de consulter le sommaire des contributions des utilisateurs de Coq, à l'URL `http://coq.inria.fr/contribs/summary.html`, pour se convaincre de la richesse des développements mathématiques disponibles en Coq aujourd'hui. En effet, l'équipe d'implémenteurs a fait sienne l'exigence de Boyer et Moore de toujours transporter au fil des versions les bibliothèques finalisées, au besoin en proposant des outils de conversion aidant au transport par conversion automatique des scripts de preuve – une assurance pour les utilisateurs que leurs développements dans la version courante ne seront pas perdus dans une version future incompatible. On remarquera que nombre de ces bibliothèques ont été développées par des utilisateurs externes à l'équipe de développement, souvent à l'étranger, parfois dans des équipes industrielles. On ne peut qu'admirer la tenacité de cette communauté d'utilisateurs à aller jusqu'au bout de développements formels de grande complexité, à l'aide d'un système Coq toujours relativement expérimental, et n'ayant pas bénéficié jusqu'ici d'un ouvrage compréhensif et progressif servant de manuel d'utilisation.

Avec Coq'Art ce besoin est maintenant rempli. Yves Bertot et Pierre Castéran sont des utilisateurs chevronnés de Coq dans ses diverses versions depuis de nombreuses années. Mais ils sont aussi des “clients" extérieurs à l'équipe de développment, et à ce titre sont moins tentés de dissimuler des difficultés “bien connues" qu'un initié dans le secret tendra à occulter. Ils ne sont pas tentés non plus d'annoncer prématurément des solutions encore à l'état de recherches préliminaires – tous leurs exemples sont exécutables dans la version courante de distribution. Ils nous présentent dans leur ouvrage une introduction progressive à l'ensemble des fonctionnalités du système. Cette quasi exhaustivité se paye bien sûr par l'épaisseur considérable du document. Que l'utilisateur débutant n'en soit pas rebuté, il sera guidé dans sa lecture par des indications de difficulté, et ne doit pas s'astreindre à une lecture exhaustive de l'ensemble de l'ouvrage. Il s'agit donc ici d'un ouvrage de référence, qu'un utilisateur au long cours consultera au fur et à mesure des difficultés qu'il rencontrera dans son utilisation du système. La taille de l'ouvrage est due également à l'utilisation de nombreux exemples de taille conséquente, qui sont disséqués progressivement. Le lecteur sera souvent heureux de détailler ces développements en les reproduisant en tête à tête avec l'animal. De fait, il est grandement conseillé de ne lire Coq'Art qu'à proximité d'un poste de travail exécutant une session Coq, afin de contrôler les comportements du système au fur et à mesure de sa lecture. Cet ouvrage présente le résultat de près de 30 ans de recherches en méthodes formelles, et la complexité intrinsèque du domaine ne peut pas être escamotée – il y a un prix non négligeable à payer pour devenir expert d'un système tel que Coq. Inversement, néanmoins, la genèse de Coq'Art au fil des 3 dernières années a été un incentif fort à uniformiser les notions et leurs notations, à présenter des outils de preuve explicables sans mystères excessifs, à communiquer à l'utilisateur les anomalies ou difficultés avec des messages d'erreur compréhensibles par des non-spécialistes de la méta-théorie – avec plus ou moins de bonheur, bien sûr. Nous souhaitons au lecteur bonne chance dans la découverte d'un monde difficile, mais passionnant – que ses efforts soient récompensés par la joie du dernier CQFD, qui clôt des semaines ou parfois des mois de travail acharné mais inconclusif par la touche finale qui valide l'ensemble de l'ouvrage.

21 Novembre 2003
Gérard Huet et Christine Paulin-Mohring

Remerciements

De nombreux collègues ont apporté leur aide enthousiaste à l'élaboration de cet ouvrage. Nous remercions tout spécialement Laurence Rideau pour son soutien enjoué et sa lecture attentive, depuis les premiers balbutiements jusqu'à la dernière version. Gérard Huet et Janet Bertot ont également investi leur temps et leurs efforts pour améliorer à la fois la précision technique et le style du livre. Nous tenons également à exprimer une gratitude particulière à Gérard Huet et Christine Paulin pour la préface.

L'équipe de developpement de Coq dans son ensemble mérite notre gratitude pour avoir produit un outil si puissant. En particulier, Christine Paulin-Mohring, Jean-Chrstophe Filliâtre, Eduardo Giménez, Jacek Chrząszcz et Pierre Letouzey nous ont guidé dans les aspects techniques des types inductifs, des modules, de la représentation des programmes impératifs, des types co-inductifs et de l'extraction, et parfois il nous ont également fourni quelques pages et quelques exemples. Hugo Herbelin et Bruno Barras ont joué un rôle central pour nous permettre de fournir un livre dans lequel tous les exemples peuvent être reproduits sur machine.

Nous avons acquis notre connaissance du domaine au cours des nombreuses expériences menées avec les étudiants que nous avons eu la chance d'encadrer et à qui nous avons enseigné. En particulier, certaines des idées développées dans ce livre n'ont été vraiment comprises qu'après nos enseignements à l'École Normale Supérieure de Lyon et à l'Université de Bordeaux et après que nous ayons étudié les questions soulevées, et souvent résolues, en collaboration avec Davy Rouillard, Antonia Balaa, Nicolas Magaud, Kuntal Das Barman et Guillaume Dufay.

De nombreux étudiants et chercheurs ont passé du temps à lire les versions préliminaires de ce livre, l'ont utilisé comme support pour leur enseignement ou leur apprentissage et ont suggéré des améliorations ou des solutions alternatives. Nous tenons à les remercier pour leurs réactions si précieuses : Hugo Herbelin, Jean-François Monin, Jean Duprat, Philippe Narbel, Laurent Théry, Gilles Kahn, David Pichardie, Jan Cederquist, Frédérique Guilhot, James McKinna, Iris Loeb, Milad Niqui, Solange Coupet-Grimal, Sébastien Hinderer et Simão Melo de Sousa.

Les institutions et les équipes de recherche qui nous ont accueillis ont également joué un rôle important dans notre réussite pour mener ce projet à bien. En particulier, nous remercions les projets Lemme et Signes de l'INRIA et de l'Uni-

versité de Bordeaux et le groupe de travail européen Types, qui nous a donné l'opportunité de rencontrer des jeunes chercheurs comme Ana Bove, Venanzio Capretta ou Conor McBride qui ont inspiré certains des exemples détaillés dans notre ouvrage.

Chapitre 2

Un bref tour d'horizon

Coq [38] est un assistant de preuves permettant l'expression de spécifications et le développement de programmes cohérents avec leur spécification. Cet outil s'applique alors parfaitement au développement de programmes en lesquels une confiance absolue est requise : télécommunications, sécurité des transports, énergie, cryptologie, etc. Dans ces domaines, le besoin de programmes rigoureusement conformes à leur spécification justifie l'effort demandé par leur validation. Nous verrons tout au long de cet ouvrage comment un *assistant de preuves* tel que *Coq* peut faciliter ce travail.

L'intérêt de *Coq* ne se limite pas au développement de programmes sûrs. *Coq* est avant tout un système permettant de faire des preuves dans une logique très expressive, dite *d'ordre supérieur*. Ces preuves sont construites de façon interactive, assistée par des outils de recherche automatique de preuves dans des domaines où cela est possible.

Les domaines d'utilisation de *Coq* sont très variés : logique, automates, syntaxe et sémantique des langues naturelles, algorithmique, etc. (voir [1]). On peut aussi le considérer comme un cadre logique permettant l'axiomatisation de logiques et le développement interactif de preuves dans ces logiques. Citons par exemple l'implémentation de systèmes de raisonnement dans des logiques modales, temporelles, logiques de ressources et les systèmes de raisonnement sur les programmes impératifs.

Coq fait partie d'une longue tradition de systèmes informatiques d'aide à la démonstration de théorèmes. Citons par exemple les systèmes *Automath* [35], *Nqthm* [18, 19], *Mizar* [85], *LCF* [50], *Nuprl* [26], *Isabelle* [75], *Lego* [62], *HOL* [49], *PVS* [70], et *ACL2* [57].

Un des points les plus remarquables de *Coq* est la possibilité de synthétiser des programmes certifiés à partir de preuves, et, depuis peu, des modules certifiés.

Dans cette introduction, nous nous proposons de présenter informellement les traits principaux de *Coq*. Les définitions rigoureuses et les notations précises seront présentées dans les chapitres ultérieurs, aussi allons-nous utiliser des notations mathématiques ou empruntées à des langages de programmation usuels.

2.1 Expressions, types et fonctions

En premier lieu, le langage de spécifications de *Coq*, appelé *Gallina*, permet de représenter les types usuels des langages de programmation, ainsi que les programmes.

Les expressions de ce langage sont formées à partir de constantes et d'identificateurs par le biais de règles de construction. Toute expression possède un type, le type d'un identificateur est défini par une *déclaration*, et les règles permettant de former des expressions composées sont accompagnées de *règles de typage* établissant la relation entre le type des constituants et celui de l'expression composée.

Prenons pour exemple le type `Z` des nombres entiers, associé à l'ensemble $\mathbb{Z}$. La constante -6 est de ce type, et si l'on déclare une variable z de type `Z`, l'expression $-6z$ est aussi de type `Z`. En revanche, la constante `true` est de type `bool`, et l'on ne peut former l'expression `true` $\times -6$.

Nous allons rencontrer une très grande variété de types en *Gallina* ; à côté de `Z` et `bool`, nous utiliserons le type `nat` des entiers naturels, construit comme le type le plus petit contenant la constante 0 et clos par le passage au successeur. Par ailleurs, des opérateurs de types nous permettront de construire le type $A \times B$ des couples de valeurs (a, b) où a est de type A et b de type B, le type `list(`A`)` des listes dont les éléments sont de type A, et le type $A \rightarrow B$ des fonctions associant à tout argument de type A un résultat de type B.

Par exemple, la fonctionnelle qui à toute fonction f de `nat` dans `Z` et à tout entier naturel n associe $\Sigma_{i=0}^{i=n} f(i)$ aura pour type `(nat→Z)→nat→Z`.

Il est important de préciser dès maintenant que nous prenons le terme *fonction* dans un sens informatique : celui d'un procédé effectif de calcul (algorithme) associant à toute valeur de type A une valeur de type B, et non au sens de la théorie des ensembles : un sous-ensemble du produit cartésien $A \times B$ possédant certaines propriétés.

En *Coq*, le calcul d'une valeur s'effectue par réductions successives jusqu'à l'obtention d'une forme irréductible. Une propriété fondamentale du formalisme associé à *Coq* est qu'un tel calcul finit toujours par terminer (propriété de *terminaison uniforme*.) Rappelons que les résultats classiques d'indécidabilité montrent qu'un langage de programmation permettant d'exprimer toutes les fonctions calculables doit permettre d'exprimer des fonctions partielles. Par conséquent, tous les calculs effectués par lse fonctions calculables ne peuvent pas être pris en charge par ce méchanisme de réduction : il existe des fonctions que nous savons décrire en *Coq*, mais que nous ne pouvons calculer qu'à l'extérieur de *Coq* (grâce à mécanisme d'extraction). Malgré cette limitation, le système de typage de *Coq* est suffisamment puissant pour qu'on puisse décrire dans ce langage une très grande sous-classe des fonctions calculables. La propriété de terminaison uniforme ne correspond pas à une réduction significative de puissance d'expression.

2.2 Propositions et preuves

Le système *Coq* ne se borne pas à être un langage de programmation fonctionnel de plus. Il permet en effet d'exprimer des *assertions* sur les expressions. Ces expressions peuvent être aussi bien des représentations d'objets mathématiques dont on étudie les propriétés, que des programmes dont on veut exprimer la correction.

Voici quelques exemples de telles assertions (ou *propositions*) :

- $3 \leq 8$
- $8 \leq 3$
- « pour tout $n \geq 2$ la suite d'entiers définie par

$$\begin{array}{rcl} u_0 & = & n \\ u_{i+1} & = & 1 \text{ si } u_i = 1 \\ & & u_i/2 \text{ si } u_i \text{ est pair,} \\ & & 3u_i + 1 \text{ sinon} \end{array}$$

  est ultimement constante de valeur 1 »,
- « l'opération de concaténation de listes est associative »,
- « l'algorithme `insertion_sort` est une méthode correcte de tri »

Nous remarquons que certaines de ces assertions sont vraies, d'autres fausses, et pour l'une — la troisième [1] — nous ne savons pas encore si elle est vraie ou fausse. Néanmoins, tous ces exemples constituent des propositions correctement formées.

Pour s'assurer de la véracité d'une proposition P, le moyen le plus classique et le plus sûr est d'en fournir une preuve. Si celle-ci est complète et lisible, on peut toujours la contrôler. Cette possibilité impose certaines contraintes que satisfait rarement la littérature scientifique. En effet, les ambiguïtés inhérentes à toute langue naturelle rendent difficile la vérification de correction d'une démonstration ; d'autre part, la preuve complète d'un théorème devient vite un texte énorme, et de nombreuses étapes de raisonnement sont omises afin de rendre le texte plus lisible.

Une solution possible à ce problème est de définir un langage formel pour les démonstrations, construit selon des règles précises empruntées à la théorie de la démonstration. Ceci assure que toute démonstration peut être vérifiée pas à pas.

La très grande taille des démonstrations complètes rend nécessaire la mécanisation de cette vérification. Il suffit pour avoir confiance dans cette vérification de montrer que l'algorithme utilisé contrôle bien l'application des règles formelles de construction de démonstration.

C'est également ce problème de taille qui rend quasiment impossible l'écriture *manuelle* des démonstrations. C'est alors que le terme d'*assistant de preuve* prend tout son sens. Étant donnée une proposition à prouver, le système propose des outils de construction de preuve. Ces outils, appelés *tactiques*, facilitent

1. La suite u dont parle cette assertion est connue sous le nom de « suite de Syracuse ».

la construction d'une preuve d'une proposition, en fonction de la proposition à prouver et du contexte de travail composé de toutes les déclarations, définitions, axiomes, hypothèses, lemmes et théorèmes utilisables.

Ces tactiques peuvent dans un grand nombre de cas permettre la construction automatique d'une preuve, mais ce n'est pas le cas général. Les résultats classiques d'indécidabilité et de complexité nous montrent qu'il est impossible de concevoir un algorithme général de construction automatique de démonstrations. C'est pourquoi *Coq* est un outil *interactif*, où l'utilisateur a souvent la responsabilité de découper une preuve difficile en un certain nombre de lemmes, ou de choisir la tactique appropriée à un cas difficile. Les tactiques offertes sont très variées, et l'utilisateur avancé a la possibilité de construire lui-même des tactiques adaptées à sa problématique (voir en section 8.5).

La présence à la fois d'une vérification fiable de démonstrations et d'outils pour construire ces dernières rend alors facultative la lecture humaine des preuves de théorèmes. En fait, l'utilisateur peut choisir à quel niveau de détail il souhaite considérer une démonstration.

2.3 Propositions et types

Une fois admise la nécessité d'un langage non-ambigu pour écrire les démonstrations, se pose la question de définir ce langage.

Selon une tradition remontant au projet Automath [35], le formalisme employé pour écrire les preuves est le même que pour écrire les programmes, à savoir le *lambda-calcul typé*. Ce formalisme, inventé par Church [25], est une des nombreuses façons de présenter des algorithmes effectifs, et a directement inspiré les langages de programmation de la famille *ML*. *Coq* utilise une version très expressive du lambda-calcul typé, le *Calcul des Constructions Inductives* [29, 72].

Le chapitre 4 est l'occasion d'aborder la relation profonde entre preuves et programmes, appelée l'*Isomorphisme de Curry-Howard*. La relation entre les programmes et leur type est la même qu'entre preuves et propositions. Le travail de vérification d'une preuve est assuré par un algorithme de vérification de types. Nous verrons tout au long de cet ouvrage comment l'utilisation de *Coq* est facilitée par une double intuition venant d'une part de la programmation fonctionnelle, d'autre part de la pratique du raisonnement.

Une des caractéristiques importantes du Calcul des Constructions est que tout type est aussi un terme, et possède donc un type. Le type des propositions est appelé `Prop`. Par exemple, la proposition “ $3 \leq 7$ ” est à la fois le type de toutes les démonstrations que 3 est inférieur ou égal à 7, mais aussi un terme de type `Prop`.

De la même manière, un *prédicat* nous permet de construire une proposition paramétrique. Ainsi le prédicat « être premier » nous permet-il de former les propositions : « 7 est premier », « 1024 est premier », etc. On peut donc considérer ce prédicat comme une fonction, dont le type est `nat`$\rightarrow$`Prop` (voir chapitre 5). De même, le prédicat « être une liste ordonnée » pourra avoir le type `(list Z)`$\rightarrow$`Prop`, et la relation binaire $\leq$ le type `Z`$\rightarrow$`Z`$\rightarrow$`Prop`.

Les prédicats exprimables en *Coq* peuvent être de nature plus complexe, dans la mesure où leurs arguments peuvent être à nouveau des prédicats. Par exemple, la propriété d'être une relation transitive sur `Z` est un prédicat de type `(Z→Z→Prop)→Prop`. De même, on peut considérer un prédicat « être une relation transitive », dont le type *polymorphe* est le suivant :

`(`$A\rightarrow$ A`→Prop)→Prop` *pour tout type de donnée* A

2.4 Spécifications complexes et programmes certifiés

Nous avons vu dans quelques exemples de propositions que celles-ci peuvent faire référence à des données ou des programmes.

Le système de types de *Coq* permet de considérer également la réciproque : le type d'un programme peut contenir des contraintes que doivent satisfaire les données, exprimées sous forme de propositions. Par exemple, si n est de type `nat`, le type : « un nombre premier diviseur de n » contient une partie *calculatoire* : « un terme de type `nat` », et une partie *logique*, exprimée par le prédicat « être premier et diviser n ».

Ce genre de type, qualifié de *dépendant* contribue largement à la puissance d'expression de *Coq*. D'autres exemples de types dépendants sont les structures de données comportant des contraintes de taille : vecteurs de longueur n, arbres de hauteur h, etc.

De même, le type des fonctions qui à tout $n > 1$ associe un diviseur premier de n est un type constructible en *Coq* (voir chapitre 10). Un terme de ce type, que l'on peut construire en bénéficiant de toute l'interactivité de ce système, est appelé un *programme certifié*, et contient à la fois des informations de calcul : comment calculer ce diviseur premier, et une preuve que le résultat de ce calcul est bien un diviseur premier de n.

Un algorithme d'*extraction* permet d'obtenir à partir de ce programme certifié un programme *OCAML* [24] que l'on peut compiler et exécuter. Un tel programme, obtenu mécaniquement à partir d'une preuve de réalisabilité d'une spécification, présente un niveau maximal de fiabilité. Cet algorithme procède en effaçant tous les arguments logiques pour ne garder que la description des calculs à effectuer. Ceci rend importante la distinction entre ces deux types d'information. Ce dispositif d'extraction sera présenté dans les chapitres 11 et 12.

2.5 L'exemple du tri

Afin d'illustrer les possibilités de *Coq* décrites ci-dessus, nous présentons un exemple de façon informelle. Le source complet de cet exemple est fourni en annexe à la fin du livre et pourra également être consulté avec les solutions des exercices de ce livre [11].

Nous considérons le type `list Z` des listes d'éléments de type `Z`. Nous représentons les listes de la façon suivante : la liste vide est notée `nil`, la liste contenant 1, 5 et -36 est notée `1::5::-36::nil` et l'ajout de n en tête de la liste l est noté $n :: l$.

Comment spécifier un programme de tri ? Un tel programme est une fonction qui à tout l de type `list Z` associe une liste l' dont les éléments sont rangés en ordre croissant et ayant exactement les mêmes éléments que l, en respectant leur ordre de multiplicité. Ces propriétés sont formalisées sous la forme de deux prédicats dont la définition est donnée ci-dessous.

2.5.1 Définitions inductives

Pour définir le prédicat « être une liste ordonnée », nous pouvons nous inspirer du langage *Prolog*. Dans ce langage, on peut définir un prédicat sous la forme de *clauses* énumérant les conditions suffisantes pour qu'il soit satisfait. Dans notre cas, nous considérons trois clauses :

- la liste vide est ordonnée,
- toute liste contenant un seul élément est ordonnée,
- si une liste de la forme $n :: l$ est ordonnée, et si $p \leq n$, alors la liste $p :: n :: l$ est ordonnée.

Ceci revient à considérer le plus petit sous-ensemble X de `list Z` contenant la liste vide, les listes à un seul élément, et tel que, si une liste de la forme $n :: l$ appartient à X et $p \leq n$, alors $p :: n :: l$ appartient à X.

Une telle définition peut se présenter sous une forme logique comme une *définition inductive* d'un prédicat `sorted`, composée de trois *règles de construction* (*i.e.*, les clauses de *Prolog*).

Inductive sorted : `list Z`$\rightarrow$`Prop` :=

sorted0 : sorted(nil)

sorted1 : $\forall z : \mathtt{Z},\ \mathrm{sorted}(z :: nil)$

sorted2 : $\forall z_1, z_2 : \mathtt{Z},\ \forall l : \mathtt{list\ Z},\ z_1 \leq z_2 \Rightarrow \mathrm{sorted}(z_2 :: l) \Rightarrow \mathrm{sorted}(z_1 :: z_2 :: l)$

Ce type de définition sera traité dans les chapitres 9 et 15.

L'application des règles de construction nous permet de prouver aisément que, par exemple, la liste `[3; 6; 9]` est ordonnée. En outre, *Coq* engendre automatiquement des lemmes permettant de raisonner sur les listes ordonnées. Par exemple, les techniques d'*inversion* (voir section 9.5.2, page 276), permettent de montrer automatiquement le lemme suivant :

$$\textbf{sorted\_inv :}\ \forall z : \mathtt{Z},\ \forall l : \mathtt{list\ Z},\ \mathrm{sorted}(n.l) \Rightarrow \mathrm{sorted}(l)$$

2.5.2 La relation « avoir les mêmes éléments »

Il reste à définir une relation binaire exprimant qu'une liste l s'obtient par permutation à partir d'une liste l'.

Pour ce faire, un moyen simple consiste à définir une fonction `nb_occ` de type $\mathtt{Z}\rightarrow\mathtt{list\ Z}\rightarrow\mathtt{nat}$ telle que $\mathtt{nb\_occ}\ z\ l$ [2] soit le nombre de fois où l'entier z apparaît dans l. Cette fonction se définit simplement par récursion sur l. Dans une seconde étape, nous pouvons définir la relation suivante sur `list Z` :

$$l \equiv l' \ \textbf{si}\ \forall z : \mathtt{Z}, \mathtt{nb\_occ}\ z\ l = \mathtt{nb\_occ}\ z\ l'$$

Il faut remarquer que cette définition ne se veut absolument pas un moyen effectif de calcul. En effet, l'appliquer à la lettre reviendrait à comparer le nombre d'occurrences de z dans l et l' pour l'infinité d'éléments de $\mathbb{Z}$! En revanche, il est facile de prouver en quelques pas d'interaction avec *Coq* que la relation $\equiv$ est une relation d'équivalence, ainsi que les deux propriétés suivantes. Ces lemmes seront utilisés dans la certification de notre programme de tri.

equiv_cons : $\forall z : \mathtt{Z},\ \forall l, l' : \mathtt{list\ Z}\ , l \equiv l' \Rightarrow z :: l \equiv z :: l'$

equiv_perm : $\forall n, p : \mathtt{Z},\ \forall l, l' : \mathtt{list\ Z},\ l \equiv l' \Rightarrow n :: p :: l \equiv p :: n :: l'$

2.5.3 La spécification du tri

Nous avons tous les éléments pour spécifier une fonction de tri sur les listes de nombres entiers. Nous avons vu page 27 que *Coq* intégrait dans son système de types des spécifications complexes pouvant contenir des contraintes reliant données et résultats. La spécification d'un programme de tri sur `Z` sera alors le type `Z_sort` des fonctions qui à toute liste l : `list Z` associent une liste l' vérifiant la proposition $\mathrm{sorted}(l') \ \wedge\ l \equiv l'$.

Construire un programme certifié de tri revient à construire un terme de type `Z_sort`. Nous donnons les principales étapes de cette construction.

2.5.4 Une fonction auxiliaire

Nous choisissons pour des raisons de simplicité de réaliser la spécification du tri par un algorithme de tri par insertion.

Nous commençons par définir une fonction de type $\mathtt{Z}\rightarrow\mathtt{list\ Z}\rightarrow\mathtt{list\ Z}$. Cette fonction auxiliaire — que nous appelerons **`aux`** — a pour but de réaliser l'insertion d'un entier n dans une liste déjà triée l, et de renvoyer comme résultat une liste triée contenant à la fois n et les éléments de l.

Il est facile de définir $\mathtt{aux}\ n\ l$ de façon récursive, en faisant varier l'argument l :

- **si** l est vide, **alors** $\mathtt{aux}\ n\ l{=}n :: nil$,
- **si** l est de la forme $p :: l'$, **alors**
  - **si** $n \leq p$, **alors** $\mathtt{aux}\ n\ l{=}n :: p :: l'$
  - **si** $p < n$, **alors** $\mathtt{aux}\ n\ l{=}p :: (\mathtt{aux}\ n\ l')$

2. Dans cet ouvrage, nous suivons la tradition de certains langages fonctionnels et du lambda-calcul : l'application de la fonction f à l'argument x se note simplement « $f\ x$ » ; la notation « $g\ x\ y$ » est une abréviation de « $(g\ x)\ y$ » ; par ailleurs, les guillemets que nous utilisons pour séparer le texte du livre des expressions *Coq* ne font pas partie de ces expressions.

Remarque 2.1 La définition ci-dessus contient un test de comparaison des entiers n et p. Il faut bien comprendre que la possibilité de faire un tel test provient d'une propriété de `Z`, à savoir la décidabilité de l'ordre $\leq$. Autrement dit, on peut programmer une fonction de deux arguments n et p qui renvoie une certaine valeur si $n \leq p$, et une valeur différente si $n > p$. En *Coq*, cette propriété est représentée par un programme certifié de la bibliothèque standard, appelé `Z_le_gt_dec` (voir page 285.) Il faut bien voir que cette possibilité n'existe pas pour toute relation d'ordre, même totale. Prenons par exemple le type `nat`$\rightarrow$`nat` des fonctions de $\mathbb{N}$ vers $\mathbb{N}$, et considérons la relation suivante :

$$f < g \textbf{ si } \exists i \in \mathbb{N},\ f(i) < g(i)\ \wedge\ (\forall j \in \mathbb{N},\ j < i \Rightarrow f(j) = g(j))$$

Cette relation d'ordre total est indécidable, c'est à dire que nous ne pourrons pas écrire un programme certifié de comparaison similaire à `Z_le_gt_dec`. Par conséquent, nous ne pourrons pas trier des listes de fonctions pour cet ordre.[3]

Il reste à remarquer que la programmation de la fonction `nb_occ` renvoie au même type de problème, l'égalité de fonctions étant en général indécidable.

L'intérêt de la fonction `aux` tient en ces deux lemmes, qui se prouvent par récurrence sur l :

aux_equiv : $\forall l$: `list Z`, $\forall n$: `Z`, `aux` $n\ l \equiv n.l$

aux_sorted : $\forall l$: `list Z`, $\forall n$: `Z`, `sorted` $l \Rightarrow$ `sorted (aux` $n\ l$`)`

2.5.5 Le tri proprement dit

Il nous reste à construire un programme certifié de tri. Il s'agit donc d'associer à toute liste l une liste l' vérifiant (`sorted` l') $\wedge\ l \equiv l'$.

Nous procédons par récurrence sur l ;

- Si l est vide, alors prenons l' = `[]`.
- Sinon, posons $l = n.l_1$.
  - Soit l'_1 telle que (`sorted` l'_1) $\wedge\ l_1 \equiv l'_1$ (hypothèse de récurrence),
  - soit l' = `aux` $n\ l'_1$,
  - on a “ `sorted` l' ” (par le lemme `aux_sorted`), et
  - $l = n.l_1 \equiv n.l'_1 \equiv$ `(aux` $n\ l'_1$`)` $= l'$ (par les lemmes `aux_equiv` et `equiv_cons`).
  - Comme $\equiv$ est une relation d'équivalence, on obtient $l \equiv l'$.

Cette construction de l' à partir de l, avec ses justifications logiques, est obtenue par un dialogue avec le système *Coq*. Le résultat en est un terme de type `Z_sort`, c'est à dire un programme certifié de tri. Son extraction vers *OCAML* fournit un programme fonctionnel de tri sur les listes d'entiers. Voici le résultat que donne la commande `Extraction` de *Coq*[4] :

3. Ce type de problème n'est pas une caractéristique de *Coq* : si un langage de programmation quelconque vous fournit une primitive de comparaison de valeurs d'un type A muni d'une relation d'ordre, c'est que l'égalité y est décidable. *Coq* ne fait qu'expliciter cette problématique.

4. Ce programme *ML* utilise un type à deux constructeurs : `Left` et `Right`, isomorphe au type des booléens

```
let rec aux z0 = function
  | Nil -> Cons (z0, Nil)
  | Cons (a, l') ->
      (match z_le_gt_dec z0 a with
         | Left -> Cons (z0, (Cons (a, l')))
         | Right -> Cons (a, (aux z0 l')))

let rec sort = function
  | Nil -> Nil
  | Cons (a, l0) -> aux a (sort l0)
```

Cette possibilité d'obtenir mécaniquement un programme à partir d'une preuve de réalisabilité d'une spécification est extrêmement importante. Les preuves de programmes que nous pourrions faire au tableau ou sur papier seraient de toutes façons incomplètes (car trop longues à écrire), et même si elles étaient correctes, rien n'assurerait que le programme écrit à la main en *OCAML* soit identique au programme prouvé.

2.6 Apprendre *Coq*

Le système *Coq* est un outil informatique avec lequel on doit communiquer en utilisant un langage précis, comportant un certain nombre de commandes et de conventions syntaxiques. Le langage dans lequel sont écrits les termes — appelé *Gallina* — et le langage de commandes du système *Coq* — appelé le *Vernaculaire* — sont décrits de façon exhaustive dans le manuel de référence de *Coq* [83].

Notre objectif est de donner au lecteur une compréhension pratique de l'outil *Coq* et des concepts théoriques qui le sous-tendent, aussi avons-nous inclus de nombreux exemples de développements. À titre didactique, certains des exemples comportent des maladresses volontaires et sont accompagnés de principes guides permettant de les éviter.

Le système *Coq* est un outil interactif et tout travail avec cet outil est un dialogue que nous avons souvent essayé de transcrire dans cet ouvrage, pour guider l'utilisateur dans son apprentissage. Dans leur grande majorité, les exemples de développement que nous avons fourni correspondent à une utilisation bien comprise du système.

Nous avons souvent essayé de décomposer les dialogues de façon que l'utilisateur puisse simplement les reproduire, soit avec un papier et un crayon, soit avec l'outil *Coq* lui-même. Nous avons aussi parfois inclus des commandes ou des termes synthétiques pouvant impressionner en première lecture, mais ces termes ont aussi été obtenus avec l'aide de l'assistant *Coq*. Le lecteur est invité à décomposer les expressions les plus compactes dans ses expériences, à les modifier, et à se les approprier. Nous conseillons au lecteur de rejouer sur sa machine les exemples de ce livre, qu'il pourra trouver sur le site de cet ouvrage [11]. Un grand nombre d'exercices sont proposés, pour la plupart entièrement résolus sur ce site.

Coq se caractérise par une très grande puissance d'expression, tant dans le domaine du raisonnement que dans celui de la programmation. Son apprentissage passe donc par des niveaux trés variés, allant de constructions très simples ou entièrement automatiques au développement de théories complexes et de tactiques de démontration élaborées. Afin de permettre au lecteur de choisir ses niveaux de lecture, nous proposons une annotation des en-tête de chapitres ou de sections indiquant le niveau nécessaire de pratique du raisonnement logique ainsi que du système *Coq* lui même :

pas d'annotation accessible en première lecture,

signe * accessible après avoir écrit quelques preuves en *Coq*,

signe ** niveau « utilisateur avancé », possibilité de faire des raisonnements complexes et de certifier des progarammes,

signe *** « 5ème dan », intérêt pour l'exploration de toutes les possibilités de *Coq*, voire son évolution.

La même symbolique est utilisée aussi pour les exercices, allant de « élémentaire » (sans annotation) à « pouvant demander des jours de réflexion » (***). Le niveau des exercices est estimé relativement à celui du chapitre courant, et en tenant compte des progrés certains du lecteur.

L'équipe de développement de *Coq* maintient un site de contributions d'utilisateurs (`http://coq.inria.fr/contribs-eng.html`), où se trouvent un grand nombre de développements dans des domaines d'application très variés. Nous invitons le lecteur à s'y plonger régulièrement. Nous conseillons aussi fortement l'abonnement à la liste de mèl `coq-club@pauillac.inria.fr`, traitant de questions d'intérêt général sur l'implémentation, le formalisme logique, et les annonces de nouvelles applicatons de *Coq*.

Au delà de l'apprentissage de l'outil *Coq*, cet ouvrage est également une introduction pratique au cadre théorique de la théorie des types et plus particulièrement au *Calcul des Constructions Inductives* qui combine plusieurs des progrès effectués dans la compréhension de la logique au travers du λ-calcul et du typage et des fondements des mathématiques, selon une tradition qui remonte au *Principia Mathematicae* de Russell et Whitehead, en passant par les travaux de Peano, Nœther, Church, Curry, Prawitz et P. Aczel [4] ; le lecteur intéressé pourra avantageusement se référer au recueil compilé par J. van Heijenoort "From Frege to Gödel" [86].

2.7 Contenu de l'ouvrage

Le Calcul des Constructions

Les chapitres 3 à 5 sont une description du *Calcul des Constructions*. Le chapitre 3 présente le lambda-calcul simplement typé et sa relation avec la programmation fonctionnelle. Les notions importantes de termes, types, sortes, réductions y sont présentées, ainsi que la syntaxe des termes en *Coq*.

Le chapitre 4 aborde l'aspect logique de *Coq*, en présentant l'*Isomorphisme de Curry-Howard* ; cette présentation se fera dans un cadre restreint reliant le lambda-calcul simplement typé et la logique minimale propositionnelle. C'est également l'occasion de présenter la notion de *tactique*, outil facilitant la construction interactive de preuves. Les tactiques peuvent comporter un niveau élevé d'automatisation, présenté en chapitre 8.

La puissance d'expression du Calcul des Constructions : polymorphisme, types dépendants, types d'ordre supérieur est analysée dans le chapitre 5 consacré au produit dépendant. L'isomorphisme de Curry-Howard s'étend à cette construction en faisant se correspondre le produit dépendant et la quantification universelle. Le produit dépendant permet d'étendre les capacités de raisonnement de *Coq*, dont les aspects pratiques sont abordés dans le chapitre 6.

Constructions Inductives

Le Calcul des Constructions *Inductives* est abordé au chapitre 7, montrant comment définir des structures de données telles que les entiers naturels, listes et arbres. De nouveaux outils : tactiques de démonstration par récurrence, règles de simplification, etc. sont associés à ces types. La notion de type inductif n'est pas restreinte aux structures de données arborescentes : des prédicats peuvent être définis de cette façon, dans le style d'un *Prolog* d'ordre supérieur [5].

C'est également dans ce cadre (chapitre 9) que sont définies des notions de base de la logique : contradiction, conjonction, disjonction, quantification existentielle.

Programmes certifiés et extraction

Le chapitre 10 est consacré aux spécifications complexes contenant des composants logiques. Divers constructeurs de types permettant d'exprimer une grande variété de spécifications de programmes sont présentés, ainsi que les tactiques associées qui facilitent la construction de programmes certifiés.

Les chapitres 11 et 12 sont consacrés à la production effective de code *OCAML* par extraction à partir de preuves.

Utilisation avancée

Les derniers chapitres de ce livre présentent des aspects avancés de *Coq*, tant du point de vue de l'utilisation que de la compréhension de ses mécanismes.

Le chapitre 13 présente le système de modules de *Coq*, très fortement inspiré de celui d'*OCAML*. Ce système permet de représenter des dépendances entre théories mathématiques, mais aussi la construction de véritables composants de programmes certifiés réutilisables.

5. Rappelons que *Prolog* considère la logique du premier ordre. Nous ne pouvons pas — dans le Prolog de base — définir un opérateur sur des relations, comme par exemple la clôture transitive d'une relation binaire *quelconque*. En *Coq*, si.

Le chapitre 14 montre comment représenter des objets infinis — tels les comportements de systèmes de transitions — dans une extension du Calcul des Constructions. Nous montrons les techniques principales permettant la spécification et la construction de tels objets, ainsi que les techniques de preuves appropriées.

Le chapitre 15 approfondit les principes de base qui régissent les définitions inductives et assurent leur cohérence logique. Il fournit également les clefs pour les utilisations les plus avancées.

Le chapitre 16 décrit les techniques qui permettent d'élargir la classe des fonctions récursives que l'on sait écrire dans le Calcul des Constructions Inductives et qui permettent de raisonner sur ces fonctions.

Automatisation des preuves

Le chapitre 8 introduit les outils permettant la construction de preuves complexes : les tactiques. Ce chapitre traite des tactiques associées aux types inductifs, des tactiques de démonstration automatique, et des tactiques spécialisées pour les démonstrations numériques. L'insertion de ce chapitre au milieu du livre permet de traiter plus rapidement les démonstrations apparaissant dans les chapitres ultérieurs. Il introduit également un langage spécialisé pour l'écriture de nouvelles tactiques par l'utilisateur.

Le chapitre 17 décrit une technique de conception de tactiques élaborées particulièrement adaptée pour le Calcul des Constructions : la technique de démonstration par réflexion.

2.8 Conventions lexicales

Cet ouvrage contient de nombreux exemples de texte *Coq* et des réponses du système. Nous utiliserons les conventions classiques dans les livres sur la programmation : emploi de la police `typewriter` pour les citations de source informatique, et de la police *italique* pour simuler les réponses du système. Le système affiche systématiquement une invite en début de ligne et cette invite peut changer suivant les commandes envoyées par l'utilisateur. Nous l'omettrons systématiquement. Nous avons bien sûr respecté la syntaxe de *Coq*, mais avons procédé à quelques transformations mineures :

Les symboles mathématiques ‘$\leq$’, ‘$\neq$’, ‘$\exists$’, ‘$\forall$’, ‘$\rightarrow$’, ‘$\leftrightarrow$’, ‘$\vee$’, ‘$\wedge$’ et ’$\Rightarrow$’, remplacent respectivement les suites de caractères lexicalement correctes suivantes : ‘`<=`’, ‘`<>`’,‘`exists`’, ‘`forall`’, ‘`->`’, ‘`<->`’ ‘`\/`’, ‘`/\`’ et ‘`=>`’. Ainsi, l'énoncé ci-dessous :

```
forall A:Set,(exists x : A | forall (y:A), x <> y) -> 2 = 3
```

pourra être cité de la façon suivante :

```
∀A:Set,(∃x:A | ∀y:A, x ≠ y)→ 2 = 3.
```

Chaque fois qu'un extrait de texte *Coq* placé dans le texte courant risque d'être confondu avec ce dernier — notamment si l'extrait *Coq* n'est pas parenthésé et contient des espaces —, nous plaçons cet extrait entre des guillemets "...". Ces guillemets ne font pas partie de la syntaxe de *Coq*.

Pour finir, signalons que tout texte placé entre "`(*`" et "`*)`" est un commentaire ignoré par le système *Coq*.

Chapitre 3

Types et expressions

Une des principales utilisations de *Coq* est la certification et plus généralement le raisonnement sur les programmes. Il est donc important de montrer comment le langage *Gallina* représente ces programmes. Nous présentons dans ce chapitre une classe restreinte de programmes purement fonctionnels — c'est à dire sans affectation ni autres « effets de bord » — et sans récursivité ni polymorphisme ; le formalisme utilisé pour représenter ces programmes est le λ-calcul simplement typé [25]. Ce formalisme simplifié sera cependant présenté de façon à rendre naturelles les extensions présentées dans les chapitres à suivre et qui permettront non seulement le raisonnement logique, mais aussi la construction de spécifications et de programmes plus complexes.

Nous présentons dans un premier temps les notions d'expression, de type, de contexte et d'environnement, puis celles de *sorte* et d'*univers*, qui permettront de plonger ce formalisme simplifié dans le riche système de types du Calcul des Constructions autorisant notamment le polymorphisme, les constructions d'ordre supérieur, les types dépendants, ainsi que la construction des propositions et de leurs preuves.

Ce chapitre sera aussi l'occasion d'un premier contact avec l'outil *Coq*, pour en apprendre la syntaxe et quelques commandes permettant la vérification de types et l'évaluation d'expressions.

Les premiers exemples d'expressions que nous allons présenter utilisent des types connus de tous les programmeurs : entiers naturels, dits *de Peano*, entiers en représentation binaire, booléens. Il faut savoir que la construction de ces types et la preuve de leurs propriétés font appel aux techniques présentées à partir du chapitre 7. Afin de fournir au lecteur des exemples simples et familiers, nous considérons ces types et les constantes associées comme prédéfinis. D'autre part, la notion de *sorte* nous permettra de considérer des types *arbitraires*, premier pas vers le polymorphisme.

3.1 Premiers pas

Notre premier contact avec *Coq* se fait en activant une boucle d'interaction (*top-level* en anglais), par l'intermédiaire de la commande `coqtop`. L'usager interagit avec le système au moyen d'un langage de commandes appelé le *Vernaculaire Coq*. Notons que toute commande *Coq* doit être suivie d'un point et d'un espace.

Le court dialogue suivant nous permet d'introduire la commande `Require` dont les arguments sont un indicateur (ici `Import`) et un nom de module ou de bibliothèque à charger. Les bibliothèques que nous chargeons contiennent des définitions, théorèmes et notations spécifiques concernant respectivement l'arithmétique de Peano, l'arithmétique des nombres entiers, et les valeurs booléennes. Le chargement de ces bibliothèques affecte un composant de *Coq* appelé l'*environnement*, que l'on peut considérer comme une table mémorisant les déclarations et définitions de constantes.

Le dialogue suivant montre le lancement de *Coq* depuis un interprête de commandes, puis le chargement des bibliothèques contenant les définitions, notations et résultats principaux pour les entiers naturels, les nombres entiers et les valeurs booléennes :

```
machine prompt %  coqtop
 Welcome to Coq 8.0 (Oct 2003)
 Require Import Arith.
 Require Import ZArith.
 Require Import Bool.
```

3.1.1 Termes, expressions, types, etc.

La notion de *terme* constitue une catégorie syntaxique très générale dans le langage de spécification *Gallina*, et correspond à la notion intuitive « d'expression bien formée prenant en compte les règles de construction du langage ». Nous reviendrons plus précisément sur ces règles par la suite. Dans ce chapitre, nous considérerons particulièrement deux sortes de termes, les *expressions*, correspondant *grosso modo* aux programmes d'un langage fonctionnel, et les *types* qui permettent de contrôler la bonne formation des programmes et le respect de leur spécification. Ce dernier rôle dévolu aux types dans les langages de programmation nous autorise à les qualifier également de *spécifications*.

3.1.2 Notion de portée

Le langage mathématique, ainsi que la plupart des langages de programmation, utilisent des conventions pour simplifier l'écriture des expressions. Ces conventions sont définies en *Coq* à l'aide de la notion de *portée* (*scope* en anglais). Une portée permet d'associer une interprétation à un ensemble de notations donné. Par exemple, une portée permet d'associer au symbole '`*`' la multiplication des entiers naturels, une deuxième portée lui associe la multiplication de nombres réels, une troisième le produit cartésien de deux types.

Coq permet d'*ouvrir* (c'est à dire rendre actives) simultanément plusieurs portées, chacune permettant d'interpréter un ensemble de notations. Ces portées sont placées sur une pile contenant au départ une portée minimale appelée `core_scope` définissant les notations de base de *Gallina*. Par ailleurs, dès que l'analyse syntaxique attend un type, une portée contenant les notations spécifiques aux types, et appelée `type_scope` est empilée automatiquement.

La commande pour empiler une portée est " `Open Scope` *`portée`* ". Si deux portées définissant la même notation (par exemple l'opérateur '`*`' de multiplication des nombres entiers et le produit cartésien) sont ouvertes, la portée la plus récemment ouverte masque la plus ancienne.

Il peut être utile d'empiler une portée pour une seule expression. La syntaxe est alors " *exp%k* ", où *k* est un symbole associé à la portée à ouvrir ("*delimiting key*"). Cette facilité permet alors de disposer de plusieurs conventions d'écriture au sein d'une même commande, par exemple si une expression contient à la fois des nombres entiers et des nombres réels.

La commande `Locate` permet de connaître à tout moment quelles interprétations sont associées à une notation donnée. Dans l'exemple suivant, nous demandons quelles sont les interprétations de l'opérateur infixe '`*`' :

```
Open Scope Z_scope.

Locate "_ * _".
...
"x * y" := prod x y : type_scope
"x * y" := Ring_normalize.Pmult x y : ring_scope
"x * y" := Pmult x y : positive_scope
"x * y" := mult x y : nat_scope
"x * y" := Zmult x y : Z_scope (default interpretation)
```

Ce dialogue montre que, par défaut, la notation "`x * y`" doit s'interpréter comme l'application de la fonction `Zmult` (multiplication des nombres entiers), selon la portée `Z_scope`.

La commande " `Print Scope` " permet de connaître toutes les notations définies par une portée, ainsi que la clef associée (champ "`Delimiting key`") :

```
Print Scope Z_scope.
Scope Z_scope
Delimiting key is Z
Bound to class Z
"- x" := Zopp x
"x * y" := Zmult x y
"x + y" := Zplus x y
"x - y" := Zminus x y
"x / y" := Zdiv x y
"x < y" := Zlt x y
```

"x < y < z" := and (Zlt x y)(Zlt y z)
"x < y <= z" := and (Zlt x y)(Zle y z)
"x <= y" := Zle x y
"x <= y < z" := and (Zle x y)(Zlt y z)
"x <= y <= z" := and (Zle x y)(Zle y z)
"x > y" := Zgt x y
"x >= y" := Zge x y
"x ?= y" := Zcompare x y
"x ^ y" := Zpower x y
"x 'mod' y" := Zmod x y

La clef de portée (en anglais *delimiting key*) associée à une portée permet de limiter l'utilisation d'une portée à un fragment d'expressions à l'intérieur d'une expression plus large. La convention est d'écrire d'abord l'expression (entre parenthèses s'il s'agit d'une expression composée) suivie du caractère %, suivi par la clef. Avec les clefs de portée, nous pouvons utiliser plusieurs jeux de notations dans la même expression, par exemple lorsque cette expression contient à la fois des nombres entiers et des nombres réels.

3.1.3 Contrôle de type

La commande " `Check` t " permet de connaître le type d'un terme t dans l'environnement courant déterminé par les déclarations et définitions préalablement effectuées ; si l'argument de `Check` ne respecte pas les règles syntaxiques de *Gallina* ou n'a pas de type, un message d'erreur approprié est affiché. Commençons par essayer cette commande sur quelques constantes simples.

Entiers naturels

Le type associé aux entiers naturels est appelé `nat`, le nombre zéro est décrit par l'identificateur `O` (il s'agit bien de la lettre majuscule O).

Les entiers naturels peuvent s'écrire en notation décimale à l'aide de la portée `nat_scope`, à laquelle est associée la clef `nat`. Par conséquent, en dehors d'une portée `nat_scope`, l'entier naturel n se note n`%N`, et n à l'intérieur d'une telle portée.

```
Check 33%nat.
```
33%nat : nat

```
Check 0%nat.
```
0%nat : nat

```
Check O.
```
0%nat : nat

```
Open Scope nat_scope.
```

```
Check 33.
```
33 : nat

```
Check 0.
```
0 : nat

Nombres entiers

Le type `Z` est associé aux nombres entiers : l'ensemble $\mathbb{Z}$ des mathématiciens ou le type `int` de nombreux langages de programmation [1]. La portée `Z_scope` est associée aux nombres entiers et autorise leur écriture décimale. On notera la présence de parenthèses autour des nombres négatifs [2]. De plus, si l'on ouvre la portée `Z_scope`, les entiers naturels devront être décrits à l'aide du suffixe %N, et réciproquement.

```
Check 33%Z.
```
33%Z : Z

```
Check (-12)%Z.
```
(-12)%Z : Z

```
Open Scope Z_scope.
```

```
Check (-12).
```
-12 : Z

```
Check (33%nat).
```
33%nat : nat

Le système de typage de *Gallina* n'a pas de notion d'inclusion de types ; le type `nat` n'est donc pas inclus dans le type `Z`, et la conversion d'un entier naturel en un nombre entier devra faire appel à des fonctions de conversion explicite.

Valeurs booléennes

Le type `bool` contient deux constantes associées aux valeurs de vérité :

```
Check true.
```
true : bool

```
Check false.
```
false : bool

1. Du moins les langages considérant des nombres entiers non bornés.
2. Comme toute commande de *Coq* ayant un terme comme argument, `Check` attend une expression atomique, c'est à dire soit mise entre parenthèses, soit de la forme *e%k*, soit réduite à une constante. L'expression " -273 " est en fait une expression composée, obtenue en appliquant l'opérateur unaire '-' à l'entier 273.

3.2 Les règles du jeu

Nous présentons ci-dessous les règles permettant de construire un sous-ensemble de *Gallina* correspondant à un λ-calcul simplement typé. Ces règles définissent à la fois la syntaxe des termes (types et expressions) et les contraintes permettant de typer les expressions, ainsi que les notions de variable, constante, déclaration et définition.

3.2.1 Types simples

Un cadre simple pour commencer l'étude de *Coq* nous est fourni par le λ-calcul simplement typé sans polymorphisme, un modèle de langage de programmation d'expressivité très réduite. Les types considérés sont de deux formes :

– Des types *atomiques*, réduits à un seul identificateur, comme `nat`, `Z` et `bool`,
– Des types de la forme[3] $A{\rightarrow}B$, où A et B sont deux types ; les types de cette forme sont appelés des *types flèche*. Le système *Coq* utilise en fait la chaine de caractères "`->`" pour représenter la flèche, mais nous utiliserons systématiquement la meilleure typographie fournie par le symbole $\rightarrow$ dans cet ouvrage. Nous verrons dans le chapitre 5 que ces types forment un cas particulier d'une construction très générale appelée *produit dépendant*.

Les types ainsi considérés s'apparentent aux types fonctionnels de *ML* formés à partir de types de base et de l'opérateur $\rightarrow$. Le type $A{\rightarrow}B$ est celui des fonctions prenant un argument de type A et retournant un résultat de type B. Rappelons que par « fonction » nous entendons « procédé effectif de calcul » et non une notion ensembliste. Par exemple, nous ne confondrons pas une fonction naïve de calcul de x^n (demandant n multiplications) et un algorithme d'exponentiation binaire, bien qu'ils retournent toujours le même résultat.

De plus, toutes les fonctions considérées en *Coq* sont telles que leur application à un argument fournit toujours un résultat au bout d'un calcul fini.

Conventions syntaxiques Dans le cas de types comportant plusieurs occurrences de $\rightarrow$, nous pouvons utiliser des parenthèses pour éviter les problèmes d'ambiguïté. Par exemple, le type d'une fonctionnelle transformant toute fonction unaire sur `Z` en une fonction unaire sur `nat` peut s'écrire

$$\texttt{(Z}{\rightarrow}\texttt{Z)}{\rightarrow}\texttt{(nat}{\rightarrow}\texttt{nat)}.$$

Nous pouvons cependant éviter l'abus de parenthèses en abrégeant un type de la forme $A{\rightarrow}(B{\rightarrow}C)$ en $A{\rightarrow}B{\rightarrow}C$. Ce type est celui des fonctions à un argument de type A qui retournent des fonctions de type `B→C`, mais on le lira plus aisément

3. On notera ici le premier emploi d'une convention que nous utiliserons fréquemment : des termes ou des commandes généralisés, respectant la syntaxe de *Gallina* mais contenant des variables en italique dénotent des constructions arbitraires ; ces variables sont souvent appelées « méta-variables » dans la littérature informatique, afin de les distinguer des variables propres au langage dont on fait la description. Ainsi la notation $A{\rightarrow}B$ désigne l'infinité de types *Coq* obtenus en remplaçant A et B par des types quelconques.

comme le type des fonctions à deux arguments de type A et B qui retournent des valeurs de type C. Par exemple, le type de la fonction `Zplus` d'addition sur `Z` est `Z→Z→Z` c'est à dire `Z→(Z→Z)`.

Cette convention s'étend à un nombre quelconque de types ; par exemple, le type de la fonction `ifb` de la bibliothèque `Bool` peut s'écrire des deux façons suivantes, la seconde étant la plus fréquemment utilisée :

- `(bool→(bool→(bool→bool)))`
- `bool→bool→bool→bool`

La structure des types simples peut se représenter graphiquement sous la forme d'arbres binaires. La figure 3.1 page 67 montre la représentation sous forme d'arbre des types `bool→bool→bool→bool` et `(Z→Z)→nat→nat`.

3.2.2 Identificateurs, environnements, contextes, etc.

Comme dans la plupart des langages de programmation, il est d'usage de distinguer les notions de *définition* et *déclaration*, ainsi que leur portée *locale* ou *globale*.

Une déclaration permet d'attacher un type à un identificateur, sans lui donner de valeur. Comme dans les fichiers d'interface de *C* ou *Java*, ou les signatures de *ML*, on peut déclarer que, par exemple, la variable `max_int` est de type `Z`, sans lui attacher de valeur précise. La déclaration d'un identificateur x de type A se note $(x : A)$.

En revanche, une définition donne une valeur à un identificateur sous la forme d'un terme associé. Dans la mesure où l'on peut déterminer le type de ce terme, une définition précise à la fois le type et la valeur d'un identificateur. La définition d'un identificateur x par un terme t de type A se note $(x := t : A)$.

Par conséquent, toute définition peut aussi jouer le rôle d'une déclaration, et un énoncé concernant une déclaration quelconque s'applique donc à une définition en « oubliant » la « valeur » qu'elle précise.

La portée d'une déclaration ou d'une définition peut être, soit *globale*, c'est à dire tout le reste du développement, soit *locale*, c'est à dire restreinte à une sous-expression ou à une *section* (nom donné en *Coq* à une structure similaire aux blocs des langages de programmation). À tout point d'un développement en *Coq* sont associés à la fois un environnement et un contexte dits *courants*. Nous réservons le nom d'*environnement* à une suite de *déclarations* ou *définitions globales*, et celui de *contexte* à une suite de déclarations ou définitions *locales*. Au démarrage d'une session, nous disposons d'un environnement *initial* et d'un contexte vide. La commande “ `Reset Initial` ” permet de revenir à cet état (et donc d'effacer toute définition ou déclaration faite depuis le démarrage). De façon plus générale, la commande “ `Reset` *id* ” efface de l'environnement toute définition ou déclaration faite à partir de *id*.

Nous utiliserons fréquemment le nom de *variable* pour qualifier un identificateur déclaré ou défini localement, de *constante* pour un identificateur défini globalement, de *variable globale* pour un identificateur déclaré globalement. Puisqu'une définition peut être considérée comme une déclaration, le terme générique de « variable » inclura également les constantes.

Sans entrer dans les détails, nous supposerons toujours qu'environnements et contextes sont *bien formés*, autrement dit toute déclaration ou définition porte sur une variable nouvelle (ni déclarée ni définie préalablement) et utilise des types ou des termes eux mêmes bien formés dans le contexte qui les précède.

On peut alors supposer qu'environnement et contexte sont disjoints (c'est à dire déclarent des symboles différents). D'un point de vue pratique, cette disjonction peut se réaliser, soit en signalant une erreur, soit en *masquant* (en rendant inaccessible) la déclaration la plus globale.

Notations

Les notations qui suivent ne font pas à proprement partie de *Gallina*, mais sont indispensables pour décrire certaines règles et mécanismes de *Coq*. Dans la mesure du possible, nous présenterons des notations simplifiées ; pour un formalisme complet, la référence absolue reste la présentation du Calcul des Constructions Inductives dans le manuel de référence [83].

- Nous emploierons les symboles E et Γ (avec des indices ou décorations éventuels) pour désigner respectivement un environnement ou un contexte arbitraire.
- Nous utilisons la notation « [] » pour dénoter le contexte vide ne déclarant aucune variable locale. C'est en particulier le contexte courant associé à une partie de développement à l'extérieur de toute section.
- Un contexte peut se présenter sous la forme d'une suite de déclarations ou définitions présentée comme ci-dessous :

  $$[v_1 : A_1;\ v_2 := t_2 : A_2;\ \ldots;\ v_n : A_n]$$

  L'adjonction d'une déclaration ou d'une définition d à un contexte Γ se note $\Gamma :: d$.
- Pour exprimer le fait que la variable v est spécifiée de type A dans le contexte Γ, on utilise la notation $(v : A) \in \Gamma$; des variantes sont également utilisées : $v \in \Gamma$ (v est déclarée dans Γ sans plus de précision), $(v : A) \in E \cup \Gamma$ (déclaration soit locale soit globale), etc.
- *Jugement de typage :* La notation $E, \Gamma \vdash t : A$ — où E, Γ, t et A dénotent respectivement un environnement, un contexte, un terme et un type — se lit « Dans l'environnement E et le contexte Γ le terme t a pour type A ».
- Pour finir — et ceci est une notation propre à ce chapitre —, appelons E_0 l'environnement obtenu après chargement des bibliothèques `Arith`, `ZArith` et `Bool`.

Définition 3.1 (Types habités) On dira qu'un type A est *habité* dans un environnement E et un contexte Γ s'il existe un terme t pour lequel le jugement $E, \Gamma \vdash t : A$ est valide.

3.2.3 Les expressions et leur type

Une fois présentées les notions de spécification, de contexte et d'environnement, nous pouvons définir la syntaxe et les règles de typage de notre langage fonctionnel simplifié.

Expressions réduites à un identificateur

La forme syntaxique la plus simple d'expression est une simple variable ou une constante x. Un tel terme ne peut être accepté que si l'identificateur x est déclaré dans le contexte ou l'environnement courant. Soit A le type de x dans cette déclaration ; alors le terme réduit à x a pour type A.

Il est d'usage de présenter une règle de typage sous forme d'une règle d'inférence ; les prémisses sont placées au dessus d'une barre horizontale, et la conclusion en dessous de cette barre.

$$\textbf{Var}\quad \frac{(x, A) \in E \cup \Gamma}{E, \Gamma \vdash x : A}$$

Cette règle se lit : « si l'identificateur x est spécifié de type A dans l'environnement E ou le contexte Γ, alors le terme x a pour type A dans cet environnement et ce contexte ». Elle s'applique dans les exemples de la section 3.1.3 avec les constantes `O:nat`, `true:bool` et `false:bool`.

D'autres exemples peuvent nous être donnés par l'addition sur `nat`, sur `Z`, la négation et la disjonction sur les booléens :

```
Check plus.
```
plus : nat→nat→nat

```
Check Zplus.
```
Zplus : Z→Z→Z

```
Check negb.
```
negb : bool→bool

```
Check orb.
```
orb : bool→bool→bool

L'échange ci-dessous illustre la nécessité de déclarer ou définir tout identificateur avant son utilisation :

```
Check zero.
```
...
Error: The reference zero was not found in the current environment

Applications

L'application d'une fonction à un argument est la principale structure de contrôle de notre langage. Nous en présentons d'abord la syntaxe, accompagnée de règles assurant la cohérence des expressions ainsi formées.

Soient un environnement E et un contexte Γ ; soient deux expressions e_1 et e_2 de types respectifs $A{\rightarrow}B$ et A dans $E \cup \Gamma$; alors *l'application de* e_1 *à* e_2 est le terme s'écrivant " $e_1\ \ e_2$ " ; ce terme est de type B dans le contexte et l'environnement considérés.

Dans l'expression " $e_1\ \ e_2$ ", e_1 est *en position fonctionnelle* et e_2 est l'*argument* de l'application. La présentation sous forme de règle d'inférence est la suivante :

$$\textbf{App}\quad \frac{E,\Gamma \vdash e_1 : A{\rightarrow}B \quad E,\Gamma \vdash e_2 : A}{E,\Gamma \vdash e_1\ \ e_2 : B}$$

Par exemple, à l'aide du type des constantes `true` et `negb`, et en considérant l'environnement E_0 (voir section 3.2.2) et le contexte vide, la règle **App** nous permet de déduire que l'expression " `negb true` " (pouvant s'abréger en `!!true`) a pour type `bool` ; nous pouvons itérer ce raisonnement et construire des expressions plus complexes :

```
Check negb.
```
negb : bool→bool

```
Check (negb true).
```
negb true : bool

```
Check (negb (negb true)).
```
negb (negb true) : bool

La syntaxe de *Coq* (à partir de la version 8) associe à l'application fonctionnelle un opérateur « associant à gauche » de très forte priorité. Ainsi, une cascade d'applications de la forme " `(`...`(`$f\ \ t_1$`)` t_2`)` ... t_n " pourra s'écrire simplement " $f\ \ t_1\ \ t_2\ \ \ldots\ \ t_n$ ". Cette convention permet d'éviter une trop grande abondance de parenthèses. Celles-ci ne sont alors nécessaires que pour expliciter une structuration en sous-termes.

L'exemple ci-dessous montre que *Coq* respecte ces conventions, y compris lors de l'impression de termes :

```
Check (((ifb (negb false)) true) false).
```
ifb (negb false) true false : bool

Dans l'exemple suivant, en revanche, l'absence de parenthésage interne conduit à considérer que l'unique argument de la première occurrence de `negb` est la seconde occurrence de cette constante.

```
Check (negb negb true).
```
Error: The term negb has type bool→bool
while it is expected to have type bool

Les règles d'écriture des types flèche et des applications coopèrent pour donner à l'utilisateur l'impression de travailler avec des fonctions à plusieurs variables. Ceci peut être résumé sous la forme d'une « règle dérivée » de **App** :

$$\textbf{App*}\quad \frac{E,\Gamma \vdash e : A_1{\rightarrow}A_2{\rightarrow}\ldots{\rightarrow}A_n{\rightarrow}B \quad E,\Gamma \vdash e_i : A_i\ \ (i = 1 \ldots n)}{E,\Gamma \vdash e\ e_1\ e_2 \ldots e_n : B}$$

Les portées `nat_scope`, `Z_scope`, etc., permettent d'éviter dans de nombreux cas la notation trop uniforme d'application de fonction, et de se rapprocher des notations usuelles des mathématiques et des langages de programmation usuels. Nous présentons ces notations pour les types `nat` et `Z`.

Entiers naturels Le type `nat` est construit suivant le modèle de Peano. Tout nombre naturel peut être obtenu, soit en prenant le nombre `O`, soit en appliquant la fonction successeur à un nombre déjà construit. Cette fonction est associée à la constante `S` de type `nat`→`nat`. Le nombre naturel n est donc représenté par le terme $\underbrace{\texttt{S(S(S(...(S(}}_{n}\texttt{O))...))}$. La portée `nat_scope` permet de représenter ces n applications imbriquées par le simple nombre n en numération décimale.

```
Open Scope nat_scope.

Check (S (S (S O))).
```
3 : nat

Cette portée nous autorise également l'emploi d'opérateurs infixes `+`, `-`, `*` pour représenter les applications de `plus`, `minus`, `mult` :

```
Check (mult (mult 5 (minus 5 4)) 7).
```
5(5-4)*7 : nat*

```
Check (5*(5-4)*7).
```
5(5-4)*7 : nat*

Il ne s'agit là que d'une facilité d'écriture. Tout nombre naturel reste représenté de façon interne comme la répétition d'applications de la fonction `S` et les opérations sont obtenues par l'application de fonctions binaires.

```
Unset Printing Notations.
Check 4.
```
S (S (S (S O))) : nat

```
Check (5*(5-4)*7).
```
```
mult (mult (S (S (S (S (S O)))))
            (minus (S (S (S (S (S O))))) (S (S (S (S O))))))
     (S (S (S (S (S (S (S O)))))))
     : nat
```

```
Set Printing Notations.
Check (minus (S (S (S (S (S O))))) (S (S (S (S O))))).
```
5 - 4 : nat

Nombres entiers La portée `Z_scope` a une utilisation similaire à celle de `nat_scope`. Les exemples suivants montrent des expressions contenant des applications de `Zplus`, `Zmult`, `Zopp`, etc., en forme infixe.

```
Open Scope Z_scope.
Check (Zopp (Zmult 3 (Zminus (-5)(-8)))).
```
-(3(-5--8)) : Z*

```
Check ((-4)*(7-7)).
```
-4(7-7) : Z*

Exemples

Pour chacune des interactions suivantes, il est possible de reconstituer la suite d'applications des règles **Var**, **App** et **App*** conduisant au diagnostic de typage effectué par *Coq*. On remarquera également les applications de `plus` et `Zplus` à un seul argument, dénotant respectivement la fonction qui ajoute 3 à tout entier naturel, et celle qui ajoute -5 à tout nombre entier. Dans ce dernier exemple, nous voyons que le type de la fonction `Zmult` impose que son premier argument soit de type `Z`, ce qui permet à *Coq* d'ouvrir automatiquement et silencieusement la portée `Z_scope` pour cet argument ; il en est de même pour la fonction `Zabs_nat`.

```
Open Scope nat_scope.

Check (plus 3).
```
plus 3 : nat→nat

```
Check (Zmult (-5)).
```
Zmult (-5) : Z→Z

```
Check Zabs_nat.
```
Zabs_nat : Z→nat

```
Check (5 + Zabs_nat (5-19)).
```
5 + Zabs_nat (5-19) : nat

Dans l'exemple suivant, le terme “ `mult 3%N` ” a pour type `nat→nat` et ne peut donc recevoir comme argument le terme `(-45)%Z` de type `Z`. Ce non-respect des règles de typage est alors signalé par un message d'erreur approprié :

```
Check (mult 3 (-45)%Z).
```
Error: The term -45%Z has type Z while it is expected to have type nat

Exercice 3.1 Reconstituer « à la main » le travail de vérification de type sur les exemples précédents.

Abstractions

La λ-abstraction (ou plus simplement *l'abstraction*) est un moyen simple de construire des fonctions en associant un paramètre formel et une expression. La fonction qui à tout v de type A associe l'expression e se note " `fun` $(v{:}A)$ $\Rightarrow$ e " ou " `fun` $v{:}A$ $\Rightarrow$ e " en *Gallina* [4]. Dans les présentations de λ-calculs typés, cette construction s'écrit classiquement sous la forme $\lambda v : A\,.\,t$ (également $\lambda v^A\,.\,t$). Dans le langage *OCAML*, l'abstraction sert à construire des fonctions anonymes fréquemment utilisées comme arguments ou valeurs de retour d'autres fonctions, que l'on appelle alors des *fonctionnelles*.

Par exemple, la fonction qui à n de type `nat` associe son cube n^3 se note respectivement en *Coq*, en notation classique, et finalement en *OCAML* des façons suivantes :

gallina : `fun n:nat ⇒ (n*n*n)%nat`

λ-calcul typé : $\lambda\, n^{\texttt{nat}}.\, n^3$ ou $\lambda\, n : \texttt{nat}.\, n^3$

***OCAML* :** `fun (n:nat) -> n*n*n`

La règle de typage de l'abstraction est donnée ci-dessous :

$$\textbf{Lam}\quad \frac{E, \Gamma :: (v : A) \vdash e : B}{E, \Gamma \vdash \texttt{fun}\ v{:}A\ \Rightarrow\ e : A \to B}$$

On notera que le contexte apparaissant dans la prémisse de cette règle est de la forme $\Gamma :: (v : A)$; nous supposons que v n'est pas déclaré dans Γ.

Exemple Reprenons l'expression " `fun n:nat ⇒ (n*n*n)%nat` ". Afin de déterminer son type dans l'environnement initial et le contexte vide, il suffit de déterminer le type de son corps `(n*n*n)%nat` dans le contexte déclarant son paramètre formel, à savoir $\Gamma_1 = $ `[n:nat]`. Comme le type de `(n*n*n)%nat` dans Γ_1 est `nat`, nous en déduisons que le type de l'abstraction entière dans le contexte vide est le type flèche `nat`$\to$`nat`.

Facilités d'écriture De même que les applications, les imbrications d'abstractions peuvent s'écrire de façon très concise ; par exemple, les écritures suivantes sont équivalentes :

```
fun n:nat ⇒ fun p:nat ⇒ fun z:Z ⇒ (Z_of_nat(n+p)+z)%Z

fun n p:nat ⇒ fun z:Z ⇒ (Z_of_nat(n+p)+z)%Z

fun (n p:nat)(z:Z) ⇒ (Z_of_nat(n+p)+z)%Z
```

Dans cet ouvrage, il nous arrivera parfois d'utiliser la notation classique " $\lambda v : A\,.\,t$ " au lieu de la notation *Gallina* pour décrire une abstraction.

Exercice 3.2 Déterminer le type de l'expression suivante :

`fun a b c:Z ⇒ (b*b-4*a*c)%Z.`

4. Sur un clavier, le symbole '⇒' se note '`=>`'

Exercice 3.3 La composée de deux fonctions f et g de type `nat→nat` est la fonction qui à tout n de type `nat` associe $g(f(n))$. La composition de fonctions sur `nat` est donc définie par l'expression :

```
fun (f g:nat→nat)(n:nat) ⇒ g (f n)
```

Quel est le type de cette expression ?

Inférence de types Le typage explicite d'une λ-abstraction peut être omis dans certains cas où le contexte de cette expression suffit à le déterminer. Dans ces cas, l'utilisateur peut écrire une abstraction sous la forme “ `fun` x ⇒ t ”. Dans l'exemple ci-dessous, la connaissance du type de `plus` et de `Zabs_nat` (c'est à dire `Z→nat`) est suffisante pour en déduire les types de `n` et `f` :

```
Check (fun n (z:Z) f ⇒ (n+(Zabs_nat (f z)))%nat).
```
fun (n:nat)(z:Z)(f:Z→Z) ⇒ n + Zabs_nat (f z)
: nat→Z→(Z→Z)→nat

Remarquons que, dans certains cas, les informations contenues dans le terme fourni par l'utilisateur sont insuffisantes pour lever des indéterminations. Dans l'exemple suivant, aucune information ne permet de déduire le type de `x` et, partant, de `f` :

```
Check (fun f x ⇒ Zabs_nat (f x x)).
```
Error: Cannot infer a type for f

Dans l'exemple suivant, le typage de l'application “ `x x` ” par la règle **App** conduit à une impossibilité : x serait d'un type pouvant se mettre à la fois sous les formes B et $B\rightarrow A$; ce type serait donc un terme infini, ce qui n'est pas accepté par la théorie à la base de *Coq*.

```
Check (fun x ⇒ x x).
```
Error: Occur check failed: tried to define ?4 with term ?4→?5

Variable anonyme Dans une abstraction “ `fun` $v{:}A$ ⇒ t ”, il peut arriver que la variable v n'ait aucune occurrence libre dans t. Dans ce cas, on peut utiliser la variable anonyme “`_`” au lieu de v.

```
Check (fun n _:nat ⇒ n).
```
fun n _:nat ⇒ n : nat→nat→nat

Le système *Coq* remplace automatiquement par une variable anonyme toute variable n'apparaissant pas dans le corps d'une abstraction, comme le montre l'exemple suivant :

```
Check (fun n p:nat ⇒ p).
```
fun _ p:nat ⇒ p : nat→nat→nat

Liaisons locales

La liaison locale (appelée `let-in` dans le manuel de référence) est une construction présente également dans les langages de type *Lisp* et de la famille *ML*. Elle sert à éviter la duplication de code et de calculs par l'utilisation de variables liées à des résultats intermédiaires.

Une liaison locale a pour forme “ `let` v`:=`e `in` e_1 ”, où v est une variable et e et e_1 sont deux expressions.

La règle **Let-in** définit les contraintes de typage de cette forme d'expression (on suppose que la variable v n'est pas déclarée dans Γ) :

$$\textbf{Let-in}\quad \frac{E,\Gamma \vdash t_1 : A \quad E,\Gamma :: (v : A) \vdash t_2 : B}{E,\Gamma \vdash \texttt{let}\ v\texttt{:=}t_1\ \texttt{in}\ t_2 : B}$$

L'exemple ci-dessous comporte deux définitions locales imbriquées servant à exprimer la fonction $\lambda\, n\, p\ .\, (n-p)^2((n-p)^2+n)$ en partageant au maximum les sous-termes ; à titre d'exercice, on pourra déterminer le type de cette expression.

```
fun n p : nat ⇒
   (let diff := n-p in
    let square := diff*diff in
        square * (square+n))%nat
```

Exercice 3.4 Combien d'instances des règles de typages sont nécessaires pour vérifier que cette expression est bien typée ?

3.2.4 Occurrences libres et liées ; α-conversion

La notion de liaison de variable est trop classique en mathématiques, logique et programmation pour devoir être développée ici ; contentons nous de quelques brefs rappels.

Les liaisons de variables sont introduites par les constructions “ `fun` `(`v`:`A`)` $\Rightarrow$ t ” et “ `let` v`:=`t `in` t' ” ; la *portée* de la variable v est dans le premier cas le terme t, et t' dans le second cas ; une occurrence d'une variable v dans un terme est *libre* si elle n'est pas dans la portée d'une liaison pour v, *liée* dans le cas contraire.

Par exemple, considérons le terme t_1 ci-dessous :

```
Definition t1 :=
 fun n:nat ⇒ let s := plus n (S n) in mult n (mult s s).
```

Toutes les occurrences des variables `S`, `plus` et `mult` dans t_1 sont libres, les occurrences de `n` sont liées par l'abstraction “ `fun n:nat ⇒ ...` ”, et celles de `s` par la liaison locale “ `let s:=plus n (S n) in ...` ”.

Tout comme en logique et en mathématiques, on peut changer le nom d'une variable liée dans une abstraction ou une liaison locale, en remplaçant par exemple le terme “ `fun` `(`v`:`A`)` $\Rightarrow$ t ” par “ `fun` `(`v'`:`A`)` $\Rightarrow$ t' ”, où t' s'obtient en remplaçant toutes les occurrences libres de v dans t par v', *à condition que*

v' n'ait pas d'occurrence libre dans t et que t ne contienne aucun sous-terme de la forme " `fun` v'`:`B $\Rightarrow$ t'' *" ou "* `let` v'`:=`e `in` t'' *" tel que v apparaisse libre dans t''*. Cette formulation est assez complexe, mais l'utilisateur n'a pas à s'en préoccuper, car le système *Coq* veille au grain.

Le renommage de variables liées respectant la condition ci-dessus s'appelle *α-conversion* ; cette transformation s'applique de la même façon aux liaisons locales (*let-in.*)

Par exemple, le terme t_2 ci-dessous s'obtient à partir de t_1 par α-conversion (et réciproquement).

```
fun i : nat ⇒
   let sum := plus i (S i) in mult i (mult sum sum).
```

A contrario, si nous remplaçons sans précaution dans t_1 la variable `s` par `n` — qui possède une occurrence libre dans le terme " `mult n (mult s s)` " —, nous ne respectons pas les restrictions associées à l'α-conversion. Nous remarquons dans ce cas que, pour respecter les règles de construction de termes, *Coq* renomme la liaison la plus interne, ce qui met en valeur la différence de structure avec t_1 et t_2.

```
fun n : nat ⇒
  let n := plus n (S n) in mult n (mult n n).
```

L'α-conversion induit une congruence $\cong_\alpha$ sur l'ensemble des termes : c'est à dire que $\cong_\alpha$ est une relation d'équivalence compatible avec la structure des termes :

si $t_1 \cong_\alpha t'_1$ **et** $t_2 \cong_\alpha t'_2$**, alors** $t_1\ \ t_2 \cong_\alpha t'_1\ \ t'_2$**,**
si $t \cong_\alpha t'$**, alors** `fun` $(v\!:\!A)$ $\Rightarrow$ $t \cong_\alpha$ `fun` $(v\!:\!A)$ $\Rightarrow$ t'**,**
si $t_1 \cong_\alpha t'_1$ **et** $t_2 \cong_\alpha t'_2$**, alors** `let` v`:=`t_1 `in` $t_2 \cong_\alpha$ `let` v`:=`t'_1 `in` t'_2

Dans la suite de cet ouvrage, nous considérerons comme égaux deux termes congruents par $\cong_\alpha$.

3.3 Déclarations et définitions

Nous rappelons que l'environnement courant est la suite de toutes les déclarations et définitions de variables globales faites depuis le chargement du système *Coq* (environnement initial), entre autres par le chargement de bibliothèques. Nous étudions les moyens d'étendre contexte et environnement par de nouvelles déclarations et définitions.

Nous présentons d'abord les déclarations et définitions globales, qui agissent sur l'environnement. Nous décrivons ensuite le mécanisme de sections, qui agit sur le contexte. Ceci permettra d'obtenir des développements paramétriques, réutilisables dans des conditions variées.

3.3.1 Déclarations et définitions globales

Déclarations

Une déclaration globale est une commande de la forme “ `Parameter` v`:`A ” (avec plusieurs variantes). L'effet de cette déclaration est simplement l'ajout à l'environnement courant de la déclaration $(v : A)$.

Par exemple, la commande suivante déclare une constante de type `Z` :

```
Parameter max_int : Z.
```
max_int is assumed

Remarquons qu'aucune valeur n'est associée à l'identificateur `max_int`. Contrairement à certains langages de programmation tels que `C`, il ne sera pas possible de revenir sur cette indétermination. La constante `max_int` reste donc arbitraire pour tout le reste du développement.

Définitions

La syntaxe d'une définition de constante est “ `Definition` c`:`A `:=` t ” ; si l'on laisse au système le soin d'inférer le type A à partir de t, on peut écrire simplement “ `Definition` c `:=` t ”.

Pour que cette définition soit acceptée, il faut que l'expression t soit typable dans l'environnement courant, soit bien de type A si ce type est spécifié dans la définition, et ne pose pas de problème lié à l'usage d'un identificateur déjà déclaré (voir le manuel pour tous ces détails). Dans ce cas, l'effet d'une telle définition est de rajouter à l'environnement courant la définition $(c := t : A)$.

Dans l'exemple suivant, nous définissons une constante `min_int` de type `Z`. Remarquons que cette définition utilise le paramètre `max_int` défini plus haut. La commande `Print` permet d'imprimer la définition d'un identificateur, ainsi que son type.

```
Open Scope Z_scope.

Definition min_int := 1-max_int.
Print min_int.
```
min_int = 1-max_int : Z

Définition d'une fonction

L'exemple suivant montre que plusieurs syntaxes peuvent être utilisées dans une définition de fonction. La première écriture exprime simplement la définition de l'identificateur `cube` par une expression, laquelle se trouve être une abstraction. Les deuxième et troisième écritures insistent sur le fait que `cube` est une fonction d'argument `z`.

```
Definition cube := fun z:Z ⇒ z*z*z.

Definition cube (z:Z) : Z := z*z*z.
```

```
Definition cube z := z*z*z.
```

Ces trois écritures sont totalement équivalentes, et la réponse de `Print` est la même dans les trois cas :

```
Print cube.
```
*cube = fun z:Z ⇒ z*z*z : Z→Z*
Argument scope is [Z_scope]

L'information imprimée indique également que l'argument donné à cette fonction sera systématiquement interprété dans la portée `Z_scope`.

Pour finir cette série d'exemples, nous définissons d'abord une fonctionnelle, que nous utilisons dans une seconde définition (nous n'imprimons pas toutes les réponses du système *Coq*).

```
Definition Z_thrice (f:Z→Z)(z:Z) := f (f (f z)).

Definition plus9 := Z_thrice (Z_thrice (fun z:Z ⇒ z+1)).
```

Exercice 3.5 Écrire une fonction qui prend cinq arguments entiers et retourne la somme de ces nombres.

3.3.2 Sections et variables locales

Les *sections* définissent un mécanisme de blocs similaire à celui de nombreux langages de programmation (*C*, *Java*, *Pascal*, etc.) permettant la déclaration et la définition de variables locales et contrôlant leur portée.

En *Coq*, les sections sont nommées, et les commandes de début et de fin de section sont respectivement “ `Section` *`id`* ” et “ `End` *`id`* ”, où *id* est le nom choisi pour la section. Naturellement, les sections peuvent être imbriquées, et les ouvertures/fermetures de sections doivent respecter une discipline de systèmes de parenthèses.

Afin de présenter ce formalisme et ses avantages, nous présentons un petit développement structuré en sections. Nous en montrons d'abord le texte, que nous reprendrons en le commentant pas à pas. Il s'agit de la définition paramétrique de polynômes du premier et du second degré.

```
Section binomial_def.
 Variables a b:Z.
 Definition binomial z:Z := a*z + b.
 Section trinomial_def.
  Variable c : Z.
  Definition trinomial z:Z := (binomial z)*z + c.
 End trinomial_def.
End binomial_def.
```

Ce développement est structuré en deux sections imbriquées, de noms respectifs `binomial_def` et `trinomial_def`. La section la plus externe `binomial_def` se situe au niveau global : en dehors de toute autre section.

Nous utilisons dans ce développement la possibilité d'ouvrir *localement* une portée. L'utilisation de `Z_scope` est donc limitée au texte de toute la section `binomial_def`.

Cette section déclare deux *variables locales* `a`, et `b` de type `Z`. On remarque que le mot clef `Variable` et sa variante `Variables` permettent de signaler une déclaration locale, par opposition à `Parameter`. La portée des déclarations de `a` et `b` est limitée au reste de la section `binomial_def`. Dans cette portée, le contexte courant sera donc le contexte vide augmenté des déclarations de `a` et `b`, c'est à dire la suite Γ_1 = `[a:Z; b:Z]`. Il en est de même pour la déclaration de `c` : à partir de cette déclaration, et jusqu'à la fin de `trinomial_def`, le contexte courant est Γ_2 = `[a:Z; b:Z ; c:Z]`.

Il est intéressant de noter que les définitions globales de `binomial` et `trinomial` se font dans un contexte non vide. Ceci permet en premier lieu d'accepter ces définitions, car le typage du terme associé à `binomial` se fait dans le contexte Γ_1, déclarant `a` et `b` de type `Z`, de même pour `trinomial` et Γ_2. Pour cette raison, *Coq* attache à toute constante le contexte dans lequel elle a été définie.

Si une constante utilise dans sa définition des variables locales, sa valeur, ainsi que le type associé, peut varier au fur et à mesure des fermetures successives de sections. En effet, si lors d'une fermeture de section une variable v « disparaît » du contexte courant, et que v est utilisée dans la définition de c, la déclaration de v est remplacée par une abstraction sur v.

Afin de bien saisir ces évolutions, reprenons le texte *Coq* précédent, en y ajoutant plusieurs commandes `Print` :

```
Reset binomial_def.

Section binomial_def.
 Variables a b:Z.
 Definition binomial (z:Z):= a*z + b.
 Print binomial.
binomial = fun z:Z ⇒ a*z + b
     : Z→Z
Argument scope is [Z_scope]
 Section trinomial_def.
  Variable c : Z.
  Definition trinomial (z:Z) := (binomial z)*z + c.
  Print trinomial.
trinomial = fun z:Z ⇒ binomial z * z + c
     : Z→Z
Argument scope is [Z_scope]
 End trinomial_def.
 Print trinomial.
```

```
 trinomial = fun c z:Z ⇒ binomial z * z + c
     : Z→Z→Z
Argument scopes are [Z_scope Z_scope]
End binomial_def.
Print binomial.
binomial = fun a b z:Z ⇒ a*z + b
     : Z→Z→Z →Z
Argument scopes are [Z_scope Z_scope Z_scope]
Print trinomial.
trinomial =
fun a b c z:Z ⇒ binomial a b z * z + c
     : Z→Z→Z→Z→Z
Argument scopes are [Z_scope Z_scope Z_scope Z_scope]
```

Cette association de définitions globales et de déclarations locales nous permet de définir des fonctions paramétriques. Les trois exemples ci-dessous montrent comment utiliser les valeurs associées aux constantes `binomial` et `trinomial`, une fois celles-ci « exportées » dans l'environnement global.

```
Definition p1 : Z→Z := binomial 5 (-3).
Definition p2 : Z→Z := trinomial 1 0 (-1).
Definition p3 := trinomial 1 (-2) 1.
```

Remarques

Si une variable locale n'est pas utilisée dans une définition globale à l'intérieur d'une section, alors elle n'intervient pas dans la construction de l'abstraction décrite ci-dessus. Dans l'exemple suivant, seules les variables `m` et `a` sont utilisées pour définir `f`. À la fin de la section `mab`, `f` sera une abstraction sur les seules variables `m` et `a`, (contrairement à `g`, dont la définition utilise toutes les variables locales).

```
Section mab.
 Variables m a b:Z.
 Definition f := m*a*m.
 Definition g := m*(a+b).
End mab.
Print f.
 f = fun m a:Z ⇒ m*a*m : Z→Z→Z
Argument scopes are [Z_scope Z_scope]

 Print g.
 g = fun m a b:Z ⇒ m*(a+b) : Z→Z→Z→Z
Argument scopes are [Z_scope Z_scope Z_scope]
```

Exercice 3.6 Utiliser le mécanisme de sections pour construire sans écrire explicitement d'abstractions une fonction qui prend 5 arguments et retourne leur somme.

3.3.2.1 Définitions locales

Il est possible de définir des variables dont la portée est limitée à l'intérieur d'une section. Ces définitions sont surtout utilisées pour aérer un texte ou éviter la duplication d'expressions ou de calculs. Une définition locale se fait grâce à la commande " `Let` v`:`A`:=` t ". Elle a pour effet d'augmenter le contexte courant de la définition $(v : t := A)$. Si l'on préfère laisser le système `Coq` inférer le type de t, on peut omettre l'indication "`:`A ".

Voici un exemple de section contenant des définitions locales ; nous remarquons qu'à la fermeture de section, les variables locales utilisées dans la définition de `h` donnent lieu à création de liaisons locales.

```
Section h_def.
 Variables a b:Z.
 Let s:Z := a+b.
 Let d:Z := a-b.
 Definition h : Z := s*s + d*d.
End h_def.
Print h.
 h =
fun a b:Z ⇒
  let s := a+b in let d := a-b in s*s + d*d
 : Z→Z→Z
Argument scopes are [Z_scope Z_scope]
```

3.4 Un peu de calcul

Rappelons que *Coq* n'est pas un environnement de programmation au sens habituel, son utilisation principale étant d'aider à *construire* des programmes corrects, plutôt que les *exécuter* plus ou moins efficacement. Si exécution il y a, elle se fait sur le programme *extrait* du développement, puis compilé.

Néanmoins, lors de développements plus ou moins ardus de programmes certifiés, on peut avoir besoin d'effectuer des calculs. Une première raison est le besoin d'expérimenter, de tester nos constructions. Une autre se fera évidente au cours de notre présentation : certains calculs seront *nécessaires* aux développements. Nous insisterons souvent sur cette utilisation de calculs à la fois par l'utilisateur et le système.

Ces calculs se présentent sous la forme de séries de *réductions* (ou *conversions*), c'est-à-dire de transformations élémentaires de termes. Diverses *tactiques de calcul* permettent de préciser quelles réductions appliquer et dans quel ordre.

La commande `Eval` nous permet de *normaliser* des termes, c'est à dire appliquer une suite de réductions jusqu'à l'obtention d'une forme irréductible (ou forme *normale*). De nombreuses options sont disponibles pour cette commande, et nous renvoyons au manuel de référence de *Coq* pour une description détaillée.

Avant de définir les 4 types de conversion utilisés dans *Coq*, nous devons présenter l'opération élémentaire de *substitution*.

3.4.1 Substitution

Soient deux termes t et u et v une variable ; nous notons $t\{v/u\}$ le remplacement des occurrences libres de v par u dans t. Cette opération ne doit pas introduire de nouvelles liaisons (captures de variable), aussi s'accompagne-t-elle de toute α-conversion nécessaire. On dit que $t\{v/u\}$ est une *instance* de t.

Par exemple, considérons les termes t = `A→A` et u = `nat→nat` ; le terme t{`A`/u} est alors : `(nat→nat)→nat→nat`.

De même, prenons t = “ `fun z:Z ⇒ z*(x+z)` ”, v = `x`, et u = “ `z+1` ” ; avant de remplacer les occurrences libres de `x` dans t par “ `z+1` ”, on renomme par α-conversion la variable liée `z` en une nouvelle variable, mettons `w`, et obtenons le terme “ `fun w:Z ⇒ w * (z+1+w)` ”. Si nous n'avions pas pris cette précaution, nous aurions obtenu le terme “ `fun z:Z ⇒ z*(z+1+z)` ”, où toutes les occurrences de `z` sont liées par la même abstraction.

3.4.2 Règles de conversion

Les conversions utilisées dans *Coq* sont de quatre sortes :

La δ-réduction (prononcez *delta*-réduction) permet de remplacer un identificateur par sa définition : soit t un terme, et v un identificateur défini par t' dans l'environnement ou le contexte courant ; alors la δ-conversion transforme le terme t en $t\{v/t'\}$.

Dans les exemples suivants, nous utilisons la δ-conversion sur les constantes `Zsqr` et `my_fun`. On notera que les arguments de la commande `Eval` précisent qu'on utilise une stratégie d'appel par valeur (`cbv`). Le mot-clef `delta` peut-être suivi d'une liste d'identificateurs, auquel cas les δ-réductions se limitent aux identificateurs présents dans cette liste.

```
Definition Zsqr (z:Z) : Z := z*z.


Definition my_fun (f:Z→Z)(z:Z) : Z := f (f z).

Eval cbv delta [my_fun Zsqr] in (my_fun Zsqr).
    = (fun (f:Z→Z)(z:Z) ⇒ f (f z))(fun z:Z ⇒ z*z)
    : Z→Z


Eval cbv delta [my_fun] in (my_fun Zsqr).
  = (fun (f:Z→Z)(z:Z) ⇒ f (f z)) Zsqr
    : Z→Z
```

La β-réduction (prononcez *beta*-réduction) permet de transformer un *β-radical*[5] c'est à dire un terme de la forme “ `(fun` v`:`τ `⇒` t`)` u ” en le terme $t\{v/u\}$.

5. *β-redex* en anglais

Si nous reprenons le terme obtenu par δ-réductions à partir de “ `my_fun Zsqr` ”, nous observons deux β-radicaux, l'un associé à l'abstraction sur `f`, l'autre à l'abstraction sur `z0`. En effectuant les β-conversions possibles, nous obtenons la suite d'expressions ci-dessous :

1. `(fun (f:Z→Z)(z:Z) ⇒ f (f z))(fun (z:Z) ⇒ z*z)`

2. ```
   fun z:Z ⇒
       (fun z1:Z ⇒ z1*z1)((fun z0:Z ⇒ z0*z0) z)
   ```

3. `fun z:Z ⇒ (fun z1:Z ⇒ z1*z1)(z*z)`

4. `fun z:Z ⇒ z*z*(z*z).`

Sous *Coq* nous pouvons obtenir directement ce résultat en considérant à la fois la β-conversion, ainsi que la δ-conversion sur les constantes `my_fun` et `Zsqr`, en utilisant la stratégie d'appel par valeur :

```
Eval cbv beta delta [my_fun Zsqr] in (my_fun Zsqr).
    = fun z:Z ⇒ z*z*(z*z) : Z→Z
```

La ζ-réduction (prononcez *zeta*-réduction) consiste en l'élimination des liaisons locales (« let-in ») ; plus précisément, elle remplace une expression de la forme “ `let` v`:=`u `in` t ” par t{v/u}.

Reprenons l'exemple présenté en 3.3.2.1, page 57 ; la constante `h`, définie à l'intérieur d'une section, contient deux liaisons locales (sur les variables `s` et `d`). L'expérimentation ci-dessous montre le résultat de l'évaluation avec ou sans ζ-conversion :

```
Eval cbv beta  delta [h] in (h 56 78).
 = let s := 56+78 in let d := 56-78 in s*s + d*d
   : Z
 Eval cbv beta zeta delta [h] in (h 56 78).
 = (56+78)*(56+78)+(56-78)*(56-78)
   :Z
```

La ι-réduction (prononcez *iota*-réduction) associée aux objets inductifs, est présentée dans une autre partie de ce livre (sections 7.1.4 et 7.3.3) ; contentons nous pour le moment de préciser que la ι-réduction est responsable de certaines simplifications liées aux schémas de programmes récursifs, par exemple les réductions de “ `plus O n` ” en `n`, de “ `mult O p` ” en `O` et de “ `mult (S n) p` ” en “ `plus p (mult n p )` ”.

Les calculs sur `Z` s'effectuent par des réductions similaires. Dans les exemples suivants, la ι-réduction nous permet de « finir » les calculs ; notons que le mot-clé `compute` est synonyme de “ `cbv iota beta zeta delta` ”.

```
Eval compute in (h 56 78).
    = 18440 : Z
```
```

```
Eval compute in (my_fun Zsqr 3).
    = 81 : Z
```

Exercice 3.7 Écrire la fonction correspondant au polynome $2\times x^2+3\times x+3$ sur les entiers, en utilisant la λ-abstraction et les fonctions `Zplus` et `Zmult` fournies dans *Coq*. Vérifier la valeur de cette fonction sur les entiers 2, 3, 4.

3.4.3 Notations

Dans la présentation de certaines règles de *Coq*, nous pourrons avoir besoin de caractériser l'obtention de termes par une suite de réductions à partir d'un terme donné.

La notation ci-dessous exprime la propriété « Dans le contexte Γ et l'environnement E, le terme t' s'obtient à partir de t par une suite de β-réductions » de la façon ci-dessous :

$$E, \Gamma \vdash t \rhd_{\beta} t'$$

Si l'on considère une combinaison quelconque de β, δ, ζ ou ι-conversions, on indique cette combinaison en indice du symbole de réduction. Par exemple, pour une combinaison de β, δ et ζ-conversions, nous aurons :

$$E, \Gamma \vdash t \rhd_{\beta\delta\zeta} t'$$

3.4.4 Propriétés abstraites de la réduction

Les combinaisons de β, δ, ζ et ι-réductions jouissent de propriétés extrêmement importantes dans tout le domaine des systèmes de récriture.

- Toute suite de réductions issue d'un terme est finie ; en d'autres termes, tout calcul sur des termes du Calcul des Constructions Inductives à base de réductions se termine ; c'est la propriété de *normalisation forte*.
- Si t se récrit d'une part en t_1, d'autre part en t_2 (en un nombre quelconque d'étapes), il existe alors t_3, tel que t_1 et t_2 se récrivent en t_3. C'est la propriété de *confluence*.
- Si t se réduit en t', alors t et t' ont le même type.

Une conséquence importante des deux premières propriétés est qu'à partir d'un terme t, toute suite de réductions converge vers une forme normale unique ne dépendant que de t et des types de réductions (β, δ, $\ldots$) considérés.

Convertibilité

Une propriété importante (et décidable dans le Calcul des Constructions) est la *convertibilité* : deux termes t et t' sont convertibles si ils se réduisent en un même troisième terme. Cette propriété se note de la façon suivante (nous considérons ici les quatre types de réductions, cependant toute combinaison de ces réductions est possible) :

$$E, \Gamma \vdash t =_{\beta\delta\zeta\iota} t'$$

Par exemple, les deux termes suivants sont convertibles, car réductibles en `3*3*3`.

– `let x := 3 in let y:= x*x in y*x`
– `(fun (z:Z) ⇒ z*z*z) 3`

La décidabilité de la convertibilité vient directement des propriétés abstraites de la réduction : pour décider si t et t' sont convertibles, il suffit de déterminer les formes normales de t et t' et de les comparer.

Les propriétés de normalisation forte et de confluence jouent un rôle central dans la conception de *Coq* et dans la confiance qu'on lui porte. Chaque fois que le Calcul des Constructions a été étendu : constructions inductives, co-inductives, modules, etc., les chercheurs développant ce système ont dû prouver que les extensions considérées ne cassaient pas ces propriétés.

3.5 Types, sortes et univers

Jusqu'à présent, nous nous sommes limités dans nos exemples à travailler avec un ensemble figé de types atomiques : `nat`, `Z` et `bool`, ainsi que leur clôture par $\rightarrow$. Il est important de pouvoir définir de nouveaux noms de types, et également de pouvoir définir des fonctions sur des types *arbitraires*, en ouvrant ainsi la voie au polymorphisme. Plutôt que de définir de nouveaux mécanismes à cet usage, les concepteurs de *Coq* ont préféré étendre ceux que nous avons déjà présentés. Il suffit de considérer que les expressions et les types que nous avons vus sont des cas particuliers de termes, et que les notions de typage, de déclarations, de définitions, etc. sont les mêmes pour toutes les sortes de termes (types ou expressions).

Il nous faut alors répondre à la question suivante :

Puisqu'un type est un terme, alors quel est son type ?

3.5.1 La sorte `Set`

Dans le Calcul des Constructions, on appelle *sorte* le type d'un type (considéré en tant que terme). Une sorte est toujours un identificateur. Si le type A d'un terme t est de sorte s, on dira de façon abrégée que t a pour sorte s.

Parmi les sortes prédéfinies en *Coq* se trouve la sorte `Set`, qui sert de type à toute spécification de programme. Plus précisément, nous pouvons donner une définition formelle de la notion de spécification.

Définition 3.2 (Spécifications) Nous appelerons *spécification* tout terme de *Gallina* dont le type est la sorte `Set`.

On remarquera que les types de toutes les expressions jusqu'ici considérées sont alors des spécifications. Par exemple, le type `nat→nat` est une spécification, celle des fonctions totales de `nat` dans `nat`. Des spécifications plus riches, comme « nombre premier supérieur à 567347 », « tri par ordre lexicographique », seront abordées dans le chapitre 10.

Une fois les spécifications définies, nous pouvons définir les programmes ou expressions.

Définition 3.3 (Programmes, expressions) Nous appelerons *programme* ou *expression* tout terme t de *Gallina* dont le type est une spécification A. On dit également que t est une *réalisation* de A.

Exemples Nous pouvons vérifier à l'aide de `Check` que toutes les spécifications données jusqu'ici en exemple sont bien des termes de type Set. Le lecteur peut multiplier les essais de la forme suivante :

```
Check Z.
Z : Set
Check ((Z→Z)→nat→nat).
(Z→Z)→nat→nat : Set
```

Remarque La formation de types de la forme $A{\rightarrow}B$ s'exprime désormais sous la forme d'une simple règle de typage, appelée à subir de nombreuses généralisations :

$$\textbf{Prod-Set}\quad \frac{E,\Gamma\vdash A:\texttt{Set}\quad E,\Gamma\vdash B:\texttt{Set}}{E,\Gamma\vdash A{\rightarrow}B:\texttt{Set}}$$

Pour justifier par exemple le jugement $E_0,[]\vdash \texttt{(Z}{\rightarrow}\texttt{Z)}{\rightarrow}\texttt{nat}{\rightarrow}\texttt{nat}:\texttt{Set}$, il suffit de considérer que les déclarations (`nat : Set`) et (`Z : Set`) font partie de l'environnement E_0, et d'appliquer 3 fois la règle ci-dessus.

3.5.2 Les univers

La sorte Set est bien un terme du Calcul des Constructions, et doit à son tour posséder un type. Mais ce type — encore un terme —, va avoir à nouveau un type, etc.

Le Calcul des Constructions considère une hiérarchie infinie de sortes appelées *univers*. Cette famille est formée des sortes $\texttt{Type}(i)$, pour tout i dans $\mathbb{N}$, et est caractérisée par les relations suivantes :

$$\begin{array}{rcll}\texttt{Set} & : & \texttt{Type}(i) & (\text{pour tout } i)\\ \texttt{Type}(i) & : & \texttt{Type}(j) & (\text{si } i<j)\end{array}$$

L'ensemble des termes du Calcul des Constructions est alors stratifié en niveaux ; nous avons jusqu'ici rencontré les catégories suivantes :

niveau 0 **:** les *expressions*, comme `O`, `S`, `trinomial`, ...

niveau 1 **:** les *spécifications*, comme `nat`, `nat→nat`, `nat→nat→nat→nat`, `(Z→Z)→nat→nat`, ...

niveau 2 **:** la *sorte* `Set`,

niveau 3 **:** l'*univers* `Type`(0),

niveau 4 **:** l'*univers* `Type`(1),

...

niveau $i+3$ **:** l'*univers* `Type`(i),

...

Chaque terme d'un niveau i a pour type un terme situé dans le niveau $i+1$. On remarquera cependant que tout univers `Type(`i`)` a non seulement le type `Type(`$i+1$`)`, mais aussi tous les types `Type(`j`)`, pour $j > i$; dans le Calcul des Constructions, le type d'un terme n'est donc pas forcément unique.

De l'inaccessibilité des univers La hiérarchie d'univers présentée ci-dessus est inaccessible à l'usager commun. Seule l'abréviation `Type` (pour `Type(0)`) est disponible en entrée, et toute impression d'un univers `Type(`i`)` est abrégée en `Type`.

Par exemple, si nous voulons vérifier le type de `Set`, nous obtenons la réponse ci-dessous, abrégeant le jugement `Set` : `Type(0)` :

`Check Set.`
Set : Type

Le dialogue suivant pourrait faire croire qu'il existe un terme appelé `Type` ayant lui-même pour type :

`Check Type.`
Type : Type

La réponse de *Coq* ne doit pas être lue littéralement comme le jugement `Type(0)` : `Type(0)`, ce qui mènerait droit à l'incohérence du Calcul des Constructions (voir [27]). Cette réponse doit être considérée comme une abréviation du jugement `Type(0)` : `Type(1)`, ou plus généralement des jugements `Type(0)` : `Type(`i`)` pour tout $i > 0$.

Extension de la notion de convertibilité La hiérarchie des univers de *Coq* est prise en compte dans une extension de la notion de convertibilité en une relation d'ordre compatible avec cette hiérarchie. Cette relation est notée de la façon suivante :

$$E, \Gamma \vdash t \leq_{\delta\beta\zeta\iota} t'$$

Nous n'entrons pas dans les détails de cette extension, dont le lecteur peut trouver la définition précise dans le manuel de référence de *Coq* et dans [61]. Nous en donnons deux propriétés :

– Si t et t' sont convertibles (pour E et Γ), alors $E, \Gamma \vdash t \leq_{\delta\beta\zeta\iota} t'$,
– $E, \Gamma \vdash$ `Set` $\leq_{\delta\beta\zeta\iota}$ `Type(i)` pour tout i.

3.5.3 Définitions et déclarations de spécifications

Les notions présentées ci-dessus nous donnent immédiatement la possibilité de définir ou déclarer de nouvelles spécifications. En effet, les commandes de définition, tant locales que globales, permettent de lier un identificateur à un terme de type donné. Si ce type est `Set`, alors le terme considéré est une spécification. Ce mécanisme nous permet donc de donner un nom à une spécification.

Par exemple, supposons que nous voulions appeler `Z_bin` le type des fonctions binaires sur `Z`. Il nous suffit alors de définir la constante `Z_bin` par le terme `Z→Z→Z`, de type `Set`.

```
Definition Z_bin : Set := Z→Z→Z.
```

La règle de conversion

À la différence de nombreux langages de programmation, l'unicité du type d'un terme est loin d'être garantie en *Coq*. Nous avons déja vu que l'univers `Type(`i`)` avait une infinité de types, et nous verrons dans le chapitre sur le produit dépendant (chapitre 5) une autre raison de cette non-unicité.

Afin d'illustrer ce point, considérons la définition suivante de la fonction $(z_0 - z_1)^2$ (sur les entiers relatifs) :

```
Check (fun z0 z1:Z ⇒ let d := z0 - z1 in d * d).
```
*fun z0 z1:Z ⇒ let d := z0 - z1 in d * d : Z→Z→Z*

```
Definition Zdist2 : Z_bin :=
  fun z0 z1:Z ⇒ let d := z0 - z1 in d * d.
```

La double abstraction définissant `Zdist2` reçoit — par la stricte application des règles de typage — le type `Z→Z→Z` , et non `Z_bin` comme spécifié.

Nous devons donc considérer comme équivalentes les deux spécifications `Z_bin` et `Z→Z→Z` (modulo le remplacement d'une constante de type par sa définition), et ajouter à notre batterie de règles de typage la *règle de conversion* suivante.

$$\textbf{Conv}\quad \frac{E,\Gamma \vdash t : A \quad E,\Gamma \vdash A \leq_{\delta\beta\zeta\iota} B}{E,\Gamma \vdash t : B}$$

Cette régle nous permet alors d'obtenir le jugement $E, [] \vdash$ `Zdist2 : Z_bin`.

Nous pouvons également vérifier en utilisant l'opérateur de coercition '`:`' que la spécification `nat→nat`, de sorte `Set`, a aussi pour sorte `Type`, ainsi que tous les univers `Type(`i`)`.

```
Check (nat→nat).
```
nat→nat : Set

```
Check (nat→nat:Type).
```
nat→nat:Type : Type

Déclarations de spécifications (un premier pas vers le polymorphisme)

Une déclaration de la forme `Variable` A`:Set` (ou sa variante globale avec `Parameter`) déclare une spécification *arbitraire*. À titre d'exemple, le script ci-dessous déclare localement une spécification `D`, trois variables dont le type s'exprime en fonction de `D`, puis la définition d'une « opération dérivée » exprimée à l'aide de ces variables. Nous pouvons constater que la programmation de `diff` en fonction de `D`, `op`, et `sym` a un certain parfum de polymorphisme. Le chapitre 5 nous permettra de définir rigoureusement cette notion. Signalons en passant la présence de commentaires dans le script ci-dessous ; comme en *pascal* et en *OCAML* ceux-ci sont délimités par le parenthésage `(* *)`.

```
Section domain.
  Variables (D:Set)(op:D→D→D)(sym:D→D)(e:D).
  Let diff : D→D→D :=
    fun (x y:D) ⇒ op x (sym y).
  (* ... *)
End domain.
```

3.6 De la spécification à la réalisation

Le problème de la réalisation d'une spécification donnée se pose en les termes suivants : on considère une spécification A, c'est à dire un terme de type `Set` dans un environnement E et un contexte Γ donnés. On cherche alors un terme de type A dans $E \cup \Gamma$; ce terme éventuel sera appelé *réalisation* de la spécification A.

Par exemple, considérons le début de section suivant :

```
Section realization.
 Variables (A B : Set).
 Let spec : Set := (((A→B)→B)→B)→A→B.
```

Le problème est alors de construire dans le contexte courant un terme de type `spec`. Avec un peu d'entraînement, on peut trouver directement une réalisation de `spec` :

```
 Let realization : spec
       := fun (f:((A→B)→B)→B) a ⇒ f (fun g ⇒ g a).
```
realization is defined

Fort heureusement, la tâche consistant à trouver une réalisation pour une spécification donnée peut être grandement facilitée par l'utilisation d'outils appelés *tactiques*, fournis en grand nombre dans le système *Coq*. Les tactiques permettent de construire peu à peu cette réalisation au cours d'un dialogue avec l'utilisateur.

Nous ne présentons cependant pas les tactiques dans ce chapitre, faute de disposer d'exemples convaincants. En effet, les spécifications que nous avons étudiées jusqu'à présent sont extrêmement pauvres et ont trop de réalisations différentes. Le risque d'utiliser un outil automatique pour réaliser une spécification un peu floue est que l'absence de contrôle peut résulter en des programmes tout à fait inintéressants.

Pour nous en convaincre, prenons la spécification suivante, associée à la transformation de fonctions sur `nat` en fonctions sur `Z` :

```
Definition nat_fun_to_Z_fun : Set := (nat→nat)→Z→Z.
```

Voici une liste de quelques réalisations, toutes correctes [6], dont seule la première paraît intuitivement intéressante.

```
Definition absolute_fun : nat_fun_to_Z_fun :=
  fun f z ⇒ Z_of_nat (f (Zabs_nat z)).

Definition always_0 : nat_fun_to_Z_fun :=
  fun _ _ ⇒ 0%Z.

Definition to_marignan : nat_fun_to_Z_fun :=
  fun _ _ ⇒ 1515%Z.

Definition ignore_f : nat_fun_to_Z_fun :=
  fun _ z ⇒ z.

Definition from_marignan : nat_fun_to_Z_fun :=
  fun f _ ⇒ Z_of_nat (f 1515%nat).
```

Le problème vient bien de ce que la spécification `Z_fun_to_nat_fun` est encore trop imprécise. Laisser à des outils plus ou moins automatiques le soin de la réalisation d'une spécification nous ferait abandonner tout contrôle sur le résultat obtenu.

Dans le cadre de la programmation, nous apprendrons à construire des spécifications plus précises (chapitre 10) : en reprenant l'exemple précédent, on pourra préciser que l'on cherche à construire une fonction ϕ de type `Z_fun_to_nat_fun`, telle que l'on ait la propriété suivante, pour tous f et z :

$$(\phi(f))(|z|) = |f(z)|$$

Dans ce cas, la spécification devient suffisamment précise pour éliminer toutes les solutions sauf la première.

À l'inverse, quand nous nous intéresserons à prouver des propositions, nous adopterons un point de vue totalement différent. Nous verrons que prouver un théorème d'énoncé A consiste à construire un terme t de type A. Mais un autre terme t' de même type remplit fort bien ce rôle de preuve de A. Cette indifférence au choix précis d'un terme de type donné est le *principe de non pertinence des preuves*, en anglais *proof irrelevance*.

6. On remarquera que, dans les 4 dernières définitions, le type des variables anonymes '`_`' est inféré à partir de la contrainte de type " `:nat_fun_to_Z_fun` ".

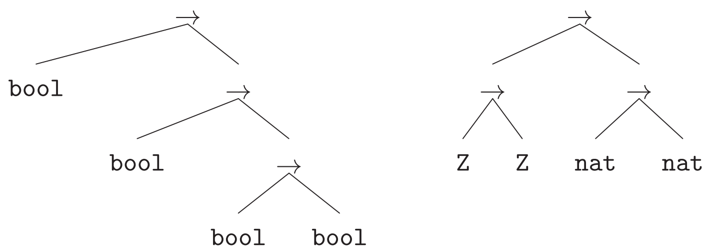


FIGURE 3.1 – Deux types simples

Chapitre 4

Propositions et preuves

Deux approches de la logique

Dans ce chapitre, nous abordons les techniques de raisonnement en *Coq*, en commençant par un « fragment » très réduit de la logique : la logique minimale propositionnelle, dont les formules sont exclusivement formées à partir de variables (dites *propositionnelles*) reliées par le connecteur d'implication. Par exemple, considérons trois variables propositionnelles, P, `Q` et `R` ; comment prouver la formule A ci-dessous ?

$$(\mathtt{P} \Rightarrow \mathtt{Q}) \Rightarrow (\mathtt{Q} \Rightarrow \mathtt{R}) \Rightarrow \mathtt{P} \Rightarrow \mathtt{R}$$

Deux démarches peuvent être suivies pour tenter de répondre à une telle question.

La première démarche, celle de Tarski [82], consiste à associer à toute proposition une *dénotation* : $\boldsymbol{v}$ (vrai), ou $\boldsymbol{f}$ (faux). La méthode des tables de vérité procède de cette approche : on considère toutes les affectations possibles des variables propositionnelles (à $\boldsymbol{v}$ ou $\boldsymbol{f}$), et on calcule dans chaque cas la valuation de la formule entière. Si cette valuation est $\boldsymbol{v}$ dans tous les cas, alors la formule est *valide*. La table de la figure 4.1 page 96 permet de s'assurer de la validité de A.

La démarche de Heyting [51], au contraire, remplace la question « La proposition P est-elle vraie ? » par la suivante : « Quelles sont les (éventuelles) preuves de P ? ». Le système *Coq* suit cette dernière démarche, qui rend alors nécessaire la définition d'une représentation informatique pour les énoncés et leurs preuves. Nous gagnons alors la possibilité d'explorer les démonstrations, pour les comprendre, en admirer l'élégance, ou en inférer des méthodes de travail utiles par la suite. Selon Heyting, une preuve d'une implication $P \Rightarrow Q$ est l'expression même de la relation de cause à effet liant P et Q, autrement dit comment une preuve de Q peut se déduire d'une preuve de P. En d'autres termes, une preuve de $P \Rightarrow Q$ est une fonction qui, étant donnée une preuve *arbitraire* de P, *construit* une preuve de Q.

Les deux approches précédentes s'appliquent en fait à deux conceptions différents de la logique. Celle de Tarski s'applique à la logique classique, qui vérifie le principe du tiers exclu « toute proposition est, soit vraie, soit fausse », ce qui justifie la construction de tables de vérité. L'approche de Heyting s'applique à la logique intuitionniste, qui n'admet pas ce principe. Nous répondons positivement page 95 à la question « La logique classique permet-elle de prouver plus de théorèmes que la logique intuitionniste ? ».

Si la logique intuitionniste paraît moins puissante que la logique classique, elle bénéficie néanmoins d'une propriété extrêmement importante pour un informaticien : la possibilité d'extraire des programmes corrects à partir de preuves (chapitre 11). C'est pour cette raison que nous suivons avec *Coq* l'approche intuitionniste de Heyting [1].

Logique et programmation fonctionnelle

Dans le cadre de la sémantique de Heyting, il est naturel de faire le lien avec les techniques de programmation fonctionnelle vues dans le chapitre précédent. Si nous considérons les preuves comme des expressions d'un langage fonctionnel, alors les énoncés sont des types (les types des preuves de ces énoncés). L'implication de Heyting $P \Rightarrow Q$ devient alors le type flèche $P{\rightarrow}Q$, et une preuve de cette implication sera une simple abstraction de la forme " `fun H:`P $\Rightarrow$ t ", où t est un terme de type Q dans un contexte étendu par la déclaration `(H:`P`)`.

Cette correspondance entre le modèle de la programmation fonctionnelle qu'est le λ-calcul et la déduction naturelle est appelée « l'isomorphisme de Curry-Howard ». Ce sujet a fait l'objet de nombreuses recherches, parmi lesquelles nous pouvons citer Scott [81], Martin-Löf [64], Girard, Lafont et Taylor [48] et les articles fondateurs de Curry et Feys[33] et Howard[54].

Cette correspondance nous permettra d'utiliser notre intuition de programmeur dans des tâches de démonstration, et, réciproquement, une intuition logique pour concevoir des programmes. De plus, les outils disponibles dans *Coq* permettent à la fois la programmation et le raisonnement.

Afin d'unifier les techniques de programmation et de raisonnement, nous renonçons aux notations spécifiques à la logique, en commençant par noter $P{\rightarrow}Q$ l'implication « P implique Q », au lieu de $P \Rightarrow Q$. La proposition A s'écrira alors de la façon suivante :

$(P{\rightarrow}Q){\rightarrow}(Q{\rightarrow}R){\rightarrow}P{\rightarrow}R$

Une preuve de cet énoncé est un λ-terme dont le type est cette proposition, par exemple le terme ci-dessous :

`fun (H:`$P{\rightarrow}Q$`)(H':`$Q{\rightarrow}R$`)(p:`P`)` $\Rightarrow$ `H' (H p)`

Ce terme exprime comment construire une preuve de `R` à partir de preuves arbitraires `H` de `P`→`Q`, `H'` de `Q`→`R` et `p` de P. Cette preuve consiste à appliquer

1. Remarquons cependant que *Coq* permet le raisonnement en logique classique, en chargeant la bibliothèque `Classical` ; le système de types de *Coq* est construit de telle façon que la faculté d'extraction de programmes n'est pas perdue.

`H` à `p` pour obtenir une preuve de `Q`, puis à appliquer `H'` à cette dernière preuve pour obtenir une preuve de `R`. Ce type de raisonnement est connu sous le nom de *syllogisme*.

Dans ce chapitre, nous étudierons comment faire évoluer le système de types du chapitre précédent pour faire coexister preuves et programmes. Nous présenterons aussi les premiers outils destinés à faciliter la preuve de théorèmes.

Parmi ces outils figurent les *tactiques*, destinées à faciliter la preuve d'une proposition, c'est à dire construire un terme de type donné. L'utilisation de ces outils permet d'obtenir des preuves de façon semi-automatique. À la différence de l'obtention d'un programme pour une certaine spécification, on peut supposer que le fait de prouver un énoncé est plus important que les détails de la preuve elle-même. Cette hypothèse — appelée *non-pertinence des preuves* — justifie la délégation de l'activité de démonstration à des outils semi-automatiques dont le but est de chercher une preuve, quelle qu'elle soit.

Enfin, signalons que nous pourrons dès à présent aborder l'aspect ludique de l'utilisation interactive de *Coq*. Le vocabulaire (*tactiques*, *buts*) est bien celui du jeu. L'usager voudrait bien que son théorème soit accepté avant de partir en week-end, en revanche le système joue le rôle d'un arbitre intransigeant qui veille à ce que toute règle soit respectée, mais suffisamment bienveillant pour proposer de l'aide à l'utilisateur un peu perdu.

4.1 La logique minimale propositionnelle

La logique s'attache aux moyens de prouver des *énoncés*, ou *propositions* ; il faut donc définir les propositions prises en compte, ainsi que les règles permettant d'établir leurs preuves.

Plus particulièrement, la *logique propositionnelle minimale* s'intéresse uniquement aux propriétés de l'*implication logique* : « `si` P `alors` Q » notée en *Coq* : " $P{\rightarrow}Q$ ". Les propositions que nous considérons sont des formules composées de propositions *atomiques*, reliées par le *connecteur* flèche $\rightarrow$. De ce fait, les énoncés de la logique propositionnelle minimale ont exactement la même structure que les spécifications du chapitre précédent. Les pages qui suivent traiteront simultanément deux thèmes : l'extension aux propositions et aux preuves du système de types du chapitre précédent, et la comparaison des deux activités de programmation et de démonstration.

4.1.1 Le monde des propositions et des preuves

La coexistence du monde des programmes et de celui des preuves est assurée en *Coq* par la création d'une nouvelle sorte située au même niveau que `Set` dans la hiérarchie des types : la sorte `Prop` des *propositions*. `Prop` se situe à coté de `Set` dans la hiérarchie de types du Calcul des Constructions, et a alors pour types tous les univers `Type(`i`)`.

De même que pour `Set`, nous avons la relation suivante :

$$E, \Gamma \vdash \texttt{Prop} \leq_{\delta\beta\zeta\iota} \texttt{Type(i)} \text{ pour tout } i$$

De la même façon que la sorte `Set` nous permettait une définition formelle de la notion de spécification, et par suite de la notion de programme, nous pouvons définir les notions de propositions et de preuves.

Définition 4.1 (Proposition, terme de preuve) On appelle *proposition* tout type P de sorte `Prop`. Si t est un terme de type P, on dit que t est un *terme de preuve* de P (ou plus brièvement *une preuve* de P).

Il arrive fréquemment qu'une proposition ne puisse se prouver qu'en supposant prouvées d'autres propositions. Ces propositions font alors l'objet de déclarations suivant le même mécanisme que les déclarations de variables des langages de programmation (voir 3.2.2). On identifie alors la notion d'hypothèse à celle de déclaration locale, et la notion d'axiome à celle de déclaration globale.

Définition 4.2 (Hypothèse) Une *hypothèse* est une déclaration locale $h : P$, où h est un identificateur (*nom* de l'hypothèse) et P une proposition (*énoncé* de l'hypothèse).

Il est conseillé, pour déclarer une hypothèse h d'énoncé P, d'utiliser la commande « `Hypothesis` $h{:}P$ », synonyme de « `Variable` $h{:}P$ ». De même, pour déclarer plusieurs hypothèses en une seule commande, on utilisera `Hypotheses` (même syntaxe que `Variables`).

Le rôle d'une hypothèse est de déclarer h comme terme de preuve arbitraire pour P. La portée locale de cette déclaration fait que h (et par conséquent l'hypothèse que P est démontrable) n'est utilisable que dans la section immédiatement englobante. L'utilisation d'hypothèses est au cœur de la méthode de raisonnement appelée *déduction naturelle* [79].

Définition 4.3 (Axiome) Un *axiome* est une déclaration globale $(x : P)$, où x est un identificateur et P une proposition.

De même que pour les hypothèses, on utilise dans un cadre logique la commande `Axiom` à la place de `Parameter`.

Un axiome sert à fournir à tout le reste du développement une preuve pour P (sous la forme du terme x), ce qui revient à supposer que P est vraie. Le danger d'utiliser des axiomes dans un développement *Coq* est la possibilité d'oublier que leur présence relativise tous les résultats obtenus lors de ce développement ; on pourrait en effet oublier le caractère arbitraire de la pose d'un axiome. Dans l'exercice 7.12, nous proposons un exemple où un axiome malencontreusement assumé provoque l'incohérence de toute la théorie ainsi construite.

Comment lire un jugement En reprenant les notations du chapitre 3, nous voyons qu'un environnement contient tous les axiomes d'une théorie, et un contexte les hypothèses courantes. La notation de jugement :

$$E, \Gamma \vdash \pi : P$$

peut alors se lire :

> « Compte-tenu des axiomes de E et des hypothèses de Γ, π est une preuve de P. »

Définition 4.4 (théorème, lemme.) Rappelons que l'environnement E et le contexte Γ sont constitués de déclarations et définitions, locales ou globales. Si P est une proposition et si $E \cup \Gamma$ contient une définition $(x := \pi : P)$, alors l'identificateur x peut remplacer le terme π (probablement plus complexe) chaque fois que l'on a besoin d'une preuve de P.

Les définitions globales d'identificateurs dont le type est une proposition sont appelés des *théorèmes* ou des *lemmes*

En résumé, dans un jugement $E, \Gamma \vdash \pi : P$, les ensembles E et Γ contiennent des faits supposés ou avérés pouvant intervenir dans la construction du terme de preuve π pour la proposition P.

4.1.2 Buts et tactiques

Les termes de preuve peuvent être très complexes, même pour des propositions simples, aussi avons-nous besoin d'outils d'aide à leur construction. Ces outils sont les *tactiques* de *Coq* :

Définition 4.5 (But) Un *but* est constitué de deux informations :
- une proposition P dont on cherche à construire une preuve,
- un contexte Γ, représentant une collection d'hypothèses utilisables pour construire la preuve de P

Soit E un environnement, Γ un contexte et P une proposition, le but associé à Γ et à P dans l'environnement E est noté " $E, \Gamma \stackrel{?}{\vdash} P$ ". Nous utiliserons parfois la notation simplifiée " $\Gamma \stackrel{?}{\vdash} P$ " lorsque l'environnement ne joue aucun rôle particulier.

Une *solution* de ce but est un terme t tel que l'on ait le jugement $E, \Gamma \vdash t : P$.

Définition 4.6 (Tactique) La résolution interactive d'un but se fait à l'aide de tactiques ; une tactique peut être définie comme une fonction qui à tout but b associe une suite (éventuellement vide) de nouveaux sous-buts $b_1, \ldots, b_n$. De plus, une tactique doit permettre, à partir des solutions des b_i, de construire une solution pour b. Notons qu'une tactique peut *échouer* sur un but, soit par impossibilité de le décomposer en sous-buts, soit dans la phase de reconstruction de la solution.

Cette notion de tactique trouve son origine dans le système *LCF* [50], et a été utilisée avec succès dans de nombreux assistants de preuve : *HOL* [49], *Nuprl* [26], *Coq* et *Isabelle* [75].

4.1.3 Un premier exemple

Une courte session nous permettra de rendre plus intuitives les notions associées aux preuves en *Coq*. Ces notions seront développées en détail par la suite. Considérons à nouveau la proposition présentée dans l'introduction de ce chapitre, c'est à dire `(P→Q)→(Q→R)→P→R`.

Déclaration des variables propositionnelles

Tout au long de ce chapitre, nous allons présenter des exemples utilisant au maximum 4 variables propositionnelles. Pour ce faire, nous ouvrons une section contenant les déclarations de 4 identificateurs de type `Prop` ; cette section sera refermée en fin de chapitre.

```
Section Minimal_propositional_logic.
 Variables P Q R T : Prop.
```
P is assumed
Q is assumed
R is assumed
T is assumed

Activation du système de buts

La commande suivante annonce que l'on souhaite prouver un théorème de nom `imp_trans`, en précisant son énoncé ; la commande `Proof` est facultative, et a pour but de signaler au lecteur (humain) le début de la preuve. Un but initial est alors créé, composé du contexte courant et de la formule à prouver. L'affichage du but courant se fait en séparant la présentation du contexte de celle de l'énoncé à prouver par une double barre horizontale.

```
Theorem imp_trans : (P→Q)→(Q→R)→P→R.
```
1 subgoal

```
  P : Prop
  Q : Prop
  R : Prop
  T : Prop
  ============================
   (P→Q)→(Q→R)→P→R
Proof.
```

La commande `Lemma` est synonyme de `Theorem`, mais s'utilise plutôt pour des résultats auxiliaires.

Introduction d'hypothèses

La tactique `intros` permet de réduire la tâche de construction d'une preuve de la proposition considérée à celle d'une preuve de `R`, dans un contexte augmenté de trois hypothèses : `H:P→Q`, `H':Q→R`, et `p:P`. Cette tactique est bien liée à l'implication de Heyting, et en particulier à la règle d'inférence d'« introduction de l'implication ». En argument d'`intros`, nous précisons le nom que nous souhaitons donner à chacune des 3 hypothèses.

```
intros H H' p.
```

1 subgoal

```
  P : Prop
  Q : Prop
  R : Prop
  T : Prop
  H : P→Q
  H' : Q→R
  p : P
  ============================
   R
```

Signalons que, par rapport à la situation précédente, nous avons à la fois simplifié l'énoncé à prouver et, par l'ajout d'hypothèses, augmenté les ressources disponibles pour continuer la preuve.

Application d'une hypothèse

L'examen du but courant tel qu'affiché par *Coq* nous montre que l'énoncé à prouver est la conclusion de l'hypothèse `H'`. La tactique “ `apply H'` ” a pour effet de donner comme nouveau but courant la prémisse de `H'`, c'est à dire la proposition `Q` ; afin d'alléger le texte.

```
apply H'.
```

1 subgoal

```
  P : Prop
  Q : Prop
  R : Prop
  T : Prop
  H : P→Q
  H' : Q→R
  p : P
  ============================
   Q
```

Le même raisonnement s'applique à la proposition `Q` et à l'hypothèse `H`.

```
apply H.
1 subgoal

  P : Prop
  Q : Prop
  R : Prop
  T : Prop
  H : P->Q
  H' : Q->R
  p : P
  ============================
   P
```

Utilisation directe d'une hypothèse

On observe que l'énoncé à prouver dans le but courant est exactement l'énoncé de l'hypothèse `p`. La tactique `assumption` reconnaît ce fait, et réussit sans engendrer aucun nouveau sous-but, avec pour solution le terme `p`. Il ne reste plus aucun but à résoudre, la preuve entière est alors terminée, ce qui est signalé par un message de *Coq* :

```
 assumption.
Proof completed.
```

Calcul et sauvegarde du terme de preuve

Une preuve se termine par la commande `Qed`. Cette commande a pour effet de calculer le terme de preuve associé à l'enchaînement de tactiques ci-dessus, de vérifier que son type est bien l'énoncé à prouver, et de sauvegarder le nouveau théorème sous la forme d'une définition reliant le nom du théorème, son énoncé (c'est à dire son type), et le terme de preuve construit. Notons que *Coq* affiche la suite de tactiques ayant conduit à la résolution du but initial. La preuve elle-même, comme n'importe quel terme de *Gallina*, peut être affichée à l'aide de la commande `Print`.

```
Qed.
intros H H' p.
apply H'.
apply H.
assumption.
imp_trans is defined

Print imp_trans.
imp_trans = fun (H:P->Q)(H':Q->R)(p:P) => H' (H p)
```

: $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$

Lecture d'un terme de preuve

Les termes de preuve en *Coq* pourraient paraître cryptiques à un lecteur non averti. Une traduction en « langue naturelle » peut faciliter leur lecture. La figure 4.2 page 96 propose une telle traduction pour le terme de preuve de `imp_trans`. Toute abstraction est traduite par l'introduction d'une hypothèse ; de même, les applications de fonctions se traduisent par des applications de lemmes.

Vers plus de concision

Nous avons volontairement écrit tous les détails de la preuve de `imp_trans`. Au fur et à mesure de l'utilisation de *Coq*, nous verrons comment obtenir des scripts de preuve beaucoup plus concis, soit en composant des tactiques, soit en faisant appel aux outils d'automatisation. Dans ce dernier cas, la preuve de `imp_trans` peut s'écrire simplement :

```
Theorem imp_trans : (P→Q)→(Q→R)→P→R.
Proof.
  auto.
Qed.
```

4.2 Règles de typage et tactiques

Nous reprenons de façon plus précise les notions abordées ci-dessus. Notre but est de montrer comment la logique propositionnelle se décrit en termes de λ-calcul simplement typé, à l'instar de la programmation fonctionnelle décrite dans le chapitre 3.

Nous verrons également les tactiques de base, montrerons quelques variantes, et soulignerons leur relation avec les règles de typage du calcul des constructions.

4.2.1 Règles de construction des propositions

Les règles de construction de propositions sont en fait des règles de typage de la même nature que celles utilisées pour construire des spécifications simples (section 3.5.1). La différence tient en l'utilisation de la sorte `Prop` à la place de `Set`.

Variables propositionnelles

Une variable propositionnelle est simplement un identificateur de type `Prop`. La règle **Var** (vue en 3.2.3) nous assure que si *id* est déclaré de type `Prop` dans

$E \cup \Gamma$ alors nous avons le jugement de typage :

$$E, \Gamma \vdash id : \texttt{Prop}$$

Nous pouvons vérifier ce fait en faisant appel à la commande `Check` :

```
Check Q.
```
Q : Prop

Construction d'implications

Nous avons vu que l'implication est représentée en *Coq* par la flèche $\rightarrow$; en d'autres termes, si P et Q sont des propositions, alors l'implication $P \Rightarrow Q$ est représentée par la proposition $P{\rightarrow}Q$.

Cette possibilité de construire des implications est représentée par la règle de typage ci-dessous :

$$\textbf{Prod-Prop}\quad \frac{E, \Gamma \vdash P : \texttt{Prop} \quad E, \Gamma \vdash Q : \texttt{Prop}}{E, \Gamma \vdash P{\rightarrow}Q : \texttt{Prop}}$$

Exercice 4.1 Reconstituer la suite de jugements permettant l'inférence suivante :

```
Check ((P→Q)→(Q→R)→P→R).
```
(P→Q)→(Q→R)→P→R : Prop

Une seule règle pour $\rightarrow$

Les règles **Prod-Set** et **Prod-Prop** ne diffèrent que par leur emploi de la sorte `Set` ou `Prop`. Nous pouvons alors les fusionner en une seule règle paramétrée par une sorte s.

$$\textbf{Prod}\quad \frac{E, \Gamma \vdash A : s \quad E, \Gamma \vdash B : s \quad s \in \{\texttt{Set}, \texttt{Prop}\}}{E, \Gamma \vdash A{\rightarrow}B : s}$$

Les deux règles vues précédemment s'obtiennent alors simplement en prenant $s = \texttt{Set}$ ou $s = \texttt{Prop}$.

4.2.2 Règles d'inférence et tactiques

Nous avons montré quelques rudiments de l'utilisation des tactiques `intros`, `apply` et `assumption` dans la construction interactive d'une démonstration. Nous reprenons en détail ces outils, en montrant leur relation avec les règles de typage du Calcul des Constructions, ainsi que leurs diverses variantes.

Tactiques exactes

Soit P une proposition ; si le contexte ou l'environnement courant contient une déclaration de la forme $x : P$, alors x est déjà un terme de preuve pour P. Ceci est une application directe de la règle **Var**, présentée page 45 dans le cadre de la programmation fonctionnelle :

$$\textbf{Var} \quad \frac{(x, P) \in E \cup \Gamma}{E, \Gamma \vdash x : P}$$

La tactique " `exact` x " permet dans ces conditions de résoudre immédiatement le but $E, \Gamma \overset{?}{\vdash} P$ en produisant la solution x, sans engendrer de nouveau sous-but.

Si $(x : P)$ est une hypothèse, c'est à dire faisant partie du contexte Γ, on peut résoudre le même but à l'aide de la tactique sans argument `assumption`, qui parcourt le contexte courant en cherchant une hypothèse d'énoncé P, et applique alors la règle **Var**. L'avantage de cette tactique est que l'on n'a pas besoin de connaître le nom de cette hypothèse. La tactique `assumption` échoue si le contexte ne contient pas une telle hypothèse.

L'exemple suivant montre une utilisation simple de la tactique `assumption`.

```
Section example_of_assumption.
 Hypothesis H : P→Q→R.

 Lemma L1 : P→Q→R.
 Proof.
  assumption.
 Qed.

End example_of_assumption.
```

Dans les cas où `assumption` ne peut être utilisée, par exemple si la déclaration $(x : P)$ est globale, on utilisera " `exact` x ".

Si l'on ne connaît pas le nom du théorème ou de l'axiome à utiliser, on peut se servir des outils de recherche de théorèmes de *Coq* (voir section 6.1.3). Une autre possibilité consiste à enregistrer les théorèmes ou axiomes importants dans les bases pour le raisonnement automatisé (voir section 8.2.1).

Pour finir, signalons que la tactique `exact` peut prendre comme argument un terme quelconque du type voulu, et n'est donc pas réduite à l'utilisation d'un axiome ou d'une hypothèse, mais peut utiliser n'importe quelle preuve de la proposition considérée. Un cas limite survient quand l'utilisateur connaît à l'avance le terme de preuve t permettant de terminer une démonstration. On utilise alors une variante de `Proof` prenant comme argument le terme t.

Voici par exemple une preuve utilisant cette possibilité :

```
Theorem delta : (P→P→Q)→P→Q.
Proof.
 exact (fun (H:P→P→Q)(p:P) ⇒ H p p).
```

```
Qed.
```

Le système Coq fournit une variante de la commande `Proof` pour ce type d'utilisation. Lorsque cette variante est utilisée on ne doit pas utiliser la command `Qed` pour terminer la démonstration. La preuve précédente peut être remplacée par la suivante :

```
Theorem delta : (P→P→Q)→P→Q.
Proof (fun (H:P→P→Q)(p:P) ⇒ H p p).
```

Modus ponens

La règle d'inférence du *modus ponens* (appelée parfois *élimination de l'implication*) permet de construire une preuve de Q à partir de preuves de $P \rightarrow Q$ et de P. En *Coq*, cette règle d'inférence est simplement la règle **App**, déjà rencontrée en 3.2.3, et utilisée cette fois-ci avec des propositions ; cette règle précise que le terme de preuve de Q est bien l'application (au sens des fonctions) d'une preuve t de $P \rightarrow Q$ à une preuve t' de P.

$$\textbf{App} \quad \frac{E, \Gamma \vdash t : P \rightarrow Q \quad E, \Gamma \vdash t' : P}{E, \Gamma \vdash t\ t' : Q}$$

Du point de vue tactique, si Q est la proposition à prouver, et que l'on dispose d'une preuve $t : P \rightarrow Q$, alors la tactique " `apply` t " engendre le but d'énoncé P. Si l'on réussit à résoudre ce dernier but avec une solution t', alors la tactique `apply` réussit avec le terme " $t\ t'$ ".

Cette tactique est utilisée à deux reprises dans notre exemple introductif page 75.

On peut remarquer que le langage courant s'accorde avec la correspondance entre preuves et programmes : n'emploie-t-on pas le verbe *appliquer* à la fois pour les théorèmes et les fonctions ? Dans les deux cas, il s'agit bien d'utiliser la règle **App**.

Précisions sur `apply`

Il est fréquent d'avoir à appliquer un terme t de type $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \rightarrow Q$ où n est un entier quelconque. Nous devons donner une description précise du comportement d'`apply` en fonction de t, de n et du but à résoudre.

Le terme t peut être utilisé pour construire un terme de type $P_k \rightarrow P_{k+1} \ldots P_n \rightarrow Q$ dès que l'on fournit des termes de type P_1, P_2, ..., P_{k-1}. Nous dirons alors que l'expression $P_k \rightarrow P_{k+1} \ldots P_n \rightarrow Q$ est le *type de tête de rang* k de t et Q est le *type final* de t si cette expression n'est pas elle-même une implication. Les types de tête d'une expression jouent un rôle dans le comportement de la tactique `apply`. Nous verrons plus tard que le type final joue également un rôle dans le comportement de certaines tactiques (voir par exemple la tactique `Elim` présentée en section 7.1.3).

Si le but courant a pour énoncé le type de tête de rang k de t, alors la tactique « `apply` t » engendre les sous-buts $\Gamma \overset{?}{\vdash} P_1$, $\Gamma \overset{?}{\vdash} P_2$, …, $\Gamma \overset{?}{\vdash} P_{k-1}$, où Γ est le contexte du but courant.

Si ces sous-buts ont pour solutions respectives t_1, t_2, …, t_k, alors le terme « $t\ t_1\ t_2\ \ldots\ t_k$ » est une solution du sous-but de départ. À titre d'exemple, considérons la preuve suivante :

```
Theorem apply_example : (Q→R→T)→(P→Q)→P→R→T.
Proof.
 intros H H0 p.
  ...
  H : Q→R→T
  H0 : P→Q
  p : P
  ============================
   R→T
```

Le but courant a pour énoncé la proposition `R→T`, et le terme `H` a pour type `Q→R→T`. La tactique « `apply H` » engendre alors un sous-but d'énoncé `Q`, lequel se résout par exemple par la tactique « `exact (H0 p)` ».

La solution produite pour le but `R→T` est alors le terme « `H (H0 p)` ». Voici la fin de la preuve :

```
 apply H.
 exact (H0 p).
Qed.
```

Dans l'exemple ci-dessous, la tactique « `apply H` » engendre deux sous-buts, associés respectivement aux propositions `P` et `Q`. On remarque qu'en présence de plusieurs sous-buts, seul le premier (actif par défaut) voit son contexte affiché.

```
Theorem imp_dist : (P→Q→R)→(P→Q)→(P→R).
 Proof.
 intros H H' p.
 1 subgoal
  ...
  H : P→Q→R
  H' : P→Q
  p : P
  ============================
   R

 apply H.

2 subgoals
  ...
  H : P→Q→R
```

```
 H' : P→Q
 p : P
 ============================
  P

subgoal 2 is :
Q
```

Le premier sous-but, d'énoncé `P`, se résout grâce à `assumption` ; la résolution du second se fait par un appel à " `apply H'` ", qui crée un sous-but d'énoncé `P`, à nouveau résolu par `assumption`.

```
 assumption.
 apply H'.
 assumption.
Qed.
```

L'impression du terme de preuve que nous venons de construire permet d'observer les applications engendrées par les deux utilisations d'`apply`.

```
Print imp_dist.
imp_dist =
fun (H:P→Q→R)(H':P→Q)(p:P) ⇒ H p (H' p)
     : (P→Q→R)→(P→Q)→P→R
```

La tactique `intro`

Puisqu'une preuve de $P{\rightarrow}Q$ est une fonction associant à toute preuve de P une preuve de Q, il est naturel d'utiliser la λ-abstraction pour prouver des implications, et donc la règle de typage **Lam** permettant de typer les abstractions ; cette règle a déjà été présentée page 49, mais nous la montrons à nouveau dans le présent cadre logique :

$$\textbf{Lam}\quad \frac{E,\Gamma :: (H:P) \vdash t \mathbin{\textbf{:}} Q}{E,\Gamma \vdash \texttt{fun}\ H{:}P\ \Rightarrow\ t \mathbin{\textbf{:}} P{\rightarrow}Q}$$

La tactique `intro` est associée à la règle de typage **Lam**. Soient un sous-but de la forme $\Gamma \overset{?}{\vdash} P{\rightarrow}Q$ et H un nom de variable non déclaré dans Γ ; la tactique `intro` H, engendre le sous but $\Gamma :: (H:P) \overset{?}{\vdash} Q$; si ce dernier sous-but a t pour solution, alors le sous-but de départ a pour solution l'abstraction " `fun (`H`:`P`)` $\Rightarrow$ t ".

Les variantes d'`intro`

La tactique `intro` admet un petit nombre de variantes, portant sur le nom ou le nombre des hypothèses à introduire.

Introduction de plusieurs variables : La tactique " `intros` v_1 v_2 ...v_n " s'applique à un but de la forme $E, \Gamma \stackrel{?}{\vdash} T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T$ et équivaut à la suite d'appels à " `intro` v_1 ", puis " `intro` v_2 ", ..., et " `intro` v_n ".

Dans l'exemple ci-dessous, la tactique " `intros p q` " peut remplacer les deux appels à `intro`.

```
Theorem K : P→Q→P.
Proof.
 intros p q.
 assumption.
Qed.
```

Non-spécification du nom de l'hypothèse : La tactique `intro` (sans paramètre) laisse au système le choix de la variable à introduire, en appliquant un algorithme de nommage et en respectant les contraintes de bonne formation des contextes.

Non-spécification du nom et du nombre d'hypothèses : La variante `intros` (sans paramètre) combine les actions des deux variantes précédentes ; on itère l'introduction de variables jusqu'à obtenir un but dont l'énoncé est atomique.

Dans l'exemple ci-dessus, le remplacement de " `intros p q` " par `intros` laisserait le système choisir les noms d'hypothèses `H`, puis `H0`. D'autres variantes existent, que l'on peut trouver dans le manuel de référence de *Coq*.

Remarque 4.1 L'utilisation de tactiques d'introduction sans paramètre, où l'on délègue au système le choix du nom des hypothèses, peut sembler commode (moins de choses à taper), mais pose parfois des problèmes de maintenance, discutés en section 4.7.

4.3 Structure d'une preuve interactive

Après avoir vu en détail quelques tactiques de base, nous pouvons décrire de façon précise comment ces tactiques s'enchaînent pour produire une preuve complète.

4.3.1 Activation du système de gestion de buts

L'activation du système de gestion de buts se fait en général par la commande " `Theorem` x `:` P " où P est l'énoncé du théorème et x est le nom qu'on souhaite lui donner. La variante “ `Lemma` x `:` P ” permet de souligner que x n'a pas une importance capitale, mais doit servir à prouver de plus importants résultats.

Le bon usage veut que `Theorem` et `Lemma` soient immédiatement suivies de la commande `Proof` (sans argument), ceci essentiellement pour faciliter la lecture des documents contenant des scripts de preuves.

Il est possible d'entamer une preuve en précisant seulement l'énoncé P à prouver, sans déclarer de nom associé au futur théorème. Ceci se fait grâce à la commande “ `Goal` P ”. Cette possibilité est maintenue par compatibilité avec les anciennes versions du système. Nous déconseillons son utilisation en dehors de petits essais « à jeter après usage ».

Une liste de sous-buts à résoudre est alors créée, ne contenant au départ que le but initial $\Gamma \overset{?}{\vdash} P$.

4.3.2 Étape courante d'une preuve interactive

Appelons $b_1, b_2, \ldots, b_n$ la liste courante de sous-buts à une étape quelconque de la preuve, chacun des sous-buts b_i étant de la forme $\Gamma_i \overset{?}{\vdash} P_i$.

Le système *Coq* affiche ces sous-buts, le premier de façon complète, les éventuels sous-buts suivants sans le contexte associé. Il est toujours possible de demander l'affichage complet du i-ème sous-but en appelant la commande “ `Show` i ”.

Une tactique est appliquée à l'un de ces sous-buts, b_i, qui est alors remplacé par la liste des nouveaux sous-buts obtenus en appliquant cette tactique. Ces nouveaux sous-buts sont insérés à la place de b_i. Cette opération s'appelle l'*expansion de* b_i. Par défaut, le sous-but développé est le premier de la liste, mais il est alors possible de s'attaquer à tout autre sous-but si l'usager le juge préférable. La commande pour activer la tactique *tac* sur le j-ème sous-but est “ j : `tac` ”. Si $j = 1$, le préfixe “ 1 : ” est généralement omis.

Si l'application d'une tactique échoue, alors la liste de sous-buts reste inchangée.

4.3.3 Hésitations

Il arrive que l'application d'une certaine suite de tactiques conduise à un but que l'on ne sait pas résoudre. Il alors possible de revenir en arrière afin de développer une meilleure approche. C'est l'emploi de la commande `Undo` et de sa variante “ `Undo` n ”, où n est le nombre d'étapes à « défaire ».

Il est possible de revenir au début de la preuve avec la commande `Restart`, voire d'abandonner totalement la tentative de preuve en exécutant `Abort` .

4.3.4 Fin normale d'un développement

La construction interactive d'un terme se termine normalement lorsque la suite courante de buts à résoudre est vide. Cette situation est signalée à l'usager par un message approprié : « *Proof completed.* ». Il faut alors procéder à la reconstruction de la solution du but initial, et sauvegarder la définition associée.

La commande permettant cette sauvegarde dépend de la façon dont la construction interactive a été démarrée :

– La preuve d'une proposition commencée avec “ `Theorem` *id*`:P` ” se termine par `Qed`; même règle pour `Lemma`,
– Si le développement a été initialisé par “ `Goal` A ”, il faut attribuer un identificateur au terme construit. Ceci se fait grâce à la commande “ `Save` *id* ”.

Exercice 4.2 Avec les tactiques de base que sont `assumption`, `intro[s]` et `apply`, prouver les lemmes suivants :

```
Lemma id_P : P→P.

Lemma id_PP : (P→P)→(P→P).

Lemma imp_trans : (P→Q)→(Q→R)→P→R.

Lemma imp_perm : (P→Q→R)→(Q→P→R).

Lemma ignore_Q : (P→R)→P→Q→R.

Lemma delta_imp  : (P→P→Q)→P→Q.

Lemma delta_impR : (P→Q)→(P→P→Q).

Lemma diamond : (P→Q)→(P→R)→(Q→R→T)→P→T.

Lemma weak_peirce : ((((P→Q)→P)→P)→Q)→Q.
```

4.4 La non pertinence des preuves

La symétrie `Prop/Set` que nous avons commencé à aborder pourrait nous induire à écrire des développements logiques exactement dans le même style que des développements de programmes. Nous devons cependant apporter une nuance : en programmation, pour une spécification A, deux programmes t et t' de type A ne sauraient être considérés comme totalement équivalents. Si par exemple A spécifie un programme de tri, et t et t' sont respectivement le tri par bulles et le tri rapide, des critères d'efficacité nous empêchent de les confondre. En revanche, si P est une proposition, on peut considérer que deux termes de preuves π et π' de P jouent exactement le même rôle : leur seule existence nous rassure sur la véracité de P. La confusion possible entre deux termes de preuves d'une même proposition s'appelle le principe de *l'indifférence aux preuves*, ou encore *non pertinence des preuves* (en anglais *Proof Irrelevance*).

Nous allons comparer les diverses techniques pour construire un terme de type donné, selon qu'elles sont plus ou moins bien adaptées à la construction de programmes ou à la démonstration de théorèmes. Nous serons guidés dans cette comparaison par le principe de non pertinence des preuves.

4.4.1 `Theorem` *versus* `Definition`

Soit P une proposition ; on pourrait penser à première vue que la suite de commandes :

```
Theorem name:P.
Proof  t.
```

est équivalente à la définition globale suivante :

```
Definition name : P := t.
```

Ce n'est pas le cas. La préférence pour l'usage de `Theorem` et ses collègues au détriment de `Definition` vient de deux raisons :

- La lisibilité d'un développement est accrue si l'on utilise à bon escient la terminologie logico-mathématique : théorèmes, lemmes, axiomes, hypothèses, *etc.*, au lieu du vocabulaire purement informatique : définitions, localité, déclarations, *etc.*,
- les définitions par `Theorem`, `Lemma`, *etc.*, présupposent l'acceptation du principe de non-pertinence des preuves, ce qui n'est pas le cas des définitions par `Definition` et `Let`. En effet, ce qui nous intéresse dans la preuve d'un théorème est son existence et non les détails de cette preuve. *Coq* définit un attribut associé aux définitions : la *transparence*, (ou son contraire l'*opacité*). Une définition $(x := t : T)$ est transparente si nous nous intéressons autant au terme t qu'au type T, et opaque si seuls le type T et l'existence de t — c'est à dire le fait que T est habité — nous importent. D'un point de vue technique, une définition transparente peut faire l'objet d'une δ-réduction, au contraire d'une définition opaque pour laquelle cela n'a aucun sens.
  Par défaut, une définition obtenue avec `Definition` est transparente, et opaque si elle est enregistrée avec `Theorem`, `Lemma`, etc. On consultera la documentation des commandes `Transparent` et `Opaque` de *Coq*.

4.4.2 Des tactiques pour construire des programmes ?

Nous avons abordé en 3.6 le problème de la construction assistée d'un terme de type A, avec A : `Set`. Les tactiques que nous avons présentées (ainsi que celles à venir), sont des outils interactifs traduisant les règles de typage. Or les règles de typage des termes sont les mêmes pour les types issus de `Set` aussi bien que `Prop`. Les tactiques, buts, et outils de développement interactifs peuvent alors s'utiliser pour réaliser des spécifications.

Mais considérons l'exemple suivant ; la fin du développement se signale par `Defined` à la place de `Qed`, ce qui annonce à *Coq* que la définition de `f` doit être considérée comme *transparente*.

```
Definition f : (nat→bool)→(nat→bool)→nat→bool.
 intros f1 f2.
 assumption.
Defined.
```

À l'aide de la commande Print, nous pouvons voir que la tactique `assumption` a privilégié la variable `f2` ; ce fait est corroboré par l'évaluation d'un terme contenant `f`.

```
Print f.
f = fun _ f2:nat→bool ⇒ f2
    : (nat→bool)→(nat→bool)→nat→bool

Argument scopes are [_ _ nat_scope]
Eval compute in (f (fun n ⇒ true)(fun n ⇒ false) 45).
    = false : bool
```

Nous constatons que la préférence accordée à `f2` au détriment de `f1` dépend de l'implémentation de la tactique `assumption`, mais ne peut être détectée à la seule vue du script de développement de `f`. Un développement de `f` par un simple appel à `auto` donnerait un résultat similaire.

Si toutefois nous déclarons `f` comme opaque, le problème lié à la δ-réduction disparaît, car `f` n'est plus réduite :

```
 Opaque f.

Eval compute in (f (fun n ⇒ true)(fun n ⇒ false) 45).
    = f (fun _:nat ⇒ true)(fun _:nat ⇒ false) 45 : bool
```

La morale de cette expérience ? Si l'on veut obtenir une définition transparente, c'est à dire si tous les détails d'une réalisation sont importants, il faut éviter l'usage de tactiques laissant un libre choix au système. Nous verrons plus loin dans cet ouvrage que la construction de programmes certifiés peut inclure la preuve de propositions ; dans ces preuves, le principe de non pertinence s'applique localement, et le recours aux outils automatiques peut alors être justifié (voir par exemple page 296).

4.5 Utilisation de sections

Toutes les preuves décrites dans ce chapitre sont bien formées dans le contexte d'une section `Minimal_proposition_logic` qui contient les déclarations de variables propositionnelles `P`, `Q`, `R`, `S`, and `T`. Le système de section nous a permis de modifier le contexte de la même manière que la tactique `intro`. Nous allons montrer comment nous pouvons utiliser ce mécanisme de sections pour construrie des preuves sans tactiques, simplement en utilisant des déclarations d'hypothèses à la place de la tactique `intro`, des applications de termes à la place de `apply` et des variables à la place de `exact` ou `assumption`.

De même que pour les programmes, nous pouvons utiliser le mécanisme de sections pour construire des termes de type $P{\rightarrow}Q$; il suffit d'ouvrir une section commençant par une hypothèse d'énoncé P et contenant un théorème x

d'énoncé Q. À la fermeture de section, le mécanisme de décharge transforme l'énoncé de x en $P{\rightarrow}Q$ (voir section 3.3.2).

L'avantage de cette méthode est que l'on peut facilement découper la preuve de x en lemmes, remarques, etc., pouvant utiliser l'hypothèse sur P.

Nous reproduisons ci-dessous une preuve du théorème `triple_impl`, d'énoncé (((P→Q)→Q)→Q)→P→Q. Le lecteur est invité à suivre le travail d'inférence de type effectué par le système, en reconnaissant les utilisations des règles de typage.

```
Section proof_of_triple_impl.
 Hypothesis H : ((P→Q)→Q)→ Q.
 Hypothesis p : P.

 Lemma Rem : (P→Q)→Q.
 Proof (fun H0:P→Q ⇒ H0 p).

 Theorem triple_impl : Q.
 Proof (H Rem).

End proof_of_triple_impl.


Print triple_impl.
 triple_impl =
fun (H:((P→Q)→Q)→Q)(p:P) ⇒ H (Rem p)
    : (((P→Q)→Q)→Q)→P→Q
```

Nous remarquons que l'énoncé de `Rem` et son terme de preuve rendent explicite l'hypothèse `(p:P)`, exploitée par la tactique `assumption`.

```
Print Rem.
Rem = fun (p:P)(H0:P→Q) ⇒ H0 p
  : P→(P→Q)→Q
```

4.6 Composition de tactiques

La construction interactive d'une preuve pourrait être très longue si elle devait se réduire à une suite d'activations de tactiques élémentaires. *Coq* propose une collection d'opérateurs permettant de composer les tactiques préexistantes pour en former de nouvelles. Par analogie avec la relation fonctions/fonctionnelles, ces opérateurs sont qualifiés de *tacticielles* ("*tacticals*" en Anglais). Nous verrons comment leur emploi permet d'alléger considérablement les preuves interactives.

Nous présentons quelques tacticielles simples, déjà utilisables dans notre contexte réduit de logique propositionnelle minimale.

Composition simple

La composition simple permet d'enchaîner l'application de deux tactiques sans s'arréter aux sous-buts intermédiaires. Plus précisément, soient *tac* et *tac'* deux tactiques ; alors la tactique composée `tac;tac'`, appliquée à un but g consiste à appliquer *tac* à g, puis *tac'* à chacun des sous-buts ainsi engendrés. En cas d'échec de *tac* ou *tac'*, la tactique composée échoue entièrement. Considérons par exemple le début de preuve ci-dessous :

```
Theorem then_example : P→Q→(P→Q→R)→R.
Proof.
 intros p q H.
1 subgoal
  ...
  p : P
  q : Q
  H : P→Q→R
  ============================
   R
```

On devine que la tactique “ `apply H` ” va engendrer deux sous-buts, d'énoncés respectifs `P` et `Q`. Chacun de ces nouveaux sous-buts se résout par `assumption`. La composition de tactiques “ `apply H; assumption` ” permet donc de résoudre ce but en une seule interaction.

```
 apply H; assumption.
Qed.
```

Il est possible d'enchaîner ainsi plusieurs tactiques, sous la forme $tac_1;tac_2;\ldots;tac_n$.

Il faut remarquer que l'utilisation de cette composition demande à l'utilisateur suffisamment d'intuition pour prévoir quels sous-buts seront engendrés à chaque étape de cette composition et quelle tactique sera appropriée pour résoudre *tous* ces nouveaux sous-buts. Cette intuition vient avec la pratique de *Coq*. Ceci est à rapprocher des jeux — tels les échecs par exemple —, où le joueur averti intègre dans ses tactiques les réponses prévisibles de l'adversaire. L'exemple suivant montre une composition de 5 tactiques :

```
Theorem triple_impl_one_shot : (((P→Q)→Q)→Q)→P→Q.
Proof.
 intros H p; apply H; intro H0; apply H0; assumption.
Qed.
```

Composition généralisée

La composition simple `tac;tac'` présuppose que la tactique *tac'* peut s'appliquer à *tous* les sous-buts crées par *tac*. Il peut cependant arriver que chacun de ces nouveaux sous-buts requière une tactique différente des autres.

L'opérateur de composition généralisée noté `tac;[tac`$_1$`|...|tac`$_n$`]` s'apparente à la composition simple, excepté le fait que la tactique *tac*$_i$ s'applique au *i*-ème sous-but engendré (en supposant que la tactique *tac* engendre exactement *n* sous-buts).

Dans l'exemple suivant, la tactique " `apply H` " engendre deux sous-buts d'énoncés respectifs `P` et `Q`; le premier sous-but est résolu immédiatement par `assumption`, le second par la composition simple " `apply H';assumption` " :

```
Theorem compose_example : (P→Q→R)→(P→Q)→(P→R).
Proof.
 intros H H' p.
1 subgoal
  ...
  H : P→Q→R
  H' : P→Q
  p : P
  ============================
   R

 apply H;[assumption | apply H'; assumption].
Qed.
```

La tacticielle '||'

Soient *tac* et *tac'* deux tactiques; appliquer la tactique " `tac || tac'` " à un but *b* revient à lui appliquer d'abord *tac*.

– Si cette application réussit, on ignore *tac'*
– Si cette application échoue, on applique *tac'* au but *b*.

Considérons par exemple la démonstration suivante :

```
Theorem orelse_example : (P→Q)→R→((P→Q)→R→(T→Q)→T)→T.
intros H r H0.
```

Un appel à " `apply H0` " engendrerait 3 sous-buts, d'énoncés respectifs `P→Q`, `R` et `T→Q`. La tactique `assumption` réussirait seulement sur les deux premiers, à l'inverse de " `intro H1` ". Par conséquent la composition suivante [2] :

```
  apply H0;(assumption || intro H1).
```

n'engendre plus qu'un seul sous-but, obtenu en ajoutant au contexte l'hypothèse `H1:T`.

2. L'opérateur '||' étant plus prioritaire que la composition de tactiques, les parenthèses que nous avons insérées par souci de clarté peuvent être supprimées.

La tactique `idtac`

La tactique `idtac` laisse un sous-but inchangé et réussit donc toujours ; son utilisation principale est de servir d'élément neutre pour certains opérateurs de composition de tactiques.

Dans la preuve ci-dessous, la tactique “ `apply H1` ” engendre trois sous-buts d'énoncés respectifs `P`, `Q` et `R` ; les deuxième et troisième sous-buts sont traités avec `apply`, le premier restant intact grâce à `idtac`. Les trois sous-buts résultants sont alors résolus par `assumption`.

```
Lemma L3 : (P→Q)→(P→R)→(P→Q→R→T)→P→T.
Proof.
 intros H H0 H1 p.
 apply H1;[idtac | apply H | apply H0]; assumption.
Qed.
```

Provoquer l'échec

Il peut sembler paradoxal d'inventer une tactique dont la seule action est d'échouer. La tactique `fail` de *Coq* a justement ce comportement. Comme `idtac`, son existence se justifie seulement comme opérande de certaines combinaisons de tactiques.

Considérons par exemple une composition “ *`tac`*`;fail` ”, appliquée à un but b ; si *tac* échoue sur b, alors toute la composition échoue ; si *tac* engendre un nombre non nul de sous-buts $b_1, \ldots, b_n$, alors la tactique composée échoue également. En revanche, si *tac* réussit immédiatement sur b, alors aucun sous-but n'est engendré, `fail` n'a aucun but à faire échouer, et la tactique composée réussit. La composition avec `fail` permet donc d'implémenter le schéma « ou je réussis tout de suite, ou bien j'échoue ».

Par exemple, la tactique composée “ `intro X;apply X;fail` ” permet de résoudre des buts de la forme $A \to A$, mais échoue sur $A \to B$ si l'application de `X` engendre au moins un sous-but :

```
Theorem then_fail_example : (P→Q)→(P→Q).
Proof.
 intro X; apply X; fail.
Qed.

Theorem then_fail_example2 : ((P→P)→(Q→Q)→R)→R.
Proof.
 intro X; apply X; fail.
```
Error : Tactic failure
```
Abort.
```

Des applications plus réalistes de la mise en échec sont présentées en section 8.2.1.

Capture d'échec

Certains opérateurs de composition de tactiques peuvent faire échouer une tactique composée dès que l'une des composantes échoue. Par exemple, dans le début de preuve suivant, la tactique " `apply H` " engendre trois sous-buts, dont seulement deux peuvent être résolus par `assumption`. Le script suivant provoque donc une erreur.

```
Theorem try_example : (P→Q→R→T)→(P→Q)→(P→R→T).
Proof.
 intros H H' p r.
 apply H; assumption.
```

Error : No such assumption

L'opérateur unaire `try` prend une tactique *tac* en paramètre et se comporte comme *tac*`||idtac`.

- Si *tac* échoue sur le sous-but considéré, alors " `try` *tac* " réussit en laissant ce sous-but inchangé (a le même effet que `idtac`),
- sinon, " `try` *tac* " a le même effet que *tac*.

Une composition de la forme " *tac* `;` `try` *tac'* " permet alors d'appliquer *tac'* aux sous-buts engendrés par *tac* pour lesquels cette application est possible. En utilisant ce schéma, nous terminons facilement notre preuve :

```
 apply H; try assumption.
```

1 subgoal

...
H : P→Q→R→T
H' : P→Q
p : P
r : R
============================
Q

```
 apply H'; assumption.
Qed.
```

En combinant `try` et `fail`, nous pouvons restreindre l'application d'une tactique *tac'* aux sous-buts engendrés par *tac* que *tac'* peut résoudre complètement, en laissant les autres sous-buts inchangés. La combinaison à utiliser est " *tac*`; try (`tac'`;fail)` ".

Exercice 4.3 Reprendre l'exercice 4.2, en utilisant le plus possible de tacticielles.

4.7 Quelques problèmes de maintenance

L'utilisation de tactiques laissant au système le soin de choisir le nom d'un identificateur peut poser un problème de maintenance des scripts de preuve ; la plupart du temps, ces scripts sont la trace d'une interaction entre le système *Coq* et l'utilisateur. Certains noms de variables peuvent donc être proposés par *Coq*, et cités par l'usager dans les commandes ultérieures. Le problème est que le script sauvegardé ne contient que la partie jouée par l'usager, et non les propositions du système. Par conséquent, certains noms apparaissant dans ce script peuvent rester inexpliqués.

Considérons à nouveau la preuve de `imp_dist`, et utilisons un appel à `intros` (sans paramètres).

```
Reset imp_dist.

Theorem imp_dist : (P→Q→R)→(P→Q)→(P→R).
 Proof.
  intros.
```

```
 ...
 H : P→Q→R
 H0 : P→Q
 H1 : P
 ============================
  R
```

Les noms d'hypothèses choisis par le système sont donc respectivement `H`, `H0` et `H1` ; la suite de la preuve utilise les deux premières variables :

```
 apply H.
 assumption.
 apply H0.
 assumption.
Qed.
```

On peut remarquer que les variables `H` et `H0` apparaissent dans le script en tant qu'arguments d'`apply`, mais ne sont pas « déclarées » préalablement (comme arguments d'`intros`).

Ce style peut non seulement affecter la lisibilité d'un script, mais aussi rendre difficile son transfert dans un autre contexte par couper/coller, car la politique de nommage des hypothèses est sensible à ce contexte.

À titre expérimental, nous allons copier la précédente preuve de `imp_dist` dans un contexte où `H` est déjà déclaré. Malheureusement, la tactique “ `apply H` ” ne permet pas de détecter de problème, et le diagnostic ne peut se faire qu'à partir de l'échec d'`assumption`.

```
Section proof_cut_and_paste.
 Hypothesis H : ((P→Q)→Q)→(P→Q)→R.
 Theorem  imp_dist_2 : (P→Q→R)→(P→Q)→(P→R) .
 Proof (* copy of imp_dist proof script *).
  intros.
1 subgoal
  ...
  H : ((P→Q)→Q)→(P→Q)→R
  H0 : P→Q→R
  H1 : P→Q
  H2 : P
  ============================
   R

  apply H.

2 subgoals
  ...
  ============================
   (P→Q)→Q

subgoal 2 is :
P → Q

  assumption.
 Error : No such assumption
 Abort.

End proof_cut_and_paste.
```

Les problèmes posés par la maintenance de preuves deviennent très complexes quand on considère des théories entières ; des techniques similaires à celles du génie logiciel sont alors nécessaires ; on pourra consulter à ce sujet les travaux d'Olivier Pons [78].

4.8 Problèmes de complétude

Nous considérons une question bien naturelle :

Quelles limites ont les outils que nous venons de présenter ?

Autrement dit, peut-on tout prouver avec un jeu de tactiques plus ou moins réduit ? Nous proposons deux réponses, suivant les propositions à démontrer.

4.8.1 Un jeu de tactiques suffisant

Nous avons jusqu'à présent travaillé avec peu de tactiques de base : `exact`, `assumption`, `apply`, et les diverses variantes d'`intro`.

Le catalogue des tactiques disponibles en *Coq* est destiné à s'accroître très rapidement, et même contenir des automatismes. On peut se demander si toutes ces nouveautés seront ou non nécessaires.

L'exercice suivant donne une réponse théorique dans le cadre restreint de la logique minimale étudiée dans ce chapitre :

Exercice 4.4 ** Montrer que s'il existe t tel qu'on puisse prouver le jugement $E, \Gamma \vdash t : P$ (avec les règles présentées dans ce chapitre,) alors le but d'énoncé P peut se résoudre dans l'environnement E et le contexte Γ par une suite d'applications de `assumption`, `apply` et `intro`. *On pourra faire la preuve par récurrence sur la structure de t ; on pourra de plus supposer que t est en forme normale.*

4.8.2 Des propositions impossibles à montrer

Il peut être intéressant de montrer quelques cas de buts n'ayant aucune solution. Considérons par exemple le but [`P:Prop`] $\overset{?}{\vdash}$ `P` (il est nécessaire pour cet exemple de considérer un contexte et un environnement ne permettant pas de prouver P par l'utilisation d'axiomes ou d'hypothèses qui rendraient l'expérience non-significative).

On peut prouver que ce but n'a aucune solution. Nous ne détaillons pas cette preuve, qui fait appel aux propriétés de normalisation (calcul de termes irréductibles) dans le λ-calcul typé [28].

Un autre exemple intéressant est la *formule de Peirce* ; considérons le but ci-dessous :

$$\texttt{P,Q:Prop} \overset{?}{\vdash} \texttt{((P}\rightarrow\texttt{Q)}\rightarrow\texttt{P)}\rightarrow\texttt{P}$$

Les mêmes techniques que précédemment permettent de montrer que ce but est insoluble ; cet exemple est d'autant plus intéressant que la formule de Peirce est valide en logique propositionnelle classique ; en témoigne la table de vérité de la figure 4.3 page 96. L'ensemble des propositions prouvables en logique minimale propositionnelle est donc un sous-ensemble strict de l'ensemble des formules valides en logique classique.

4.9 Autres tactiques

Nous avons vu que les trois tactiques `assumption`, `intro[s]` et `apply` suffisent pour trouver le terme de preuve — s'il existe — de toute formule de la logique minimale propositionnelle.

Néanmoins, il est temps de se familiariser avec des outils plus élaborés, dont l'usage s'imposera quand nous nous confronterons à des énoncés de théorèmes plus complexes.

P	Q	R	P⇒Q	Q⇒R	P⇒R	(Q⇒R)⇒P⇒R	(P⇒Q)⇒(Q⇒R)⇒P⇒R
f	*f*	*f*	*v*	*v*	*v*	*v*	*v*
f	*f*	*v*	*v*	*v*	*v*	*v*	*v*
f	*v*	*f*	*v*	*f*	*v*	*v*	*v*
f	*v*	*v*	*v*	*v*	*v*	*v*	*v*
v	*f*	*f*	*f*	*v*	*f*	*f*	*v*
v	*f*	*v*	*f*	*v*	*v*	*v*	*v*
v	*v*	*f*	*v*	*f*	*f*	*v*	*v*
v	*v*	*v*	*v*	*v*	*v*	*v*	*v*

FIGURE 4.1 – Une table de vérité

```
(H)  : Supposons P→ Q
(H'): Supposons Q→ R
(p):  Supposons P
 (1)  : En appliquant H à p on prouve Q
 (2)   : En appliquant H'à (1) on prouve R
• : Par raisonnement hypothétique sur H,H' et p sur (2),
 on prouve (P→ Q)→(Q→ R)→ P→ R
```

FIGURE 4.2 – Une preuve en français

P	Q	P⇒Q	(P⇒Q)⇒P	((P⇒Q)⇒P)⇒P
f	*f*	*v*	*f*	*v*
f	*v*	*v*	*f*	*v*
v	*f*	*f*	*v*	*v*
v	*v*	*v*	*v*	*v*

FIGURE 4.3 – Une vue classique de la formule de Peirce

4.9.1 Les tactiques `cut` et `assert`

La tactique `cut`

Considérons un but b de la forme $\Gamma \stackrel{?}{\vdash} P$. Parmi les choix tactiques que peut prendre l'usager, on peut envisager le schéma suivant (où Q est une proposition) :
– je peux prouver l'implication $Q \rightarrow P$,
– je peux prouver Q.

Ceci correspond à un plan de travail consistant à résoudre les buts $\Gamma \stackrel{?}{\vdash} Q \rightarrow P$ et $\Gamma \stackrel{?}{\vdash} Q$. Si t_1 et t_2 sont les solutions respectives de ces deux buts, alors l'application $t_1\ \ t_2$ est une solution du but initial.

D'un point de vue pratique, on utilise en *Coq* la tactique " `cut` Q ".

Prenons un exemple simple. On considère une section débutant par quatre hypothèses sous lesquelles nous voulons prouver la proposition `Q`.

```
Section section_for_cut_example.
  Hypotheses (H : P→Q)
             (H0 : Q→R)
             (H1 : (P→R)→T→Q)
             (H2 : (P→R)→T).
```

Afin de prouver `Q`, on pourrait appliquer `H1`, ce qui engendrerait deux sous-buts, d'énoncés respectifs `P→R` et `T`. Or la résolution du second sous-but (par " `apply H2` ") demande à nouveau de prouver `P→R`. D'où un surcroît de travail.

L'utilisation d'une *coupure* permet dans un premier temps d'introduire dans le contexte une hypothèse `H3:P→R`, qui facilite la preuve de `Q`, puis de montrer `P→R` *une seule fois*.

En résumé, la preuve de `Q` a la structure suivante, typique de l'utilisation de `cut` :

1. Appel de la tactique " `cut (P→R)` ",
2. introduction d'une hypothèse `(H3: P→R)` ; dans ce nouveau contexte, preuve du but initial `Q`,
3. preuve de la proposition `P→R`

Voici le script de preuve :

```
Theorem cut_example : Q.
Proof.
 cut (P→R).
  ...
  ============================
   (P→R)→Q

subgoal 2 is:
  P→R

 intro H3.
```

```
 apply H1;[assumption | apply H2; assumption].
 intro; apply H0; apply H; assumption.
Qed.
```

L'impression du terme de preuve associé à `cut_example` montre un ζ-radical correspondant au « lemme » `H3`.

```
Print cut_example.
```
cut_example = let H3 := fun H3:P ⇒ H0 (H H3) in H1 H3 (H2 H3)
: Q

Remarque 4.2 Dans l'exemple ci-dessus, on suppose que `H3` ne présente pas d'autre intérêt que de « factoriser » la preuve considérée. Si l'on a besoin d'un résultat de portée plus globale, il faut alors prouver un lemme qui pourra être utilisé ultérieurement. Notons que *Coq* permet l'édition simultanée de plusieurs preuves ; on peut donc démarrer la preuve d'un théorème, l'interrompre pour prouver un lemme imprévu, puis — une fois ce lemme prouvé — continuer la preuve du théorème principal (voir les détails dans le manuel de référence).

Exercice 4.5 Donner une preuve de `cut_example` sans utiliser `cut` et observer le terme de preuve construit.

4.9.1.1 La tactique `assert`

La variante `assert` permet d'aborder les deux buts dans l'ordre opposé : d'abord démontrer Q, puis l'utiliser. Par ailleurs, la tactique " `assert H:`Q " provoque immédiatement l'introduction de Q comme hypothèse de nom `H` dans le deuxième but. Cette variante `assert` peut être utile pour donner aux scripts de preuve une organisation plus lisible. Elle favorise entre autres la démonstration par « chaînage avant » : si, pour prouver une proposition B, on souhaite prouver (dans l'ordre) les propositions A_1, A_2, …, A_n, on peut développer un script de la forme suivante :

```
assert H1: A1.
 preuve de A1
assert H2: A2.
 preuve de A2
...
assert Hn: An.
 preuve de An

preuve de B
```

4.9.2 Tactiques et automatismes

La tactique `auto`

Toutes les preuves effectuées manuellement dans ce chapitre peuvent l'être par un simple appel à la tactique `auto`. Nous en présentons une version réduite, sans les capacités de réglage fin qui seront décrites dans la section 8.2.

Dans sa version réduite, on peut considérer `auto` comme une combinaison récursive d'`assumption`, d'`intros` et d'appels à `apply` v où v est une variable locale.

La tactique `auto` peut prendre en paramètre la profondeur maximale de recherche d'une solution. Par défaut, cette valeur est de 5.

On peut montrer que si le but $[], \Gamma \vdash^{?} A$ admet une solution (avec les règles de typage vues jusqu'à présent) alors il existe un n tel que la tactique " `auto` n " construise une solution pour ce but.

Voici un exemple d'utilisation d'`auto`.

```
Theorem triple_impl2 : (((P→Q)→Q)→Q)→P→Q.
Proof.
 auto.
Qed.
```

Propriétés de la tactique `auto`

La tactique `auto` possède quelques propriétés qu'il est bon de connaître :

- Tout d'abord, `auto` n'échoue jamais : appliquée à un but b, soit b est résolu par `auto`, soit b est inchangé. De plus, la réussite d'`auto` doit être totale et immédiate ; son application ne peut résulter en la génération de nouveaux sous-buts.
- Une composition de la forme " *`tac`* `; auto` ", appliquée à un but b, ne laissera actifs que les sous-buts engendrés par *tac* non solubles par `auto`. Par exemple, soit à prouver la proposition suivante :
  `((P→Q→P)→(Q→R)→((((P→Q)→Q)→Q)→P→Q)→R)→R`
  La tactique composée " `intro H; apply H; auto; clear H` " ne laissera comme sous-but que l'implication `Q→R` (si `auto` ne peut résoudre ce but dans le contexte courant.)

Exercice 4.6 * Construire un but (en logique minimale propositionnelle) résolu par " `auto 6` ", mais pas par `auto` (c'est à dire " `auto 5` "). Généraliser.

La tactique `trivial`

La tactique `trivial` est encore plus limitée que la tactique `auto` car elle n'utilise qu'un fragment de la base de théorèmes disponible. Il est néanmoins intéressant d'utiliser cette tactique de préférence à `auto` car elle rend le caractère facile de certaines étapes de preuve plus explicite, ce qui améliore la lisibilité des scripts de démonstration.

4.10 Un nouveau type d'abstractions

Tout le travail de preuve que nous avons effectué s'est fait dans le cadre d'une section `Minimal_propositional_logic` contenant les déclarations des variables propositionnelles `P`, `Q`, `R`, `S` et `T`. Nous avons vu d'autre part que le mécanisme de fermeture de section s'accompagne de la création d'abstractions. Que se passe-t-il quand nous fermons notre section de travail ?

```
End Minimal_propositional_logic.
```

Utilisons la commande `Print` pour observer sous quelle forme sont exportés les théorèmes démontrés à l'intérieur de la section précédente (le théorème `imp_dist` a été prouvé dans la section 4.2.2) :

```
Print imp_dist.
imp_dist = fun (P Q R:Prop)(H:P→Q→R)(H0:P→Q)(H1:P) ⇒
           H H1 (H0 H1)
: ∀ P Q R:Prop, (P→Q→R)→(P→Q)→P→R
Argument scopes are [type_scope type_scope type_scope _ _ _]
```

Les déclarations de variables propositionnelles se retrouvent dans les abstractions des termes de preuve de `triple_impl2` et `imp_dist` ; on remarquera de nouveau que, sur les 5 variables propositionnelles déclarées localement, seules celles ayant au moins une occurrence libre dans le terme de preuve considéré font l'objet d'une abstraction.

Quant aux énoncés de ces théorèmes, la transformation qu'ils subissent à l'exportation exprime le fait que, par exemple, la distributivité de l'implication est prouvée *pour n'importe quelles propositions `P`, `Q`, et `R`*. En d'autres termes, le théorème `imp_dist` s'applique *en tant que fonction* à n'importe quel triplet de propositions, comme le montre la session suivante :

```
Section using_imp_dist.
 Variables (P1 P2 P3 : Prop).
 Check (imp_dist P1 P2 P3).
 imp_dist P1 P2 P3
   : (P1→P2→P3)→(P1→P2)→P1→P3

 Check (imp_dist (P1→P2)(P2→P3)(P3→P1)).
 imp_dist (P1→P2)(P2→P3)(P3→P1)
     : ((P1→P2)→(P2→P3)→P3→P1)
       →((P1→P2)→P2→P3)→(P1→P2)→P3→P1
End using_imp_dist.
```

Puisque cette fonction permet d'obtenir une instance de la distributivité *pour toutes les propositions P, Q et R*, le type de cette fonction doit en fait être lu comme une *quantification universelle* sur les propositions (on parle alors de quantification du second ordre).

Ce nouveau constructeur de type et son interprétation comme une construction de quantification universelle seront les objets du prochain chapitre.

Chapitre 5

Le produit dépendant *ou* la boîte de Pandore

Jusqu'à présent, nous nous sommes restreints au λ-calcul simplement typé, ce qui limitait considérablement la puissance d'expression des spécifications et propositions considérées. Cette limitation disparaît avec l'étude d'une nouvelle construction de types, appelée *produit dépendant*. Cette construction généralise la flèche $A{\rightarrow}B$, étudiée dans les deux chapitres précédents (types fonctionnels ou implication, suivant la sorte de A et B). Elle nous permet de travailler avec des fonctions dont le type du résultat dépend de celui de leur argument. On pense bien sûr au polymorphisme de certains langages de programmation, et surtout au polymorphisme paramétrique des langages de la famille *ML*.

Le produit dépendant permet aussi d'exprimer la quantification universelle, tant sur des expressions que sur des types ; citons par exemple le théorème affirmant que la relation $\leq$ est réflexive sur $\mathbb{N}$, ainsi que la commutativité de la disjonction :

$$\forall n : \texttt{nat}.\ n \leq n$$

$$\forall P, Q : \texttt{Prop}.\ P \vee Q {\rightarrow} Q \vee P$$

Nous verrons également que le produit dépendant nous permet d'exprimer des assertions sur les programmes, et réciproquement, d'exprimer des spécifications complexes comme par exemple :

> « une fonction qui à tout entier n, tel que $n > 1$, associe le plus petit nombre premier diviseur de n »

Rappelons à ce propos que les spécifications vues au chapitre 3 ne nous autoriseraient dans ce cas que la « sous-spécification » `nat`$\rightarrow$`nat`, c'est à dire celle des fonctions qui à tout entier naturel associent un entier naturel, sans plus de précision.

Dans un premier temps, nous allons justifier la nécessité d'étendre les constructions de types vues jusqu'à présent ; cette étude débouchera sur une extension du λ-calcul simplement typé. Nous nous attacherons à décrire comment s'utilise cette construction, du point de vue des règles de typage des termes. La fin du chapitre sera consacrée aux règles du Calcul des Constructions contrôlant la formation de types dépendants.

5.1 Éloge de la dépendance

Considérons la situation à l'issue des deux précédents chapitres. Nous avons un formalisme commun pour, d'une part, déclarer et construire des programmes de type donné, d'autre part pour énoncer et prouver des propositions. Il ne nous est pas encore possible de considérer des propositions portant sur ces programmes, ni de construire des programmes certifiés possédant telle ou telle propriété.

Nous allons pallier cette insuffisance en étendant le système de types considéré ; cette extension présentera plusieurs aspects :

– avec de nouvelles possibilités de construire des types de la forme $A{\rightarrow}B$, nous pourrons construire des fonctions pouvant prendre des types comme arguments, ou dont le résultat est un type ; nous pourrons ainsi utiliser des familles entières de types.
– à l'aide d'un mécanisme de liaison de variable et de substitution, nous pourrons considérer des fonctions dont le type du résultat peut varier en fonction des arguments ; nous exprimerons bien sûr le polymorphisme paramétrique (à la *ML*), mais nous ne nous limiterons pas à cet aspect.
– la conjonction des deux aspects précédents nous permettra de définir des types comme résultat d'applications de fonctions à des termes appropriés ; ces types — dont l'expression même contient en général ces termes — sont qualifiés de *dépendants*.

Quelques exemples simples nous serviront de fil conducteur, afin de justifier et rendre intuitives les définitions à venir.

5.1.1 De nouveaux types flèches

Nous présentons à l'aide d'exemples l'intérêt d'autoriser la construction de types de la forme $A{\rightarrow}B$, qui viendront s'ajouter aux constructions décrites page 78 (règle **Prod**).

Le type des prédicats

Supposons que nous voulions étudier une fonction `prime_divisor` permettant de trouver un diviseur premier de tout entier naturel supérieur ou égal à 2. Le système de types simples du chapitre 3 ne nous autoriserait que la spécification `nat`→`nat`.

Nous nous ne intéressons pas dans ce chapitre à la construction d'une telle fonction, mais seulement à sa spécification et aux possibilités de décrire son comportement.

Soit n un terme de type `nat`, pour raisonner sur `prime_divisor`, il est nécessaire de pouvoir exprimer en *Coq* les propositions suivantes :

« $2 \leq n$ »

« `prime_divisor` n est un nombre premier »

« `prime_divisor` n est un diviseur de n »

Un moyen très simple pour construire ces propositions est de les considérer comme le résultat de l'application de fonctions prenant en paramètre un ou deux entiers naturels et retournant une proposition. De telles fonctions sont appelées *prédicats*.

Le prédicat `le` est défini dans l'environnement initial de *Coq* ; sa définition est étudiée en 9.1.1. Si t_1 et t_2 sont deux termes de type `nat`, alors l'inégalité $t_1 \leq t_2$ est représentée en *Coq* par l'application " `le` t_1 t_2 " (également notée " $t_1 \leq t_2$ " dans la portée `nat_scope`). Ceci est possible grâce à deux propriétés :

1. le type `nat→nat→Prop` est admissible dans le Calcul des Constructions,
2. la constante `le` est déclarée de ce type.

Nous verrons plus loin la règle permettant de construire des types de prédicats ; il nous suffit pour le moment d'admettre que les types `nat→nat→Prop` et `nat→Prop` existent en *Coq*, et de déclarer deux nouveaux prédicats :

– `divides:nat→nat→Prop` ; la proposition " `divides` t_1 t_2 " se lit « t_1 est un diviseur de t_2 »,

– `prime:nat→Prop` ; la proposition " `prime` t_1 " se lit « t_1 est un nombre premier ».

Les propositions " `divides` t_1 t_2 " et " `prime` t_1 ", obtenues par application d'un prédicat, sont donc des types dépendants.

Remarquons que nous n'avons pas *défini* `prime_divisor`, `divides`, ni `prime`. Nous considérons ces identificateurs seulement comme des briques de base pour former des propositions plus complexes ; seul nous importe le fait que `divides` et `prime` sont des prédicats ; un traitement réaliste des nombres premiers se trouve en 17.2.

La constante `le` est déjà définie, et nous nous contentons de déclarer les trois autres identificateurs dont nous avons besoin.

```
Require Import Arith.

Parameters (prime_divisor : nat→nat)
           (prime : nat→Prop)
           (divides : nat→nat→Prop).
```

L'exemple suivant montre comment écrire en *Coq* [1] les propositions « l'appel de `prime_divisor` sur 1917 retourne un nombre premier », « l'appel de `prime_divisor` sur 1917 retourne un diviseur de 1917 », ainsi que le prédicat « être un multiple de 3 ». Tous ces jugements de typage sont obtenus à l'aide de la règle **App** (page 46).

1. La majeure partie de ce chapitre se placera dans une portée globale `nat_scope`

```
Open Scope nat_scope.

Check (prime (prime_divisor 220)).
```
prime (prime_ divisor 220) : Prop

```
Check (divides (prime_divisor 220) 220).
```
divides (prime_ divisor 220) 220 : Prop

```
Check (divides 3).
```
divides 3 : nat→Prop

Types de données paramétrés

Considérons la déclaration d'un type de donnée « mot binaire » ; il est naturel de pouvoir considérer un type paramétré par la taille des mots. D'une façon similaire aux prédicats, il est intéressant de pouvoir disposer du type `nat→Set`, et de déclarer `binary_word` comme une fonction de ce type ; si n est un entier naturel, nous interpréterons “ `binary_word` n ” comme le type des mots de taille n ; nous pouvons alors définir des types associés à des tailles particulières par simple passage de paramètre.

Ici encore, nous nous contentons de déclarations ; l'exercice 7.47 est consacré à la définition du type `binary_word`.

```
Parameter binary_word : nat→Set.
```
binary_ word is assumed

```
Definition short : Set := binary_word 32.
```
short is defined

```
Definition long : Set := binary_word 64.
```
long is defined

Formation de types dépendants

Les deux types d'exemples ci-dessus appliquent l'amendement suivant à la règle **Prod**, qui, en conjonction avec la règle **App**, nous permettra de construire des *types dépendants*.

$$\textbf{Prod-dep}\quad \frac{E,\Gamma \vdash A : \texttt{Set} \quad E,\Gamma \vdash B : \texttt{Type}}{E,\Gamma \vdash A \to B : \texttt{Type}}$$

En effet, la formation de prédicats se fait en prenant $B = \texttt{Prop}$ et celle de types de données paramétrés avec $B = \texttt{Set}$. De plus, cette règle nous indique que ces types ont pour sorte `Type`.

Exercice 5.1 Considérer les types suivants, et vérifier qu'ils sont bien admissibles dans le Calcul des Constructions. Pour chacun d'entre eux, trouver un exemple naturel d'habitant, défini de façon intuitive comme dans les exemples ci-dessus.

– `(nat→nat)→Prop`
– `(nat→nat)→(nat→nat)→Prop`
– `nat→nat→Set`

5.1.1.1 Le type des connecteurs logiques

Supposons que nous voulions exprimer la propriété suivante :

> « le nombre obtenu en appelant `prime_divisor` sur le nombre 220
> est un diviseur premier de 220. »

Il est simple de l'exprimer comme la conjonction des deux propositions ci-dessous :

– `prime (prime_divisor 220)`
– `divides (prime_divisor 220) 220`.

De même, la proposition « 33 n'est pas un nombre premier » est la négation de la proposition “ `prime 33` ”.

La conjonction de deux propositions est une proposition ; on peut donc considérer la conjonction comme une fonction prenant deux propositions en argument, etn retournant une proposition. En *Coq*, on lui associe une constante `and` de type `Prop→Prop→Prop`. La disjonction est représentée par la constante `or` du même type, et la négation par la constante `not` de type `Prop→Prop`. Ces trois constantes sont définies dans le prélude de *Coq*, et commentées pages 128, 149 et 249.

La règle d'application nous permet alors de former de nouvelles propositions par conjonction ou négation d'autres propositions. *Coq* permet d'utiliser respectivement les syntaxes P/\\Q, P\\/Q et ~P pour “ `and` P Q ”, “ `or` P Q ” et “ `not` P ”, mais nous utiliserons les notations $P{\wedge}Q$ et $P{\vee}Q$ pour la conjonction et la disjonction dans cet ouvrage.

```
Check (not (divides 3 81)).
```
~divides 3 81 : Prop

```
Check (let d := prime_divisor 220 in prime d ∧ divides d 220).
```
let d := prime_ divisor 220
in prime d ∧ divides d 220 : Prop

Remarque 5.1 Dans l'exemple précédent, nous avons utilisé pour la première fois la règle **Let-in** pour construire une proposition (c'est-à-dire un type) et non une expression dans un langage fonctionnel (comme en page 51). Les types A et B paramètrant cette règle peuvent être de n'importe quelle sorte ; dans cet exemple, B est la sorte `Prop`, elle même de sorte `Type`.

Opérateurs sur les types de données

La plupart des langages de programmation permettent de construire des types de données en appliquant des *opérateurs* à des types plus simples. C'est le cas par exemple des types suivants :

- « le type des couples dont la première composante est de type A et la seconde de type B »,
- « le type des listes dont tous les éléments sont de type A ».

Il est commode d'exprimer ces types sous la forme d'applications :

- " `prod` A B "
- " `list` A "

Ceci est possible en admettant les deux types `Set`→`Set`→`Set` et `Set`→`Set`, ce qui autorise les déclarations de `prod` et `list`.

Avec la règle **App**, nous obtenons bien que si A et B sont deux spécifications, alors " `list` A " et " `prod` A B " sont des spécifications ; la première est celle des listes dont les éléments sont de type A, la seconde est associée au produit cartésien $A \times B$; notons que *Coq* utilise la notation[2] A`*`B pour abréger " `prod` A B ". La définition précise de `prod` est décrite page 208, celle des listes page 204 ; ces types sont définis dans *Coq* respectivement dans la bibliothèque initiale `Init` et dans le module `List`.

Ces constructions peuvent par exemple servir à déclarer deux nouvelles fonctions :

```
Require Import List.
Parameters (decomp : nat → list nat)(decomp2 : nat→nat*nat).
decomp is assumed
decomp2 is assumed

Check (decomp 220).
decomp 220 : list nat

Check (decomp2 284).
decomp2 284 : (nat*nat)%type
```

Types d'ordre supérieur

Le type des connecteurs et celui des opérateurs comme `prod` et `list` peuvent se construire à l'aide de la règle suivante, nouvelle extension de `Prod`. Les types ainsi formés sont qualifiés de « types d'ordre supérieur ».

$$\textbf{Prod-sup}\quad \frac{E,\Gamma \vdash A : \mathtt{Type} \quad E,\Gamma \vdash B : \mathtt{Type}}{E,\Gamma \vdash A{\to}B : \mathtt{Type}}$$

Cette règle nous indique en outre que les types `Prop`→`Prop`, `Prop`→`Prop`→`Prop`, `Set`→`Set`, etc., sont de sorte `Type`.

2. L'opérateur '`*`' est donc surchargé ; la portée `type_scope`, de clef `%T`, permet d'associer à '`*`' le produit cartésien ; cette portée est implicitement considérée dès que *Coq* attend une contrainte de type (après '`:`')

5.1.2 Les liaisons nécessaires

Les possibilités ouvertes avec les règles **Prod-dep** et **Prod-sup**, bien que très prometteuses, conduisent vite à la frustration. Par exemple, nous ne pouvons pas exprimer des propositions comme « pour tout entier naturel n, si $2 \leq n$, alors `prime_divisor` n retourne un diviseur premier de n », alors que les mathématiciens utilisent très bien le quantificateur universel :

$$\forall n \in \mathbb{N}\ (2 \leq n \Rightarrow \texttt{prime}(\texttt{prime\_divisor}\ (n)) \wedge \texttt{prime\_divisor}(n)|n)$$

De la même façon, le produit cartésien A*B et le type “ `list` A ” posent des problèmes : le constructeur `pair` de couples et les projections `fst` et `snd`, ainsi que la liste vide `nil` et le constructeur de listes `cons` doivent avoir un type *polymorphe*, comme en *OCAML*, afin d'être appliquables quels que soient les types A et B. Or ce polymorphisme ne s'exprime pas avec le typage simple des deux chapitres précédents.

Il est classique en théorie des types (voir Mitchell [67] et Girard-Lafont-Taylor [48]) d'indiquer le polymorphisme par une notation de produit indiquant des variables « universelles » de types, introduites par le symbole Π. Ainsi, le type du constructeur de liste `cons` sera le produit :

$$\Pi A : \texttt{Set}.\ A \rightarrow \texttt{list}\ \ A \rightarrow \texttt{list}\ \ A$$

Le type de la première projection `fst` sera un double produit :

$$\Pi A : \texttt{Set}.\ \Pi B : \texttt{Set}.\ \texttt{A*B} \rightarrow A$$

Pour terminer ces exemples, considérons quel type devrait avoir une fonction de concaténation de mots binaires. Si l'on concatène un mot de longueur n et un mot de longueur p, le résultat doit être de longueur $n + p$. La concaténation doit s'adapter à toutes les situations, son type pourrait être le produit suivant :

$$\begin{array}{ll}\Pi n : & \texttt{nat}.\ \Pi p : \texttt{nat}. \\ & \texttt{binary\_word}\ \ n \rightarrow \texttt{binary\_word}\ \ p \rightarrow \texttt{binary\_word}\ \ (n + p)\end{array}$$

On remarque que, dans ce dernier exemple comme pour la quantification universelle sur $\mathbb{N}$, les variables n et p ont pour type une spécification, alors que dans le cas du polymorphisme, les variables A et B ont pour type l'univers `Type`. La puissance d'expression et la simplicité d'utilisation du Calcul des Constructions viennent de ce que le même mécanisme et les mêmes notations sont utilisés dans des niveaux d'abstraction très variés.

La construction de *produit dépendant* propose une formalisation unifiée de la quantification universelle et des types produits.

5.1.3 Une nouvelle construction

Définition 5.1 (Produit dépendant)

Un *produit dépendant* est un type de la forme “ $\forall v : A, B$ ”, où A et B sont des types et v est une variable liée dont la portée est le type B. La variable v peut avoir des occurrences libres dans B. Le produit dépendant ainsi écrit se lit « pour tout v de type A, B ».

Sur un clavier, le produit dépendant $\forall v : A, B$ s'écrit “ `forall` v : A, B ”. Nous utiliserons néammoins le symbole $\forall$ dans cet ouvrage, en lieu et place du mot-clef “ `forall` ”.

Remarquons qu'un produit dépendant est à la fois un type et un terme du Calcul des Constructions ; à ce titre, tout produit dépendant doit avoir une sorte. Cette sorte, ainsi que les conditions de formation de produits, seront déterminées à la fin de ce chapitre.

Remarque 5.2 Le terme « produit dépendant » peut surprendre ; il faut y voir une analogie avec les généralisations du produit cartésien vu en mathématiques, où nous pouvons former un produit — noté « $\prod_{i \in I} A_i$ » —, d'ensembles indexés par une famille quelconque ; un élément d'un tel ensemble est bien une fonction qui à tout i dans I associe un élément de l'ensemble A_i correspondant. Si la famille I est réduite au doubleton $\{1, 2\}$, nous retrouvons immédiatement le produit cartésien $A_1 \times A_2$.

La différence essentielle entre le produit dépendant de *Coq* et le produit ci-dessus est que les mathématiques traitent d'*ensembles* et le Calcul des Constructions de *types*.

Une autre façon d'appréhender le produit dépendant est la quantification universelle ; celle-ci sera explorée en profondeur dans ce chapitre et tous les suivants.

Notations abrégées

Les produits dépendants imbriqués disposent d'une notation abrégée similaire à celle des abstractions, en utilisant des parenthèses pour séparer les « liaisons » portant sur des types différents.

Par exemple, les écritures suivantes sont tout à fait équivalentes :

- $\forall v_1 : A, \forall v_2 : A, \forall v_3 : B, \forall v_4 : B, U$
- $\forall v_1\ v_2 : A, \forall v_3\ v_4 : B, U$
- $\forall\ (v_1\ v_2 : A)\ (v_3\ v_4 : B), U$

Exemples

Nous présentons la notation *Coq* pour les types cités dans cette introduction. La constante `prime_divisor_correct` est le nom d'un théorème de correction de la fonction `prime_divisor`, dont nous ne donnons pas la preuve (à faire en exercice après avoir acquis suffisamment d'expérience sur les problèmes arithmétiques en *Coq*).

```
Check prime_divisor_correct.
```

prime_ divisor_ correct

: ∀ n:nat, 2≤ n → let d:= prime_ divisor n in prime d ∧ divides d n

```
Check cons.
```
cons : ∀ A:Set, A → list A → list A

```
Check pair.
```
*pair : ∀ A B :Set, A→B→A*B*

```
Check (∀A B :Set, A→B→A*B).
```
*∀ A B :Set, A→B→A*B : Type*

```
Check fst.
```
*fst : ∀ A B:Set, A * B → A*

Prenons par exemple le type de `pair` ; la variable `A`, liée dans ce produit dépendant, possède deux occurrences libres dans le type `∀B:Set, A→B→A*B`.

Remarque 5.3 (Produit dépendant et α-conversion) Comme pour toute syntaxe à base de liaison de variables, nous considérons comme égaux deux termes équivalents par α-conversion. Ainsi les deux produits ci-dessous sont deux façons d'écrire le même type :

- `∀ U V:Set, U →V →U*V`
- `∀ A B:Set, A →B →A*B`

5.2 Règles de typage associées au produit dépendant

Nous présentons les règles permettant de former ou d'utiliser les termes dont le type est un produit dépendant. Certaines tactiques de *Coq*, déjà présentées dans le chapitre 4, seront modifiées pour prendre en compte cette nouvelle construction. Les règles de typage contrôlant la formation de produits dépendants seront présentées en section 5.3.1.

L'idée maîtresse de ces règles est que le produit " $\forall v\!:\!A,\, B$ " est le type des fonctions qui à tout v de type A associent un terme de type B, ce dernier pouvant dépendre de v. L'utilisation d'un terme t de ce type se fait alors par application fonctionnelle. Ceci est exprimé dans l'adaptation aux produits dépendants de la règle **App**.

5.2.1 Règle d'application

La règle suivante diffère de la règle **App** de la page 46 par la présence d'une opération de substitution exprimant la dépendance entre l'argument de l'application et le type du résultat :

$$\textbf{App}\quad \frac{E,\Gamma \vdash t_1 : \forall v\!:\!A,\, B \quad E,\Gamma \vdash t_2 : A}{E,\Gamma \vdash t_1\ \ t_2 : B\{v/t_2\}}$$

5.2.1.1 Premiers exemples

Nous montrons quelques exemples d'utilisation de la règle **App** sur des produits dépendants.

Rappelons que l'ordre total sur $\mathbb{N}$ est décrit par le prédicat `le` de type `nat`$\rightarrow$`nat`$\rightarrow$`Prop`. Les deux théorèmes [3] suivants sont la traduction en *Coq* des propriétés bien connues ci-dessous :

$$\forall n \in \mathbb{N}.\ n \leq n$$
$$\forall n\, m \in \mathbb{N}.\ n \leq m \Rightarrow n \leq m+1$$

```
Check le_n.
```
le_n : ∀ n:nat, n ≤ n

```
Check le_S.
```
le_S : ∀ n m:nat, n ≤ m → n ≤ S m

Le théorème `le_n` est bien une fonction qui saura toujours fabriquer un habitant du type " `n ≤ n` ", c'est-à-dire un habitant d'un type différent pour chaque valeur différente de `n`.

Par exemple, une preuve de $36 \leq 36$ s'obtient bien par application de `le_n` au terme `36` :

```
Check (le_n 36).
```
le_n 36 : 36 ≤ 36

De même, `le_S` est une fonction qui transforme tout terme de preuve de " `n ≤ m` " en un terme de preuve de " `n ≤ S m` " ; plus précisément, cette fonction a trois arguments : deux arguments `n` et `m` de type `nat` et un argument de type " `n ≤ m` ".

Par exemple, l'inégalité $36 \leq 38$ se démontre en *Coq* en construisant un terme de preuve composé à l'aide des constantes `le_n` et `le_S` ; on notera la présence du caractère « joker » '`_`', destiné à remplacer tout argument que le système *Coq* est capable d'inférer à partir du contexte ; toutes les occurrences de ce caractère dans un terme sont indépendantes.

Dans notre exemple, il y a 4 occurrences de '`_`' ; les deux occurrences les plus à droite sont remplacées par `36`, grâce au type de " `le_n 36` " et de `le_S`. Les deux occurrences les plus à gauche de '`_`' sont déterminées de façons similaire.

```
Check (le_S _ _ (le_S _ _ (le_n 36))).
```
le_S 36 37 (le_S 36 36 (le_n 36)) : 36 ≤ 38

Ce terme de preuve est représenté de façon graphique par l'arbre de la figure 5.1.

3. En fait, nous verrons page 242 que ces deux théorèmes font partie de la *définition inductive* de `le`.

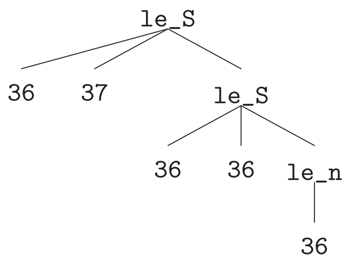


FIGURE 5.1 – Une preuve de l'inégalité $36 \leq 38$

Ainsi, l'application d'un théorème dont le type est un produit dépendant se réduit-elle à l'application de ce théorème (vu comme une fonction) à une valeur particulière. C'est donc la même chose d'appliquer une fonction et d'appliquer un théorème. Nous retrouvons bien l'isomorphisme de Curry-Howard.

De la même façon, l'application de `prime_divisor_correct` (page 108), au nombre 220, retourne un théorème exprimant que le résultat de l'application de `prime_divisor` à 220 est bien un diviseur premier de ce nombre (à condition de vérifier l'inégalité $2 \leq 220$, ce qui se fait sans problème).

```
Check (prime_divisor_correct 220).
prime_divisor_correct 220
   : 2 ≤ 220→let d:= prime_divisor 220 in prime d ∧ divides d 220
```

La série d'exemples suivants aborde le polymorphisme. Considérons une fonctionnelle permettant d'itérer une fonction (unaire, d'un type quelconque vers lui même). Nous pourrons la définir en *Coq* en utilisant les outils abordés dans la section 7.3.3. Pour le moment, considérons simplement son type :

```
iterate : ∀A:Set, (A→A)→nat→A→A.
```

La fonction `iterate` demande comme arguments (dans l'ordre) :

1. un type A,
2. la fonction à itérer, de type $A \rightarrow A$,
3. le nombre d'itérations à effectuer, de type `nat`,
4. l'élément de départ, à partir duquel l'application de la fonction va être itérée, de type A.

Nous illustrons le mécanisme de l'application en montrant les 4 possibilités de fournir partiellement des arguments à `iterate`. Notons encore l'utilisation du symbole "`_`" dès qu'un argument d'`iterate` peut être déterminé par le contexte.

```
Check (iterate nat).
iterate nat : (nat→nat)→nat→nat→nat

Check (iterate _ (mult 2)).
```

iterate nat (mult 2) : nat→nat→nat

```
Check (iterate _ (mult 2) 10).
```
iterate nat (mult 2) 10 : nat → nat

```
Check (iterate _ (mult 2) 10 1).
```
iterate nat (mult 2) 10 1 : nat

```
Eval compute in (iterate _ (mult 2) 10 1).
```
= 1024 : nat

En revanche, l'exemple suivant montre un conflit de types : le premier argument fourni à `iterate` fait attendre un quatrième argument de type `Z`. Or, c'est `36%N` qui lui est transmis.

```
Check (iterate Z (Zmult 2) 5 36).
```
Error: The term 36 has type nat while it is expected to have type Z

Remarque 5.4 Nous remarquons que la fonction `iterate` a bien quatre arguments, dont le premier est un type et le reste des expressions, et que le passage d'argument s'utilise pour instancier des types polymorphes.

Ce fait appelle un commentaire : le mécanisme permettant d'instancier une variable de type dans une construction polymorphe est le simple appel de fonction. Le lecteur ayant pratiqué plusieurs langages de programmation pourra comparer cette approche avec celles suivies dans les langages suivants :

***OCAML* :** l'instanciation des paramètres de type (`'a`, `'b`, ...) se fait par inférence de type ; les contraintes provenant des constantes présentes dans l'expression courante peuvent forcer cette instanciation,

***C* :** on pratique des jeux bien dangereux à coups de coercitions (« *casts* ») et de `void *` (type des pointeurs génériques),

***Java* :** le polymorphisme s'exprimant à l'aide de la notion de classe, cette instanciation se fait en utilisant l'héritage.

L'approche utilisée par *Coq* est bien plus simple, quoique plus abstraite : les types sont des arguments de fonctions au même titre que des entiers, listes, fonctions, etc. Pour reprendre un anglicisme fort à la mode depuis *Scheme* « les types sont des citoyens de première classe ».

Pour finir, nous reprenons notre exemple de types de données dépendants ; nous remarquons que la réduction de “ `32 + 32` ” en `64` n'est pas automatiquement effectuée. Cet aspect sera traité en section 5.2.4.

```
Check (binary_word_concat 32).
```
binary_word_concat 32
: ∀ p:nat, binary_word 32 → binary_word p → binary_word (32+p)

```
Check (binary_word_concat 32 32).
```

binary_word_concat 32 32
  : binary_word 32 → binary_word 32 → binary_word (32+32)

5.2.2 Règle d'abstraction

De même que pour l'application, nous devons généraliser la règle présentée page 48. En fait, nous devons prendre en compte la possibilité que, dans un contexte déclarant v de type A, le type B d'un terme t contienne des occurrences de v. Le type de l'abstraction `fun` $(v:A) \Rightarrow t$ est alors $\forall\, v{:}A,\, B$.

$$\textbf{Lam}\quad \frac{E, \Gamma :: (v : A) \vdash t : B}{E, \Gamma \vdash \texttt{fun}\ (v{:}A)\ \Rightarrow\ t : \forall\, v{:}A,\, B}$$

Dans cette règle le produit dépendant $\forall\, v{:}A,\, B$ *doit être construit selon les règles de bonne formation qui ne seront exposées que dans la section 5.3.1.*

Compatibilité avec le « produit non dépendant »

Si, dans les règles **App** et **Lam**, nous supposons que le type B ne contient aucune occurrence de v, la substitution de v par A dans B n'a plus lieu d'être, et on retrouve les règles **App** et **Lam** du λ-calcul simplement typé.

En conséquence, la construction $A{\rightarrow}B$ est désormais considérée comme une abréviation du produit $\forall\, v{:}A,\, B$ *si la variable v n'a pas d'occurrences libres dans B*. Un type de la forme $A{\rightarrow}B$ sera alors qualifié de *produit non dépendant* pour mettre en relief l'absence d'occurrence de la variable v dans B. *Coq* applique systématiquement cette simplification dans ses entrées-sorties.

Considérons par exemple, le type de `pair` :

```
∀A B:Set, A→B→A*B.
```

C'est en fait une abréviation du quadruple produit suivant, où les variables `a` et `b` n'apparaissent pas dans le type final `A*B`.

```
∀(A B:Set) (a:A) (b:B), A*B.
```

Dorénavant, l'appellation « produit » sera utilisée à la fois pour les produits dépendants et non dépendants.

Exemples simples

Nous donnons une première série d'exemples volontairement très simples, la puissance d'expression atteinte à l'aide du produit dépendant sera abordée dans tout le reste du chapitre.

Le premier exemple définit une fonction `twice` permettant de composer une fonction avec elle-même. Cette fonction est naturellement polymorphe et prend alors trois arguments : le premier est une spécification A, le second un terme de type $A{\rightarrow}A$ et le troisième est terme de type A.

```
Definition twice : ∀A:Set, (A→A)→A→A
          := fun A f a ⇒ f (f a).
```

```
Check (twice Z).
```
twice Z : (Z→Z)→Z→Z

```
Check (twice Z (fun z ⇒ (z*z)%Z)).
```
*twice Z (fun z:Z ⇒ (z*z)%Z) : Z→Z*

```
Check (twice _ S 56).
```
twice nat S 56 : nat

```
Check (twice (nat→nat)(fun f x ⇒ f (f x))(mult 3)).
```
twice (nat→nat)(fun (f:nat→nat)(x:nat) ⇒ f (f x))(mult 3) : nat→nat

```
Eval compute in
     (twice (nat→nat)(fun f x ⇒ f (f x))(mult 3) 1).
```
= 81 : nat

Le second exemple utilise les types de données dépendants ; on remarque que les deux premiers arguments de `binary_word_concat`, prennent pour valeur `n` ; cette valeur est automatiquement inférée à partir du type (dépendant) de `w`.

```
Definition binary_word_duplicate (n:nat)(w:binary_word n)
 : binary_word (n+n)
 :=  binary_word_concat _ _ w w.
```
binary_ word_ duplicate is defined

Enfin, l'isomorphisme de Curry-Howard nous indique que la preuve d'une proposition de la forme "$\forall x : A,\ P(x)$" prend en général la forme d'une abstraction sur x dont le corps est une preuve de $P(x)$. Par exemple, le théorème d'énoncé $\forall i : \mathbb{N}.\ i \leq i + 2$ a pour preuve l'abstraction ci-dessous.

```
Theorem le_i_SSi : ∀i:nat, i ≤ S (S i).
Proof (fun i:nat ⇒ le_S _ _ (le_S _ _ (le_n i))).
```
le_ i_ SSi is defined

Le terme de preuve de `le_i_SSi` est présenté sous forme arborescente en figure 5.2 et en langue naturelle figure 5.3.

5.2.3 Inférence de type

Certains langages de programmation, comme par exemple *ML*, disposent d'un mécanisme permettant à l'utilisateur de ne pas expliciter toutes les informations de type, laissant au compilateur le soin de les inférer. Considérons par exemple la définition ci-dessous en *OCAML* :

```
let k x y = x;;
```
val k : 'a -> 'b -> 'a = <fun>

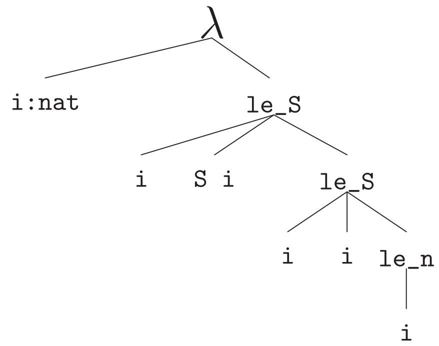


FIGURE 5.2 – Une preuve de la proposition “ $\forall$i:nat, i $\leq$ S (S i) ”

```
Soit i:nat
 (1) : En appliquant le_n à i, on prouve “ i ≤ i ”,
 (2)  : En appliquant le_S à (1) , on prouve   “ i ≤ S i ”,
 (3)  : En appliquant le_S à (2) , on prouve    “ i ≤ S S i ”,
• : Comme i était quelconque, (3)  permet de prouver
                       “ ∀i:nat,i ≤ S (S i) ”
```

FIGURE 5.3 – Une preuve en français de “ $\forall$i:nat, i $\leq$ S (S i) ”

On notera le caractère *implicite* de l'attribution de types aux variables `x` et `y`. De même, l'instanciation des paramètres de type `'a` et `'b` se fait par la reconnaissance du type des arguments de `k` :

```
k 3;;
- : '_a -> int = <fun>
k 3 true;;
- : int = 3
```

Il existe en *Coq* deux façons de simplifier l'écriture des applications de fonctions polymorphes, en utilisant les capacités d'inférence de type du système.

Le joker '_'

Nous avons déjà rencontré le symbole '_', qui peut remplacer un argument de fonction si le contexte permet de le déterminer automatiquement, auquel cas le système remplace chaque occurrence de '_' par le terme approprié. Notons qu'une abstraction “ `fun` $v_1\ v_2\ \ldots v_n$`:_` $\Rightarrow$ t ” peut se noter simplement “ `fun` $v_1\ v_2\ \ldots v_n \Rightarrow t$ ”

Par exemple, dans le dialogue suivant, nous définissons et appliquons l'opérateur `compose` de composition de fonctions ; les réponses à la commande `Check` montrent bien quels types sont inférés par *Coq* :

```
Definition compose : ∀A B C : Set, (A→B)→(B→C)→A→C
   := fun A B C f g x ⇒ g (f x).
```
compose is defined

```
Print compose.
```
compose =
fun (A B C:Set)(f:A→B)(g:B→C)(x:A) ⇒ g (f x)
: ∀ A B C:Set, (A→B)→(B→C)→A→C
Argument scopes are [type_scope type_scope type_scope _ _ _]

```
Check (fun (A:Set)(f:Z→A) ⇒ compose _ _ _ Z_of_nat f).
```
fun (A:Set)(f:Z→A) ⇒ compose nat Z A Z_of_nat f :
∀ A:Set, (Z→A)→nat→A

```
Check (compose _ _ _ Zabs_nat (plus 78) 45%Z).
```
compose Z nat nat Zabs_nat (plus 78) 45%Z
: nat

Dans l'exemple suivant, le type du terme `(le_i_SSi (1515))` permet de reconstituer les deux premiers arguments de `le_S` :

```
Check (le_i_SSi 1515).
```
le_i_SSi 1515 : 1515 ≤ 1517

```
Check (le_S _ _ (le_i_SSi 1515)).
```
le_S 1515 1517 (le_i_SSi 1515) : 1515 ≤ 1518

Il peut cependant arriver que *Coq* ne puisse déterminer de terme à associer à certains jokers, et rejette le terme en question. Voici un exemple de message d'erreur signalant ce type de situation :

```
Check (iterate _ (fun x ⇒ x) 23).
```
Error: Cannot infer a term for this placeholder

Arguments implicites

Les « arguments implicites » permettent d'éviter une surabondance de '_' dans un développement *Coq*. Il suffit de préciser à l'avance des positions d'arguments d'une fonction f pouvant naturellement être inférés à partir des autres. Dans une application de f, ces arguments seront simplement omis.

Par exemple, on peut demander que les arguments `A`, `B` et `C` de `compose`, ainsi que les arguments `n` et `m` de `le_S` soient implicites :

```
Implicit Arguments compose [A B C].
Implicit Arguments le_S [n m].

Check (compose Zabs_nat (plus 78)).
```

compose Zabs_nat (plus 78) : Z→nat

```
Check (le_S (le_i_SSi 1515)).
```
le_S (le_i_SSi 1515) : 1515 ≤ 1518

Il peut arriver que, faute d'arguments en nombre suffisant, *Coq* ne puisse inférer des arguments prédéclarés implicites ; dans un tel cas, on précise ces arguments à l'aide de la notation " *argument* :=*valeur* ".C'est le cas de l'exemple ci-dessous, où nous ne donnons qu'un seul argument à `compose` et `le_S`, dans une position déclarée implicite :

```
Check (compose (C := Z) S).
```
compose (C := Z) S : (nat→Z)→nat→Z

```
Check (le_S (n := 45)).
```
le_S (n := 45) : ∀ m:nat, 45 ≤ m → 45 ≤ S m

Plutôt que d'associer à une fonction déjà écrite des arguments implicites, il est possible de laisser *Coq* déterminer automatiquement quelles positions peuvent être rendues implicites.

Cette possibilité s'utilise comme un *mode* que l'on peut activer ou désactiver à loisir. Ce mode permet d'affecter les constructions déclarées alors que ce mode est actif, en rendant implicites les arguments des constructions qui peuvent automatiquement se déduire des autres arguments de la fonction. L'utilisateur peut à tout moment vérifier quels sont les arguments implicites associés à une constante, en utilisant la commande `Print`.

Ce mode s'active avec la commande " `Set Implicit Arguments` " et se désactive par " `Unset Implicit Arguments` ". Dans le dialogue suivant, nous utilisons ce mode pour redéfinir `compose` et définir une nouvelle fonctionnelle.

```
Reset compose.
Set Implicit Arguments.

Definition compose (A B C:Set)(f:A→B)(g:B→C)(a:A) := g (f a).

Definition thrice (A:Set)(f:A→A) := compose f (compose f f).

Unset Implicit Arguments.

Print compose.
```
compose = fun (A B C:Set)(f:A→B)(g:B→C)(a:A) ⇒ g (f a)
: ∀ A B C:Set, (A→B)→(B→C)→A→C
Arguments A, B, C are implicit
Argument scopes are [type_scope type_scope type_scope _ _ _]

```
Print thrice.
```
thrice = fun (A:Set)(f:A→A) ⇒ compose f (compose f f)

: ∀ A:Set, (A→A)→A→A
Argument A is implicit
Argument scopes are [type_scope _ _]

```
Eval cbv beta delta in (thrice (thrice (A:=nat)) S 0).
```
= 27 : nat

On remarquera dans le dernier exemple que la fonctionnelle `thrice` est utilisée dans le même terme avec deux types différents. On remarquera également que la désactivation du mode par la commande "`Unset Implicit Arguments`" n'affecte pas les positions implicites de `compose` et `thrice` ; seules les définitions à venir seront prises en compte dans cette désactivation.

Exercice 5.2 Quels arguments aurait-il fallu donner à `thrice` dans l'exemple précédent si cette fonctionnelle n'avait pas été définie sous le mode d'arguments implicites ?

5.2.4 Règle de conversion

La règle de conversion, vue en 3.5.3, n'a été jusqu'à présent utilisée qu'avec la δ-réduction, pour traiter correctement les définitions de constantes ou variables de type. La complexité des types que nous pouvons former à l'aide du produit dépendant nous oblige à considérer tous les types de réduction.

Considérons par exemple la définition suivante :

```
Definition short_concat : short→short→long
                        := binary_word_concat 32 32.
```
short_concat is defined

L'exemple page 112 nous montre que le type inféré pour le terme définissant `short_concat` n'est pas `short→short→long`. En revanche, en utilisant les δ, β et ι réductions, nous pouvons conclure que les types ci-dessous sont convertibles :

1. `short→short→long`
2. `binary_word 32→`
   `binary_word 32→`
   `binary_word (32 + 32)`

À l'aide de la règle de conversion, donnée page 64, nous pouvons conclure que `short_concat` a bien le type spécifié dans la contrainte, et en accepter la définition.

5.2.5 Produit dépendant et ordre de convertibilité

Nous illustrons par un exemple simple la relation entre le produit dépendant et l'ordre associé à la convertibilité. La constante `eq` a pour type le produit dépendant ∀ `A:Type, A→A→Prop`. Dans le module `Reals` de la bibliothèque standard, le type `R` associé aux nombres réels est de sorte `Type`.

Il est alors normal que le terme " `eq R` " soit de type `R→R→Prop`. En revanche, le type `nat` est de sorte `Set`, et par conséquent de sorte `Type`, par conséquent le terme `eq nat` est également bien formé, de type `nat→nat→Prop`.

Dans le Calcul des Constructions, aucune règle ne permet d'affirmer que la constante `eq` ait le type `∀A:Set, A→A→Prop` ; en revanche on peut obtenir facilement un terme de ce type par η-expansion, c'est à dire le terme `fun (A:Set) ⇒ eq A`.

Remarquons que, dans les premières versions de *Coq*, plusieurs constantes étaient liées à la description du prédicat d'égalité (voir page 126). Ces versions distinguaient entre autres les deux constantes ci-dessous :

```
eq : (A:Set)A→A→Prop
eqT : (A:Type)A→A→Prop
```

La discussion ci-dessus montre que la seconde constante a un usage plus général que la première ; pour cette raison, la version actuelle de *Coq* attribue à la constante `eq` le type `∀A:Type, A→A→Prop`, et `eqT` devient un synonyme de `eq`. Il est remarquable que cette évolution ne provoque aucun problème de compatibilité avec les développements effectués avec l'ancien typage. Ce type d'évolution concerne d'autres constructions similaires, telles la quantification existentielle (section 5.3.5) et la bibliothèque sur les relations binaires.

De façon générale, une construction polymorphe c de type `∀A:Type, B` aura un domaine d'application plus général que si on lui attribue le type `∀A:Set, B`. Si X est de sorte `Set`, " $c\ X$ " est bien typé dans les deux cas.

5.3 * Puissance d'expression du produit dépendant

Parmi les différents systèmes de types que l'on peut considérer, le Calcul des Constructions utilise les choix de typage les plus puissants possibles avant l'incohérence. Le lecteur intéressé pourra consulter les articles de T. Coquand et G. Huet [28, 29, 27].

Nous nous proposons d'explorer la puissance d'expression du Calcul des Constructions, en donnant la règle de formation du produit dépendant, et en montrant de nombreux exemples d'application de ses diverses facettes.

5.3.1 Règle de formation du produit dépendant

Puisque les types peuvent être obtenus en appliquant des fonctions à des arguments, les expressions de types doivent également être vérifiées pour assurer qu'elles respectent bien la discipline de typage. L'un des principes du calcul des constructions est que tout type est un terme ; par conséquent, ce type doit également être bien typé. De plus, la structure des règles de typage pour les types est directement reliée aux propriétés qui permettent d'assurer la cohérence du calcul [27].

Triplet (s, s', s'')	contraintes	nom populaire
(s, s', s')	$s, s' \in \{\texttt{Set}, \texttt{Prop}\}$	typage simple
$(\texttt{Type}(i), \texttt{Prop}, \texttt{Prop})$		imprédicativité de `Prop`
$(s, \texttt{Type}(i), \texttt{Type}(i))$	$s \in \{\texttt{Set}, \texttt{Prop}\}$	dépendance
$(\texttt{Type}(i), \texttt{Type}(j), \texttt{Type}(k))$	$(i \leq k,\ j \leq k)$	ordre supérieur

FIGURE 5.4 – Les triplets du Calcul des Constructions

Il faut distinguer le problème de construire un type produit bien formé du problème de construire un terme ayant ce type. Puisque les types représentent des formules logiques, il doit être possible d'exprimer des propositions fausses, même s'il doit naturellement être impossible de les prouver. Dans cette section, c'est la formation des types (ou du point de vue de la logique, la formation des formules) que nous considérons.

Les différentes possibilités de construction du produit dépendant se présentent traditionnellement sous la forme d'une seule règle de typage paramétrée par trois sortes : s, s' et s'' ; après avoir donné cette règle, nous préciserons quelles valeurs ces paramètres peuvent prendre, chaque cas étant illustré par quelques exemples.

La règle de formation du produit dépendant se présente ainsi :

$$\textbf{Prod}(s,s',s'') \quad \frac{E,\Gamma \vdash A : s \quad E,\Gamma :: (a : A) \vdash B : s'}{E,\Gamma \vdash \forall\, a{:}A,\, B \ :\ s''}$$

Nous rappelons en outre que, si le type B ne contient aucune occurrence de la variable a, alors le produit dépendant “ $\forall\, a{:}A,\, B$ ” se note simplement $A{\rightarrow}B$.

C'est le choix des triplets possibles pour (s, s', s'') qui détermine la puissance d'expression de la théorie des types. Le Calcul des Constructions est basé sur un choix différent de celui utilisé pour la théorie des types de Martin-Löf [69].

Nous pouvons maintenant nous intéresser aux triplets (s, s', s'') autorisés par le Calcul des Constructions. Chacun de ces triplets contribue à la puissance d'expression de *Coq*. Il sont présentés en figure 5.4. Chacune des lignes de ce tableau représente une facette de la diversité des types utilisables en *Coq*, que nous commentons à l'aide de nombreux exemples. Nous reprenons en fait la classification de systèmes de types due à H. Barendregt[8] connue sous le nom de « Cube de Barendregt ».

Nous reprendrons en les commentant les 4 lignes du tableau 5.4 ; il est cependant à remarquer que pratiquement tous les exemples intéressants utilisent plusieurs lignes de ce tableau, et la classification qui suit est forcément très imparfaite. Nous remarquons aussi que la première ligne ne comporte pas de nom ; les triplets qui lui sont associés permettent de reconstruire les systèmes de typage simple des chapitres 3 et 4. Il suffit pour ce faire de considérer les triplets $(\texttt{Set}, \texttt{Set}, \texttt{Set})$ (construisant les spécifications de fonctions) et $(\texttt{Prop}, \texttt{Prop}, \texttt{Prop})$ (autorisant la formation d'implications). Ces triplets, utilisés avec des types dépendants, autorisent de nouvelles constructions : quantification universelle, spécifications de fonctions à type dépendant, etc.

5.3.2 Types dépendants

La création de *types dépendants* est associée aux triplets de la 3-ième ligne du tableau 5.4, étiquetée « dépendance ».

Prédicats

Les types des prédicats (voir page 102) s'obtiennent en prenant $s = \texttt{Set}$ et $B = \texttt{Prop}$ dans la règle **Prod** de formation du produit dépendant. Le type d'un prédicat est lui même de sorte `Type`.

Par exemple, c'est ainsi que l'on obtient le type des prédicats unaires sur un type A : `Set`, que l'on note $A \rightarrow \texttt{Prop}$. Un prédicat exprimant la correction de programmes de tri sur `Z` pourra être du type `(list Z → list Z)→Prop`. Son application à une fonction de transformation de listes sur `Z` est bien une proposition exprimant sa correction.

En itérant la construction précédente, nous n'avons aucune difficulté à construire des types de prédicats à n places, de la forme $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow Prop$.

Quantification universelle

Réexaminons la première ligne du tableau 5.4. Le triplet $(\texttt{Set}, \texttt{Prop}, \texttt{Prop})$ se traduit en la règle de typage suivante, permettant de construire une proposition par quantification universelle sur le type A :

$$\textbf{Prod}(\texttt{Set},\texttt{Prop},\texttt{Prop}) \quad \frac{E, \Gamma \vdash A : \texttt{Set} \quad E, \Gamma :: (a : A) \vdash B : \texttt{Prop}}{E, \Gamma \vdash \forall \texttt{a}{:}A,\, B \ : \texttt{Prop}}$$

Par exemple, l'énoncé du théorème `le_i_SSi` (page 114) :

`∀i:nat, i ≤ S (S i)`

est bien construit par cette règle, car `nat` est de type `Set` et, lorsque `i` est de type `nat`, le terme " i ≤ S (S i) " est bien une proposition, dont la quantification universelle sur `i` est également de type `Prop`.

Exercice 5.3 On considère une section déclarant une spécification A, ainsi que trois prédicats sur A. Vérifier que les énoncés des trois lemmes ci-dessous sont bien de type `Prop`, puis construire des termes habitant ces types.

```
Section A_declared.
 Variables (A:Set)(P Q:A→Prop)(R:A→A→Prop).

 Theorem all_perm : (∀a b:A, R a b)→ ∀a b:A, R b a.
```

```
 Theorem all_imp_dist :
   (∀a:A, P a → Q a)→(∀a:A, P a)→∀a:A, Q a.

 Theorem all_delta : (∀a b:A, R a b)→∀a:A, R a a.

End A_declared.
```

Types de données dépendants

La section 5.1.1 montre l'intérêt de types de données dépendants de paramètres. C'est bien la règle **Prod**(`Set, Type`, `Type`) qui autorise la construction de types comme celui de `binary_word`, à savoir `nat→Set`.

```
Check (nat→Set).
```
nat→Set : Type

De même que pour la quantification universelle, la présence de types dépendant donne un nouvel intérêt au « typage simple » de la première ligne du tableau 5.4. Considérons par exemple, le type de la fonction de concaténation de mots binaires :

```
 ∀n p:nat, binary_word n → binary_word p → binary_word (n+p)
```

Une fois les trois types dépendants formés par application de `binary_word` à des termes de type `nat`, quatre applications successives de la règle **Prod**(`Set, Set`, `Set`) nous permettent de définir le type de la concaténation. Remarquons que, sur ces quatre constructions, deux se traduisent par une simple flèche. En effet le type du résultat d'une concaténation de mots booléens dépend de la taille de ces mots, et non de leur contenu.

Fonctions partielles

L'utilisation conjointe de types dépendants et du typage simple permet d'exprimer le type de fonctions partielles. Par exemple, le type d'une fonction de logarithme discret prenant des valeurs de type `nat` pourra être le suivant :

```
Check (∀n:nat, 0 < n → nat).
```
∀n:nat, 0 < n → nat : Set

Le type ainsi formé est habité par toute fonction associant à tout n : `nat` et à toute preuve π : `0 < n` un terme de type `nat`. Les techniques de construction de telles fonctions sont présentées en section 10.2.3.

5.3.3 Le polymorphisme

Le polymorphisme se traite différemment selon qu'il s'agit de construire les types polymorphes des langages de programmation ou des enoncés de règles d'inférence en logique.

Fonctions polymorphes

Nous avons déjà abordé le polymorphisme à la *ML* page 111. Un autre exemple est fourni par le type des couples que nous avons vu en page 108. Considérons par exemple le type de `pair`, le type de la fonctionnelle `iterate` étant laissé en exercice.

```
Check pair.
```
*pair : ∀ A B:Set, A → B → A * B*

```
Check (∀ A B:Set, A → B → A * B).
```
*∀ A B:Set, A → B → A * B : Type*

La dernière inférence de type contient les étapes suivantes :

$$\begin{array}{rcl} \dots & & \dots \\ \mathtt{A}:\mathtt{Set},\ \mathtt{B}:\mathtt{Set} & \vdash & \mathtt{A{\rightarrow}B{\rightarrow}A*B\ :\ Set} \\ \mathtt{A}:\mathtt{Set},\ \mathtt{B}:\mathtt{Set} & \vdash & \mathtt{A{\rightarrow}B{\rightarrow}A*B\ :\ Type} \\ \mathtt{A}:\mathtt{Set} & \vdash & \forall\mathtt{B{:}Set,A{\rightarrow}B{\rightarrow}A*B\ :\ Type} \\ & \vdash & \forall\ \mathtt{A\ B{:}Set,A{\rightarrow}B{\rightarrow}A*B\ :\ Type} \end{array}$$

Nous remarquons l'utilisation de la règle de conversion permettant de considérer que le type de `A→B→A*B` est `Type`, et autorisant deux utilisations successives du triplet (`Type`, `Type`, `Type`).

5.3.3.1 Polymorphisme et puissance d'expression

Le polymorphisme apporte bien plus que la possibilité de décrire des fonctions génériques. Reprenons par exemple la fonctionnelle `iterate`, déjà vue en section 5.2.1.1 page 111, dont nous rappelons le type :

```
iterate : ∀A:Set, (A→A)→nat→A→A
```

La fonctionnelle monomorphe “ `iterate nat` ” nous permet alors de redéfinir quelques fonctions primitives récursives usuelles :

```
Definition my_plus : nat→nat→nat := iterate nat S.

Definition my_mult (n p:nat) : nat :=
  iterate nat (my_plus n) p 0.

Definition my_expo (x n:nat) : nat :=
  iterate nat (my_mult x) n 1.
```

Le polymorphisme d'ordre supérieur de la fonctionnelle `iterate` nous donne une puissance d'expression considérable. En effet, nous pouvons définir par un *schéma récursif primitif* la fonction d'Ackermann, qui n'est pas *récursive primitive*. Pour mémoire, nous rappelons la définition usuelle de cette fonction, telle qu'on la trouve dans les cours de calculabilité.

$$\begin{array}{rcl} Ack(0,n) & = & n+1 \\ Ack(m+1,0) & = & Ack(m,1) \\ Ack(m+1,n+1) & = & Ack(m,Ack(m+1,n)) \end{array}$$

Nous pouvons la définir en *Coq* comme une application de la fonctionnelle `iterate`. On remarquera que cette définition utilise à la fois “ `iterate nat` ” et “ `iterate (nat→nat)` ”.

```
Definition ackermann (n:nat) : nat→nat :=
  iterate (nat→nat)
          (fun (f:nat→nat)(p:nat) ⇒ iterate nat f (S p) 1)
          n
          S.
```

Le schéma de *programmation récursive primitive* a été proposé dans la première moitié du vingtième siècle pour donner une description formelle des définitions de fonctions récursives. Ce schéma permet de définir des fonctions récursives totales, mais Ackermann [2] a démontré que son pouvoir expressif était trop limité. Ackermann a décrit une fonction qui était mathématiquement bien définie mais ne pouvait pas être décrite comme une fonction récursive primitive.

Sans entrer dans les détails, les fonctions primitives récursives sont les fonctions qui peuvent être définies à l'aide de la fonction « successeur » et les itérations sur les types non-fonctionnels (les types de premier-ordre).

Exercice 5.4 Pour chacune des spécifications suivantes, vérifier que le type associé est bien de sorte `Type` et donner une fonction la réalisant :

1. `id: ∀A:Set, A→A`
2. `diag: ∀A B:Set, (A→A→B)→A→B`
3. `permute: ∀A B C:Set, (A→B→C)→B→A→C`
4. `f_nat_Z : ∀A:Set, (nat→A)→Z→A`

Remarque 5.5 Dans les versions précédentes de *Coq*, un type polymorphe comme “ `∀A:Set, A→A` ” avait pour sorte `Set`, et non `Type`, ceci étant dû à la présence d'un triplet supplémentaire (`Type`, `Set`, `Set`). On dit alors que dans ces versions de *Coq*, `Set` était « imprédicatif », c'est à dire que l'on pouvait définir un type A de sorte `Set` par un produit sur tous les éléments de sorte `Set`, y compris A lui-même. Il est remarquable que ce changement — appelé « abandon de l'imprédicativité de `Set` » — ne rend caduc aucun développement construit dans les version antérieures du système *Coq*. On dit alors que la sorte `Set` est devenue *prédicative*.

Logique minimale propositionnelle polymorphe

À la différence de `Set`, la sorte `Prop` est imprédicative, c'est à dire que nous pouvons construire des propositions par quantification universelle sur toutes les propositions. Techniquement, cette faculté est dûe à la présence du triplet (`Type`, `Prop`, `Prop`)) (polymorphisme). En voici un exemple simple :

Dans le chapitre 4, nous avons considéré la proposition `P→P`, c'est à dire un terme de type `Prop` dans le contexte `[P:Prop]` ; en utilisant la règle **Prod**(`Type`,`Prop`, `Prop`), nous pouvons déduire que le produit dépendant " `∀P:Prop, P→P` " est bien une proposition, cette fois dans le contexte vide.

D'autre part, la règle d'abstraction **Lam** de la section 5.2.2 (appliquée deux fois) nous permet de construire facilement une preuve de cette dernière proposition :

```
Check (∀P:Prop, P→P).
```
∀ P:Prop, P→P : Prop

```
Check (fun (P:Prop)(p:P) ⇒ p).
```
fun (P:Prop)(p:P) ⇒ p : ∀ P:Prop, P→P

5.3.3.2 Logique minimale polymorphe

La démarche suivie pour la logique propositionnelle peut très bien s'étendre aux prédicats. En effet, si A est de sorte `Type`, alors A→`Prop` est de sorte `Type`, ce qui nous autorise à former des propositions de la forme " $\forall P : A \to$`Prop`, Q ", où Q est une proposition dans le contexte où P est déclaré. De plus, le type A peut faire l'objet d'une quantification universelle, en appliquant le polymorphisme associé aux triplets (`Type(`i`)`, `Prop`, `Prop`).

Cette construction s'étend bien sûr aux prédicats à plusieurs arguments.

Exercice 5.5 Pour chacun des théorèmes suivants, vérifier que l'énoncé est bien une proposition puis construire un terme habitant ce type.

```
Theorem all_perm :
 ∀(A:Type)(P:A→A→Prop), (∀x y:A, P x y)→∀x y:A, P y x.
```

```
Theorem resolution :
 ∀(A:Type)(P Q R S:A→Prop), (∀a:A, Q a → R a → S a)→
   (∀b:A, P b → Q b)→(∀c:A, P c → R c → S c).
```

L'élimination du faux

Un intérêt du polymorphisme en logique est l'expression de véritables règles d'inférence. Parmi celles-ci, les règles permettant le raisonnement par l'absurde forment un exemple très caractéristique. Par exemple la proposition fausse est représentée en *Coq* par la constante `False : Prop` (voir section 9.2.2). La vraie nature de cette proposition s'exprime à l'aide de trois constantes dont les deux premières ont un type construit à l'aide de polymorphisme.

```
False_ind : ∀P:Prop, False→P
False_rec : ∀P:Set, False→P
False_rect : ∀C:Type, False→C
```

L'énoncé de `False_ind` indique clairement son emploi : si P est une proposition quelconque et t une preuve de `False`, alors le terme " `False_ind` P t " est une preuve de P. En termes logiques, on dit que `False_ind` permet d'implanter la *règle d'élimination du faux* :

$$\mathbf{False}_e \quad \frac{E,\Gamma \vdash t : \texttt{False}}{E,\Gamma \vdash \texttt{False\_ind}\ P\ t : P}$$

L'élimination de `False` se fait en général dans un contexte non vide. En effet, supposons que nous puissions obtenir un jugement de la forme suivante :

$$E,\Gamma \vdash t : \texttt{False}$$

Dans le même environnement, nous aurions une preuve de n'importe quel énoncé, même par exemple, la proposition `22=28` : [4]

$$E,[] \vdash \texttt{False\_ind (22=28)}\ t\ \texttt{: 22 = 28}$$

5.3.4 L'égalité en *Coq*

La représentation de l'égalité en *Coq* est une autre application du polymorphisme ; étant donnée son importance en raisonnement, nous y consacrons quelques pages.

L'égalité est définie en *Coq* de façon inductive (voir 9.2.6). Nous nous contentons ici de souligner les relations entre cette notion et le produit dépendant.

Règle de typage

La constante `eq` a pour type " `∀A:Type, A→A→Prop` " ; en d'autres termes, si t_1 et t_2 sont deux expressions de type A, alors l'application " `eq (A:=`A`)` t_1 t_2 " est une proposition, laquelle s'abrège en "$t_1 = t_2$".

Le type de `eq` impose que l'on ne peut consider que des égalités entre objets de même type : même si nous voulons exprimer que deux objets sont différents, il est nécessaire que ces deux termes soient de même type :

```
Check (~true=1).
```
Error: The term 1 has type nat while it is expected to have type bool

4. Nous n'utilisons ici aucune propriété particulière de l'égalité ; seul nous importe le fait que " `22=28` " soit une proposition.

Pour l'usage général, cette limitation du prédicat d'égalité n'est pas significative car il est rarement nécessaire de considérer des égalités entre objets de type différent. Néanmoins, d'autres codages de l'égalité sont disponibles pour les cas exceptionnels (voir section 9.2.7).

Règle d'introduction de l'égalité

La constante `refl_equal`, exprime de façon polymorphe le caractère réflexif de l'égalité ; son type est le suivant :

refl_ equal : ∀ (A:Type)(x : A), x=x

```
Theorem ThirtySix : 9*4=6*6.
Proof (refl_equal 36).
```

Règle d'élimination de l'égalité

De même que pour `False`, trois théorèmes permettent d'utiliser (on dit également *élimininer*) une égalité dans la construction d'un terme pour un type donné. Ces théorèmes diffèrent seulement par la sorte de ce type. Le premier de ces théorèmes est très proche de la définition que Leibniz a donné de l'égalité, reprise page 160.

- `eq_ind :`
  `∀(A:Type) (x:A) (P:A→Prop), P x → ∀y:A, x = y → P y`
- `eq_rec :`
  `∀(A:Type) (x:A) (P:A→Set), P x → ∀y:A, x = y → P y`
- `eq_rect :`
  `∀(A:Type) (x:A) (P:A→Type), P x → ∀y:A, x = y → P y`

Comme exemple de démonstration utilisant l'un de ces théorèmes, nous considérons la démonstration suivante qui montre que l'égalité est symétrique [5] (ceci reproduit un théorème déjà présent dans les bibliothèques de *Coq*) :

```
Definition eq_sym  (A:Type)(x y:A)(h : x=y) : y=x :=
 eq_ind  x (fun z ⇒ z=x) (refl_equal x) y h.

Check (eq_sym _ _ _ ThirtySix).
```
*eq_ sym nat (9*4) (6*6) ThirtySix : 6*6 = 9*4*

Nous verrons en section 9.2.6 que ces règles d'élimination jouent un rôle clef dans les tactiques de réécriture.

5. Le théorème `eq_ind`, comme la plupart des constantes définies dans le noyau logique de *Coq*, est défini sous le mode des arguments implicites. Si l'on avait voulu expliciter tous les arguments de `eq_ind`, le corps de la fonction aurait été " `eq_ind (A:= A) (x:= x) (fun z => z = x) (refl_equal x) (y:= y) h` ".

5.3.5 Types d'ordre supérieur

Il nous reste une dernière famille de produits dépendants à étudier : ceux correspondant aux triplets de sortes de la forme :

$$s = \mathbf{Type}(i), \quad s' = \mathbf{Type}(j), \quad s'' = \mathbf{Type}(k) \quad (i \leq k,\ j \leq k)$$

Cette famille de règles de typage rend possible la construction de types à partir d'autres types, en d'autres termes, elles permettent de typer des *constructeurs de types*. Ces constructeurs se retrouvent à la fois en programmation : la constante `list` prend en argument un type de donnée A et construit le type des listes d'éléments de type A ; de même pour la constante `prod`, qui à partir de deux types A et B, construit le produit cartésien de A par B. En logique, nous pouvons donner un type aux *connecteurs*, lesquels permettent de construire de nouvelles propositions à partir de propositions plus simples.

Connecteurs propositionnels

Les connecteurs logiques : négation, conjonction, disjonction, ..., permettent chacun de construire une nouvelle proposition à partir d'une ou deux propositions. On peut donc considérer la négation comme une fonction unaire de `Prop` dans `Prop`, la disjonction et la conjonction comme des fonctions binaires sur `Prop`.

Or la règle de construction de types d'ordre supérieur (avec $i = j = k = 0$) nous permet de construire les types `Prop→Prop` et `Prop→Prop→Prop`, tous deux de sorte `Type`.

La négation

La négation est définie en *Coq* à partir de `False` (voir page 125) par une fonction `not` de type `Prop→Prop`, c'est à dire comme une opération unaire sur les propositions.

```
Definition not (P:Prop) : Prop := P→False.
```

La syntaxe $\sim P$ permet d'abréger la notation préfixe " `not` P ".

Conjonction et disjonction

Nous verrons en 9.2.3 et 9.2.4 comment se définissent la conjonction et la disjonction ; nous nous intéressons ici à ces deux connecteurs sous le point de vue des règles de typage.

La conjonction et la disjonction sont respectivement représentées par les constantes `and` et `or` de type `Prop→Prop→Prop`. Les écritures P`/\`Q et P`\/`Q peuvent remplacer respectivement " `and` P Q " et " `or` P Q ", les opérateurs `/\` et `\/` ayant les mêmes conventions d'association à droite que la flèche →. De même que pour la flèche →, nous utiliserons fréquemment les symboles $\wedge$ et $\vee$ en lieu et place des assemblages de caractères `/\` et `\/`. Certaines autres notations de *Coq* contiennent une conjonction implicite. Par exemple, une double inégalité de la forme " `x ≤ y ≤ z` " n'est qu'une abréviation de la conjonction " `x ≤ y ∧ y ≤ z` ".

Règles d'introduction

Les théorèmes suivants permettent de prouver des propositions dont la conclusion est une conjonction ou une disjonction ; dans le jargon logique, ces théorèmes s'appellent des « règles d'introduction ». Il sont bien typés grâce aux deux premières lignes du tableau 5.4 page 120 :

```
Check conj.
```
conj : ∀ A B:Prop, A→B→A∧B

```
Check or_introl.
```
or_introl : ∀ A B:Prop, A→A∨B

```
Check or_intror.
```
or_intror : ∀ A B:Prop, B→A∨B

Les règles duales, appelées règles d'élimination, sont également fournies dans le système *Coq* car leur typage ne pose aucun problème. Par exemple, la règle d'élimination de la conjonction a la forme suivante :

```
Check and_ind.
```
and_ind : ∀ A B P:Prop, (A→B→P)→A∧B→P

Pour illustrer ces règles, nous montrons les preuves de deux propositions simples :

```
Theorem conj3 : ∀P Q R:Prop, P→Q→R→P∧Q∧R.
Proof (fun P Q R p q r ⇒ conj p (conj q r)).

Theorem disj4_3 : ∀P Q R S:Prop, R → P∨Q∨R∨S.
Proof
 (fun P Q R S r ⇒ or_intror _ (or_intror _ (or_introl _ r))).
```

Un exemple simple d'utilisation de la règle d'élimination de la conjonction est un théorème qui exprime que l'on peut déduire d'une conjonction le premier terme de cette conjonction :

```
Definition proj1' :  ∀A B:Prop, A∧B→A :=
 fun (A B:Prop)(H:A∧B) ⇒ and_ind (fun (H0:A)(_:B) ⇒ H0) H.
```

Le quantificateur existentiel

Nous avons vu en 5.2.1.1 que la quantification universelle « $\forall x : A\ P(x)$ » s'exprime directement à l'aide du produit dépendant . Qu'en est-il de la quantification existentielle « $\exists x : A.\ P(x)$ » ? *Coq* la définit au moyen d'une constante `ex` de type " `∀A:Type, (A→Prop)→Prop` ".

La définition de cette constante fait appel aux constructions inductives et sera commentée en 9.2.5. Le type de `ex` est de sorte `Type`, et sa construction est autorisée par la suite de jugements ci-dessous, dont seuls le troisième et le quatrième utilisent la construction de types d'ordre supérieur :

$$\begin{aligned}
[\texttt{A:Type}] \vdash \texttt{Prop} &: \texttt{Type}\\
[\texttt{A:Type}] \vdash \texttt{A}{\rightarrow}\texttt{Prop} &: \texttt{Type}\\
[\texttt{A:Type}] \vdash \texttt{(A}{\rightarrow}\texttt{Prop)}{\rightarrow}\texttt{Prop} &: \texttt{Type}\\
\vdash \forall\,\texttt{A:Type,(A}{\rightarrow}\texttt{Prop)}{\rightarrow}\texttt{Prop} &: \texttt{Type}
\end{aligned}$$

Soit alors $P : A{\rightarrow}\texttt{Prop}$ un prédicat ; le type de `ex` nous permet de traduire la notation usuelle $\exists x : A\ P(x)$ en l'application “ `ex` P ” [6].

Coq fournit la notation “ `exists x:`A`,` t ” pour “ `ex (fun x:`$A{\Rightarrow}t$`)` ”. Nous utiliserons également le symbole '∃' en lieu et place du symbole '`exists`'.

```
Check (ex (fun z:Z ⇒ (z*z ≤ 37 ∧ 37 < (z+1)*(z+1))%Z)).
```
*∃ z:Z, (z*z ≤ 37 < (z+1)*(z+1))%Z*
: Prop

La règle d'introduction et la règle d'élimination pour ce connecteur logique sont les constantes `ex_intro` et `ex_ind` dont voici les types.

```
Check ex_intro.
```
ex_intro : ∀ (A:Type)(P:A→Prop)(x:A), P x → ex P
```
Check ex_ind.
```
ex_ind :
∀ (A:Type)(P:A→Prop)(P0:Prop), (∀ x:A, P x → P0)→ ex P → P0

Types de données polymorphes

Le typage d'ordre supérieur nous autorise à construire des types comme `Set→Set`, `Set→Set→Set`, etc. Le premier type est celui de `list`, le second celui de `prod` (voir page 106).

Le module `List` définit entre autres les constantes polymorphes suivantes :
- `nil` : `∀A:Set,list A` (liste vide)
- `cons` : `∀A:Set,A → list A → list A` (ajout en tête de liste)
- `app` : `∀A:Set,A → list A → list A` (concaténation de listes)

Le tableau ci-dessous contient quelques jugements de typage ; notons que les définitions de `List` ont été écrites sous le mode d'arguments implicites, ce qui évite une trop grande lourdeur dans l'écriture des termes. Dans ce tableau les termes sont donnés sous leur forme externe (acceptée par *Coq* et en utilisant la convention des arguments implicites), et non sous leur forme complète ; notons

6. Le premier argument de `ex` est implicite

également que le module `List` doit avoir été chargé (par `Require`).

```
                        list  :  Set→Set
                    list nat  :  Set
               list (Z →nat)  :  Set
               nil (A:=nat)   :  list nat
           cons (A:=Z→nat)    :  (Z→nat)→
                                 list (Z→nat)→
                                 list (Z→nat)
                    Zabs_nat  :  Z→nat
           cons Zabs_nat nil  :  list (Z→nat)
cons (cons (-273)%Z nil) nil) :  list (list Z)
```

On remarquera que l'unique argument de `nil` est implicite, et n'a donc pas besoin d'être spécifié dès qu'il peut être inféré à partir du contexte.

Remarquons que le type de la constante `list` interdit son utilisation pour construire des listes de *propositions*, ainsi qu'en témoigne le court dialogue suivant :

```
Check (cons (3≤6)%Z nil).
```
Error: Illegal application (Type Error):
The term "@nil" of type "∀ A:Set, list A"
cannot be applied to the term
"Prop" : "Type"
This term has type "Type" which should be coercible to "Set"

```
Check (list Prop).
```
Error: The term "Prop" has type "Type" while it is expected to have type
"Set"

De même, le typage de `cons` interdit les listes de type hétérogène, ainsi qu'en témoigne l'échange suivant, mais ceci est cohérent avec la pratique habituelle des langages de programmation fonctionnels typés.

```
Check (cons  655 (cons (-273)%Z nil)).
```
Error: The term "cons (-273)%Z nil" has type "list Z"
while it is expected to have type "list nat"

Afin de bien comprendre le message d'erreur, on peut considérer que ce dernier terme aurait la forme suivante si l'on fournissait tous ses arguments implicites :

```
cons (A:=nat) 655 (cons (A:=Z)(-273)%Z (nil (A:=Z))).
```

On trouvera en section 7.4.1 quelques exemples de programmation et preuves sur les listes.

Chapitre 6

Logique de tous les jours

Le chapitre précédent montre que le système de types du Calcul des Constructions est assez puissant pour permettre la représentation des formules logiques et des théorèmes de la logique usuelle. L'interprétation des types comme des formules et des termes comme des preuves fonctionne donc bien et permettra bien de ramener la vérification de développements formels à la vérification de types de certains termes. Ce chapitre prend un point de vue plus pragmatique et montre comment les tactiques permettent la construction des termes de preuve et facilitent la formalisation du raisonnement logique.

6.1 Pratique du produit dépendant

Nous allons nous familiariser avec le produit dépendant en nous basant sur plusieurs exemples-types d'utilisation. Ce sera l'occasion de montrer comment les tactiques de *Coq* opèrent sur cette structure.

Comme dans le chapitre précédent, nous supposons ouverte la portée `nat_scope`, et chargées les bibliothèques `Arith` et `ZArith`.

6.1.1 `exact` et `assumption`

Les tactiques `exact` et `assumption` ont déjà été présentées en 4.2.2 ; leur comportement doit être précisé en relation avec la règle de conversion.

La tactique “ `exact` t ” réussit lorsque le type du terme t et l'énoncé du sous-but courant sont *convertibles* (et non pas seulement identiques). De même `assumption` réussit si le sous-but courant est convertible avec le type d'une hypothèse du contexte courant.

Exemple

Considérons à nouveau le prédicat `le:nat→nat→Prop` associé à la relation d'ordre $\leq$ sur $\mathbb{N}$; l'ordre strict $<$ sur $\mathbb{N}$ est défini par « $n < p$ si $n+1 \leq p$ », c'est à dire :

```
Definition lt (n p:nat) : Prop := S n ≤ p.
```

Considérons alors le but suivant :

```
Theorem conv_example : ∀n:nat, 7*5 < n → 6*6 ≤ n.
 intros.
  ...
  n : nat
  H : 7*5 < n
  ============================
  6*6 ≤ n

 assumption.
Qed.
```

L'activation de la tactique `assumption` (ou bien “ `exact H` ”) contrôle la convertibilité entre l'hypothèse `H` et l'énoncé à prouver, ce qui comporte les réductions suivantes :

- une δ-réduction sur `lt`,
- deux β-réductions aboutissant au type “ `S (7 * 5) ≤ n` ” pour l'hypothèse `H`,
- deux suites de δ, ι et β réductions aboutissant chacune à la proposition “ `36 ≤ n` ”.

Le succès de la tactique activée est alors assuré.

6.1.2 La tactique `intro`

L'extension de la tactique `intro` et de ses variantes ne pose pas de problèmes particuliers. Appliquée à un but $\Gamma \overset{?}{\vdash} (\forall\, x\!:\!A,\, B)$, la tactique “ `intro` x ” produit le sous-but $\Gamma :: (x : A) \overset{?}{\vdash} B$ si l'ajout au contexte d'une déclaration portant sur x ne pose pas de problème de nom. Dans le cas contraire, le système *Coq* produit un message d'erreur.

```
Theorem bad_example_for_intro : ∀n m:nat, n ≤ n → m ≤ m.
Proof.
 intro x.
  ...
  x : nat
  ============================
   ∀ m:nat, x ≤ x → m ≤ m.

 intro x.
Error: x is already used
```

L'utilisateur peut également proposer un nouveau nom, sous la forme “ `intro` y ”, par exemple. Le nouveau but est alors B{x/y} pour tenir compte de ce renommage. Ces considérations s'étendent bien entendu aux variantes plurielles. Pour les variantes anonymes, le système *Coq* utilise naturellement de nouveaux noms.

Le début de preuve suivant montre une application simple d'`intros` : le contexte courant s'enrichit des déclarations de `n` et `H`.

```
Lemma L_35_36 : ∀n:nat, 7*5 < n → 6*6 ≤ n.
Proof.
 intro n.
  ...
  n : nat
  ============================
   7*5 < n → 6*6 ≤ n
 intro H; assumption.
Qed.
```

On remarquera la simplicité du terme de preuve obtenu ; la règle de conversion permet de masquer tous les calculs arithmétiques liés à la multiplication :

```
Print L_35_36.
 L_35_36 =
fun (n:nat)(H: 7*5 < n) ⇒ H
   : ∀ n:nat, 7*5 < n → 6*6 ≤ n

Argument scopes are [nat_scope _]
```

Logique minimale propositionnelle polymorphe

La tactique `intro` peut introduire dans le contexte des variables représentant des propositions, correspondant à l'utilisation du produit dépendant pour représenter la quantification universelle sur des propositions. Cette extension de la tactique `intro` permet donc de démontrer les théorèmes de la logique minimale propositionnelle polymorphe. Par exemple, nous pouvons considérer une version polymorphe de la « transitivité de l'implication ».

```
Theorem imp_trans : ∀P Q R:Prop, (P→Q)→(Q→R)→P→R.
Proof.
  intros P Q R H H0 p.
  apply H0; apply H; assumption.
Qed.

Print imp_trans.
imp_trans =
fun (P Q R:Prop)(H:P→Q)(H0:Q→R)(p:P) ⇒ H0 (H p)
     : ∀ P Q R:Prop, (P→Q)→(Q→R)→ P → R
Argument scopes are [type_scope type_scope type_scope _ _ _]
```

L'intérêt de cette version polymorphe est d'obtenir une règle applicable sur n'importe quelle proposition :

```
Check (imp_trans _ _ _ (le_S 0 1)(le_S 0 2)).
```
imp_trans $(0 \leq 1)(0 \leq 2)(0 \leq 3)$*(le_S 0 1)(le_S 0 2)*
  $: 0 \leq 1 \rightarrow 0 \leq 3$

Exercice 6.1 Reprendre l'exercice 4.2, page 85, en considérant non plus des énoncés contenant les variables prédéclarées `P`, `Q`, `R`, etc., mais des propositions « closes » formées par produit dépendant sur ces variables.

La tactique `intro` et la convertibilité

On trouvera dans le manuel de référence une description précise des utilisations de la tactique `intro` et de ses variantes. En particulier, si le but courant n'est pas un produit, mais réductible en un produit, la tactique `intro` peut procéder aux réductions nécessaires.

Considérons par exemple le début de preuve suivant :

```
Definition neutral_left (A:Set)(op:A→A→A)(e:A) : Prop :=
  ∀x:A, op e x = x.

Theorem one_neutral_left : neutral_left Z Zmult 1%Z.
```
1 subgoal

```
  ============================
   neutral_left Z Zmult 1%Z
```

```
Proof.
 intro z.
```
1 subgoal

```
  z : Z
  ============================
   (1*z)%Z = z
```

On constate que l'appel à “ `intro z` ” provoque la δ-réduction de la constante `neutral_left`, puis une suite de β-conversions ; le but courant devient alors un produit. La suite de la preuve fait appel aux techniques d'automatisation présentées dans le chapitre 8 : un appel à “ `auto with zarith` ” et tout est terminé.

6.1.3 La tactique `apply`

Nous avons vu en section 4.2.2 que la tactique `apply` fait intervenir les types de tête possibles d'un terme fonctionnel. En présence de fonctions à type dépendant cette notion doit être revue.

Types de tête et type final

Nous adaptons légèrement les notions présentées page 80 pour tenir compte de la construction du produit dépendant.

Soit un terme t de type

$\forall$ (v_1:A_1)...(v_n:A_n), B

et k un entier compris entre 1 et $n+1$; le type de tête de rang k de t sera le produit " $\forall$ (v_k:A_k)...(v_n:A_n), B ", et B est le type final de t si B n'est pas lui même un produit.

Par la suite nous appellerons « type de tête » un type de tête de rang k ou un type final (le type final peut être assimilé au type de tête de rang $n+1$).

Pour illustrer ces concepts, prenons des exemples sur l'ordre $\leq$ sur $\mathbb{N}$. Nous utiliserons quelques théorèmes de la bibliothèque `Arith`.

```
le_n :  ∀n:nat, n ≤ n
le_S : ∀n m:nat, n ≤ m → n ≤ S m
le_trans  : ∀n m p:nat, n ≤ m → m ≤ p → n ≤ p
```

Prenons par exemple le théorème `le_trans` ; le tableau ci-dessous en donne les types de tête de rang 1 à 5, ainsi que le type final :

rang	
1	`∀n m p:nat, n ≤ m → m ≤ p → n ≤ p`
2	`∀m p:nat, n ≤ m → m ≤ p → n ≤ p`
3	`∀p:nat, n ≤ m → m ≤ p → n ≤ p`
4	`n ≤ m → m ≤ p → n ≤ p`
5	`m ≤ p → n ≤ p`
type final	`n ≤ p`

On remarquera que les types de tête de rang supérieur à 1 et le type final comportent des variables libres prises dans $\{\texttt{n}, \texttt{m}, \texttt{p}\}$; il faut en fait plus les considérer comme l'expression de familles entières de types obtenus en substituant des termes de type approprié. Par exemple, le type " `33 ≤ 63 → 32 ≤ 63` " est une instance du type de tête de rang 5.

De façon générale, si une variable v_i apparaît dans un type de tête de rang supérieur à i ou dans le type final, on dira que v_i est une *variable dépendante*. Si v_i est une variable non-dépendante, le produit " $\forall v_i$:A_i, C " est simplement noté $A_i \rightarrow C$.

Les cas les plus simples

Considérons par exemple le but " `33 ≤ 34` " ; ce but est une instance du type de tête de rang 3 de `le_S`, obtenue avec la substitution $\sigma = \{\texttt{n}/\texttt{33}; \texttt{m}/\texttt{33}\}$. Appliquer la tactique " `apply le_S` " au but courant revient donc à chercher un terme de type de la forme " `le_S 33 33` π ", où π est un terme de preuve de " `n ≤ m` "$\{\texttt{n}/\texttt{33}; \texttt{m}/\texttt{33}\}$, c'est à dire " `33 ≤ 33` ". Ce terme π ne peut être déterminé par confrontation du but courant avec le type de σ(`le_S`) ; sa construction devient donc l'objet d'un nouveau but.

Ce type de situation — ou l'on emploie la tactique " `apply` t " et les variables dépendantes de t permettent de déterminer une substitution σ faisant coïncider le but courant avec une instance d'un type de tête de t, et σ est définie sur toutes les variables dépendantes de t — est le plus simple pour utiliser `apply`. Les buts engendrés correspondent aux instances par σ des types des variables non dépendantes, c'est à dire des prémisses de t.

Nous illustrons ce cas simple par quelques exemples. Considérons en premier lieu le théorème

$$\forall i \in \mathbb{N}.\ i \leq (i+2)$$

Un terme de preuve en a été donné page 114, et illustré par les figures 5.2 et 5.3. La preuve interactive suivante construit exactement ce terme de preuve, à l'aide de tactiques faciles à employer :

```
Theorem le_i_SSi : ∀i:nat, i ≤ S (S i).
Proof.
 intro i.
 ...
 i : nat
 ============================
  i ≤ S (S i)

 apply le_S.
 ...
 ============================
  i ≤ S i

 apply le_S.
 ...
 ============================
  i ≤ i

 apply le_n.
Qed.
```

Logique minimale polymorphe

De manière générale, le polymorphisme nous permet aussi de quantifier sur des prédicats et l'extension de la tactique `apply` permet d'instancier les arguments lors de l'application d'hypothèses portant sur ces prédicats.

Voici par exemple une preuve de la distributivité de la quantification universelle sur l'implication :

```
Theorem all_imp_dist   :
 ∀(A:Type)(P Q:A→Prop), (∀x:A, P x → Q x)→(∀y:A, P y)→
 ∀z:A, Q z.
```

```
Proof.
 intros A P Q H H0 z.
 apply H; apply H0; assumption.
Qed.
```

On remarquera une fois de plus la cohérence interne de Coq *: la tactique* `intros` *reçoit six arguments dont l'un est un type A, deux sont des prédicats, deux sont des propositions et le dernier une donnée (de type A) ; tous ces arguments sont traités de la même façon.*

Exercice 6.2 En utilisant les tactiques, refaire les démonstrations de l'exercice 5.5, page 125.

Comment aider `apply`

Tous les cas d'utilisation de la tactique " `apply` t " ne sont pas aussi favorables que ceux décrits ci-dessus. Il peut arriver que la confrontation du but courant et d'un type de tête de t détermine une partie seulement des variables dépendantes.

À titre d'exemple, considérons les trois théorèmes suivants ; les deux premiers font partie la bibliothèque `Arith`, et le troisième est laissé en exercice (à faire après avoir lu jusqu'à la page 158).

le_trans : $\forall$ *n m p:nat*, $n \leq m \rightarrow m \leq p \rightarrow n \leq p$

mult_le_compat_l : $\forall$ *n m p:nat*, $n \leq m \rightarrow p*n \leq p*m$

mult_le_r : $\forall$ *m n p:nat*, $n \leq p \rightarrow n*m \leq p*m$

Commençons à prouver un résultat reliant la multiplication et l'ordre $\leq$.

```
Theorem le_mult_mult :
   ∀a b c d:nat, a ≤ c → b ≤ d → a*b ≤ c*d.
Proof.
 intros a b c d H H0.
  ...
  H :  a ≤ c
  H0 : b ≤ d
  ============================
   a*b ≤  c*d
```

Une bonne approche est d'appliquer la transitivité de $\leq$, mais rien ne permet d'instancier automatiquement la variable `m` de l'énoncé de `le_trans`.

```
apply le_trans.
  ...
Error: generated subgoal a*b ≤ ?META30 has metavariables in it
```

En effet, seule la substitution $\sigma = \{$n/ $a * b$; p/ $c * d\}$ est calculée à partir du but courant et du type de tête de le_trans. Aucune substitution pour la variable dépendante m n'est inferée, ce qui empêche la génération de sous-buts associés aux prémisses " n ≤ m " et " m ≤ p ".

Une solution possible est d'associer explicitement le terme " c * b " à la variable m pour compléter la substitution σ :

```
 apply le_trans with (m := c*b).
 apply mult_le_r; assumption.
 apply mult_le_compat_l; assumption.
Qed.
```

En résumé, la tactique " apply t with (v_{i_1} := t_1)...(v_{i_k}:=t_k) " peut s'utiliser dès que les variables dépendantes $v_{i_1} \dots v_{i_k}$ ne peuvent être déterminées par comparaison du but avec un type de tête de t.

Une variante : la tactique eapply

Le système *Coq* propose une variante de la tactique apply permettant d'éviter de fournir trop tôt des substitutions explicites (arguments suivant with). Si t est un terme, la tactique " eapply t " se comporte comme " apply t ", mais n'échoue pas si elle n'arrive pas à déduire des instanciations pour certaines variables des hypothèses de t. Cet échec est évité en remplaçant ces variables par des « variables existentielles », qui restent à déterminer dans le reste de la preuve. Ces variables sont identifiées par un nom de la forme ?n, où n est un entier naturel. Voici notre exemple précédent, utilisant cette fois *eapply*.

```
Theorem le_mult_mult' :
 ∀a b c d:nat, a ≤ c → b ≤ d → a*b ≤ c*d.
Proof.
 intros a b c d H H0.
 ...
 H :  a ≤ c
 H0 : b ≤ d
 ============================
  a*b ≤  c*d

eapply le_trans.

 ...
 H :  a ≤ c
 H0 : b ≤ d
 ============================
  a*b ≤ ?2
subgoal 2 is:
  ?2 ≤ c*d
```

En pratique, la tactique " `eapply le_trans` " applique le théorème `le_trans` sans connaître le terme qui devra être lié à la variable `m`; ce terme est provisoirement représenté par une variable existentielle, qui doit être instanciée ultérieurement (c'est à dire dans la suite de la preuve).

Les deux buts engendrés partagent donc la variable existentielle `?2` et sont respectivement " `a * b ≤ ?2` " et " `?2 ≤ c * d` ".

Le premier sous-but peut s'attaquer en appliquant le théorème `mult_le_compat_l`, ce qui crée une nouvelle variable existentielle.

```
eapply mult_le_compat_l.
2 subgoals
  ...
  H0 : b ≤ d
  ============================
   b ≤ ?4
subgoal 2 is:
  a*?4 ≤ c*d
```

L'instanciation de `?4` en `d` se fait par unification (confrontation) avec l'énoncé de l'hypothèse `H0`. Ceci se fait par la tactique " `eexact H0` ". On remarquera que cette instanciation se propage dans le second sous-but, qui ne contient plus aucune variable existentielle.

```
 eexact H0.
1 subgoal
  ...
  H : a ≤ c
  H0 : b ≤ d
  ============================
   a*d ≤ c*d
 apply mult_le_r.
 assumption.
Qed.
```

`apply` et la conversion

La comparaison entre le but courant et les types de tête de t se fait en tenant compte de la convertibilité. Dans l'exemple suivant les multiplications vont se simplifier en `0`, afin de pouvoir appliquer `le_n` (voir page 59).

```
Theorem le_0_mult : ∀n p:nat, 0*n ≤ 0*p.
Proof.
 intros n p; apply le_n.
Qed.
```

En revanche, un terme de la forme " `n * 0` " n'est pas réductible en `0` (Voir l'exercice 7.13, page 194).

```
Theorem le_O_mult_R : ∀n p:nat, n*0 ≤ p*0.
Proof.
 intros n p; apply le_n.
  ...
> intros n p; apply le_n.
>              ^^^^^^^^^^
Error: Impossible to unify p*0 with n*0
Abort.
Current goal aborted
```

Pour prouver ce lemme, une solution sera d'exploiter la commutativité de la multiplication avant d'appliquer `le_n` (voir la tactique `rewrite`, page 155.)

Lors d'un appel à la tactique “ `apply` t ”, le type de t et le but courant peuvent faire l'objet de réductions avant d'être confrontés.

En revanche, `apply` peut échouer parce que cette tactique ne procède pas à une δ-réduction sur le symbole de tête du but courant. Considérons par exemple le but suivant :

```
Lemma lt_8_9 : 8 < 9.
Proof.
```

La tactique “ `apply le_n` ” échoue faute de δ-réduction sur `lt`.

```
 apply le_n.
Error: Impossible to unify ?META86 ≤ ?META86 with 8 < 9
```

Pour résoudre ce but, il suffit de développer `lt` en `le` (voir page 143).

```
 unfold lt; apply le_n.
Qed.
```

De manière plus générale, “ `apply` t ” utilise l'unification d'ordre supérieur pour confronter le but courant avec les types de tête de t. Or cette unification est indécidable, ce qui explique que cette tactique peut échouer sur des configurations paraissant faciles à l'utilisateur. Dans de tels cas, on peut utiliser `pattern` (voir page 155) ou `change` (voir page 181) pour préparer le terrain à `apply`. Il est clair et moral que la puissance d'expression du Calcul des Constructions Inductives se paye par une perte d'automaticité.

Recherche de théorèmes pour apply

Lorsque l'on doit résoudre un but de la forme “ $\Gamma \overset{?}{\vdash} p\ \ a_1\ \ \ldots a_n$ ”, où p est un prédicat déclaré dans le contexte global, il est utile de demander au système *Coq* de produire l'ensemble des théorèmes du contexte global qui permettent de de prouver une expression de la forme “ $p\ \ a_1\ \ \ldots a_n$ ” ; il s'agit en fait de tous les théorèmes dont le type final est une application de p à n arguments. Cette requête est faite avec la commande “ `Search` p ”.

Par exemple, nous pouvons demander de retrouver l'ensemble des théorèmes qui permettent de prouver une comparaison entre deux nombres entiers :

```
Search Zle.
```
...
Zle_0_nat : $\forall$ *n:nat, (0* $\leq$ *Z_of_nat n)%Z*
...
Zmult_le_approx : $\forall$ *n m p:Z,*
*(n > 0)%Z→(n > p)%Z→(0≤m*n+p)%Z→(0≤m)%Z*

Lorsqu'il y a un trop grand nombre de réponses, il est possible de demander une recherche plus spécifique, en précisant la forme que devraient prendre certains arguments. La commande s'appelle alors `SearchPattern`, dont voici un exemple d'utilisation :

```
SearchPattern (_ + _ ≤ _)%Z.
```
Zplus_le_compat_l: $\forall$ *n m p:Z, (n* $\leq$ *m)%Z →(p+n* $\leq$ *p+m)%Z*
Zplus_le_compat_r: $\forall$ *n m p:Z, (n* $\leq$ *m)%Z →(n+p* $\leq$ *m+p)%Z*
Zplus_le_compat:
$\forall$ *n m p q:Z, (n* $\leq$ *m)%Z →(p* $\leq$ *q)%Z →(n+p* $\leq$ *m+q)%Z*

Notez que les expressions qui peuvent varier entre les différents théorèmes sont représentées par des symboles '`_`'. On effectue donc un filtrage des théorèmes par un schéma, et les « jokers » jouent le rôle de variables de filtrage anonyme. La commande `SearchPattern` permet également de chercher les théorèmes en effectuant un filtrage non linéaire, il faut alors associer un nom de la forme `?X` ou `?my_name` aux emplacements devant être associés à la même expression (les deux expressions doivent être syntaxiquement égales, il ne suffit pas qu'elles soient égales modulo convertibilité). L'exemple ci-dessous montre une recherche avec un schéma de filtrage non linéaire :

```
SearchPattern (?X1 * _ ≤ ?X1 * _)%Z.
```
Zmult_le_compat_l: $\forall$ *n m p:Z, (n* $\leq$ *m)%Z →(0* $\leq$ *p)%Z →(p*n* $\leq$ *p*m)%Z*

Il faut cependant remarquer que ces fonctions de recherche ne parcourent que l'environnement courant. Un théorème extrait d'une bibliothèque de *Coq* ne sera trouvé que si cette bibliothèque a été préalablement chargée.

6.1.4 La tactique `unfold`

Les types que nous manipulons dans nos développements utilisent fréquemment des constantes ayant fait l'objet d'une définition préalable. Si une constante est transparente (voir page 86), il peut être utile d'effectuer une δ-réduction (fréquemment suivie de β-réductions) dans le but ou dans une hypothèse.

La tactique `unfold` $q_1 \ \ldots q_n$ provoque une δ-réduction du but sur les occurrences des identificateurs qualifiés $q_1 \ \ldots \ q_n$, suivie d'une mise sous forme $\beta\iota$-normale.

Considérons l'exemple suivant, où le sous-but " `n < S p` " est réduit en " `S n ≤ S p` " par la tactique " `unfold lt` ". *Rappelons que les écritures*

“ t_1 `<` t_2 ” et “ $t_1 \leq t_2$ ” ne sont que des abréviations des applications “ `lt` t_1 t_2 ” et “ `le` t_1 t_2 ”.

```
Theorem lt_S : ∀n p:nat, n < p → n < S p.
Proof.
 intros n p H.
 unfold lt; apply le_S; trivial.
Qed.
```

On remarquera que pour l'occurrence de `lt` présente *dans l'hypothèse `H`*, la tactique `assumption` prend en charge la δ-réduction sur `lt` sans qu'il soit nécessaire de faire de nouveau appel à `unfold`.

Il est important d'insister sur la condition de transparence des identificateurs à δ-réduire. Dans l'exemple suivant, nous définissons une constante opaque (en sauvegardant sa définition par `Qed`) et perdons la possibilité d'exploiter sa définition précise. Si la définition de `opaque_f` avait été terminée par `Defined`, la tactique `unfold` aurait pu être appliquée. Remarquons que dans ce cas l'utilisation d'`assumption` serait à éviter (voir page 86).

```
Definition opaque_f : nat→nat→nat.
  intros; assumption.
Qed.

Lemma bad_proof_example_for_opaque : ∀x y:nat, opaque_f x y = y.
 intros; unfold opaque_f.
  Error: opaque_f is opaque
Abort.
```

La tactique `unfold` permet également de réduire seulement quelques-unes des occurrences possibles d'un identificateur, dans les cas où l'expansion de toutes les occurrences de ce symbole pourrait nuire à la lisibilité des buts. La commande prend alors la forme “ `unfold` *id* `at` n_1 n_2 `...` ”, pour indiquer que seules les occurrences en position n_1 n_2 ..., du symbole *id* doivent faire l'objet d'une δ-réduction.

Le fragment de preuve suivant montre un exemple de cette utilisation de la tactique `unfold` ; en effet seule la première occurrence de `Zsquare_diff` présente un intérêt pour la poursuite de la preuve (à finir en exercice.)

```
Open Scope Z_scope.

Definition Zsquare_diff (x y:Z):= x*x - y*y.

Theorem unfold_example :
 ∀x y:Z,
   x*x = y*y →
   Zsquare_diff x y * Zsquare_diff (x+y)(x*y) = 0.
Proof.
intros x y Heq.
```

```
unfold Zsquare_diff at 1.
  ...
  Heq : x*x = y*y
  ============================
    (x*x - y*y) * Zsquare_diff (x+y)(x*y) = 0
```

6.2 Connecteurs logiques

Nous avons vu dans le chapitre 5 que les règles d'introduction et élimination des connecteurs logiques usuels étaient décrites par des constantes correctement typées dans le calcul des constructions. Le système *Coq* fournit également quelques tactiques pour manipuler ces connecteurs logiques de façon intuitive.

6.2.1 Règles d'introduction et d'élimination

Dans cet ouvrage, nous utilisons plusieurs fois les notions de règles d'introduction et d'élimination pour les connecteurs logiques. Ces mots viennent de la théorie de la preuve et mérite une attention particulière car ils apparaissent aussi en *Coq*, particulièrement dans le nom des tactiques `intros` et `elim`.

Nous parlons d'*introduction* lorsqu'une étape de raisonnement permet d'introduire une nouvelle formule comme un fait établi dans le processus logique. Une règle d'introduction pour un connecteur logique est une fonction qui produit un terme dont le type est une formule construite avec ce connecteur. Par exemple, la constante `conj` (voir section 5.3.5) a le type suivant :

```
Check conj.
conj : ∀ A B:Prop, A→B→A∧B
```

Elle peut être utilisée pour construire une preuve de $A \wedge B$ si nous sommes également capable de construire des preuves de A et B.

Nous parlons d'*élimination* lorsqu'une étape de raisonnement permet d'utiliser une formule déjà prouvée dans le processus logique pour en tirer les conséquences possibles. Une règle d'élimination pour un connecteur logique est une fonction qui prend comme argument un terme dont le type est une formule construite avec ce connecteur pour produire la preuve d'une autre formule. Le connecteur logique est alors « éliminé » du discours car nous pouvons continuer la preuve en utilisant seulement les conséquences de la formule. Par exemple, la règle d'élimination pour la conjonction est `and_ind` :

```
Check and_ind.
and_ind : ∀ A B P:Prop, (A→B→P)→A∧B→P
```

Si nous voulons démontrer une formule C, nous disposons d'une preuve de $A \wedge B$, et nous utilisons `and_ind`, nous devons seulement produire une preuve de $A \to B \to C$. Dans cette dernière formule, A et B sont les conséquences de $A \wedge B$ et la conjonction a disparu.

Dans l'utilisation pratique de *Coq*, le processus de raisonnement est effectué dans des preuves dirigées par les buts. Nous utilisons des tactiques pour représenter les étapes de raisonnement et pour cette raison nous parlons plutôt de tactiques d'introduction et d'élimination.

Les tactiques d'introduction permettent de prouver des buts dont la structure principale est donnée par un connecteur logique. Par exemple, si nous avons un but de la forme $A \wedge B$, nous pouvons utiliser la tactique `split` et ceci nous mène à deux nouveaux buts A et B. C'est donc la tactique `split` qui joue ici le rôle de tactique d'introduction.

Les tactiques d'élimination permettent d'utiliser des faits dont la structure est donnée par un connecteur logique. Ces faits sont donnés sous la forme de types de termes du calcul des constructions, et le plus fréquemment ces termes sont des identificateurs du contexte. Par exemple, si nous avons une hypothèse H de type $A \wedge B$, nous pouvons utiliser la tactique “ `elim` H ” pour avancer en utilisant A et B comme des faits séparés. C'est presque toujours la tactique `elim` qui joue le rôle de tactique d'élimination.

L'implication et la quantification universelle ne s'intègrent pas dans ce cadre, car ces connecteurs sont représentés directement par la notion la plus primitive du calcul des constructions : les produits. La tactique `intro` joue bien le rôle d'une tactique d'introduction pour l'implication. Néanmoins, la tactique d'élimination pour les produits est plutôt la tactique `apply`.

Dans le reste de cette section, nous présentons les connecteurs de base et nous décrivons les tactiques d'introduction et d'élimination. D'un point de vue pragmatique, l'introduction est un moyen de prouver un but ayant un certain connecteur logique et l'élimination est un moyen d'utiliser une hypothèse du contexte ou un théorème de l'environnement.

6.2.2 Elimination du faux

Il n'y a pas de règle d'introduction pour la constante `False` qui représente la proposition fausse. Ainsi, on ne pourra démontrer cette proposition que dans un contexte contenant déjà une contradiction.

En revanche, la règle d'élimination du faux indique comment nous pouvons utiliser le fait qu'il existe une contradiction dans le contexte. Pour résumer, cette règle exprime que l'on peut déduire n'importe quoi de la proposition fausse. Les tactiques du système *Coq* permettent de mettre en œuvre rapidement cette étape de raisonnement, en particulier une nouvelle tactique qui jouera un rôle important dans toute la suite du livre, la tactique `elim`.

Pour illustrer ce type de raisonnement, nous nous plaçons dans un contexte particulier contenant une hypothèse affirmant `False` et nous cherchons à démontrer une égalité contradictoire.

Il existe deux façons de prouver cette égalité à l'aide de tactiques ; la première avec `apply`, la seconde utilisant la tactique `elim`. En quelques mots, si t a pour type `False`, un appel de “ `elim` t ” résoud immédiatement le but courant. Cette seconde manière est particulièrement recommandée, à cause du grand nombre d'applications et du grand confort d'utilisation de la tactique `elim`.

```
Section ex_falso_quodlibet.
 Hypothesis ff : False.

 Lemma ex1 : 220 = 284.
 Proof.
   apply False_ind.
   exact ff.
 Qed.

 Lemma ex2 : 220 = 284.
 Proof.
   elim ff.
 Qed.

End ex_falso_quodlibet.
Print ex2.
```

ex2 = fun ff:False ⇒ False_ind (220 = 284) ff
    : False → 220 = 284

La preuve construite par la tactique `elim` est la même que la preuve construite par la tactique `apply`, mais la tactique `elim` s'est chargée de trouver le théorème `False_ind` pour nous.

On trouvera dans les sections 11.1.1 et 10.2.3 des exemples d'utilisation de `False_rec`.

6.2.3 Négation

Les démonstrations sur la négation se ramènent toutes à des démonstrations sur la proposition fausse. Prouver la négation d'une formule, c'est démontrer que supposer cette formule mènerait contradiction représentée par `False`. Utiliser une hypothèse qui est la négation d'une formule c'est montrer que cette formule est vérifiée et que le contexte est donc contradictoire. Nous donnons quelques exemples dans cette section.

La preuve ci-dessous reprend le théorème `absurd` de *Coq*.

```
Theorem absurd : ∀P Q:Prop, P→~P→Q.
Proof.
 intros P Q p H.
 elim H.
  ...
  p : P
  H : ~P
  ============================
   P
```

```
 assumption.
Qed.

Print absurd.
absurd =
fun (P Q:Prop)(p:P)(H:~P) ⇒ False_ind Q (H p)
     : ∀ P Q:Prop, P → ~P → Q
Argument scopes are [type_scope type_scope _ _]
```

Nous utilisons l'hypothèse `(H:~P)` dans la tactique “ `elim H` ” parce que nous savons que cette hypothèse contredit l'autre hypothèse `p`.

Notons que cette preuve utilise la règle de conversion ; en effet, cette dernière permet de considérer que l'hypothèse `H`, de type `~P`, a également pour type `P→False` et par conséquent l'application “ `H p` ” a bien pour type `False`.

Certains théorèmes concernant la négation n'utilisent pas de propriétés particulières de `False`. Dans la preuve suivante, l'appel à `intros` provoque la δ-réduction de l'occurrence la plus externe de `not` (correspondant au symbole ~ le plus à gauche). La tactique “ `apply H` ” provoque la réduction de `~P` en `P→False` :

```
Theorem double_neg_i : ∀P:Prop, P→~~P.
Proof.
 intros P p H.
  ...
  P : Prop
  p : P
  H : ~P
  ============================
    False
 apply H; assumption.
Qed.
```

La proposition `False` ne joue aucun rôle particulier dans le théorème précédent, qui est une simple instance de la règle de modus ponens :

```
Theorem modus_ponens :∀P Q:Prop, P→(P→Q)→Q.
Proof.
 auto.
Qed.

Theorem double_neg_i' : ∀P:Prop, P→~~P.
Proof.
 intro P.
Proof (modus_ponens P False).
```

De même, la règle de *contraposition* est une application directe de la « transitivité de l'implication » :

```
Theorem contrap :∀A B:Prop, (A→B)→~B→~A.
Proof.
 intros A B; unfold not.
  ...
  ============================
     (A→B)→(B→False)→A→False
 apply imp_trans.
Qed.
```

Exercice 6.3 Prouver chacune des propositions suivantes :

– `~False`
– `∀P : Prop, ~~~P→~P`
– `∀P Q: Prop,~~~P→P→Q`
– `∀P Q:Prop, (P→Q)→~Q→~P`
– `∀P Q R:Prop, (P→Q)→(P→~Q)→P→R`

Pour chaque preuve n'utilisant pas `False_ind`, montrer que le théorème correspondant peut se dériver d'un théorème de la logique propositionnelle minimale.

Exercice 6.4 Parmi les fautes de raisonnement usuelles, on trouve des pseudo-règles d'inférence, dont l'utilisation peut mener à des absurdités.

Prenons deux exemples bien connus de dyslexie : la dyslexie de l'implication (confondre $P{\rightarrow}Q$ et $Q{\rightarrow}P$), et la contraposée dyslexique :

```
Definition dyslexic_imp := ∀P Q:Prop, (P→Q)→(Q→P).

Definition dyslexic_contrap := ∀P Q:Prop, (P→Q)→(~P→~Q).
```

Montrer que, si l'un ou l'autre de ces types était habité, alors on pourrait prouver `False`, donc n'importe quoi.

6.2.4 Conjonction et disjonction

Trois tactiques sont associées aux règles d'introduction des connecteurs logiques pour la conjonction et la disjonction.

– `split` remplace “ `intros; apply conj` ”
– `left` remplace “ `intros; apply or_introl` ”
– `right` remplace “ `intros; apply or_intror` ”

Forts de ces outils, nous pouvons prouver interactivement les deux propositions que nous avons déjà vues en section 5.3.5.

```
Theorem conj3' : ∀P Q R:Prop, P→Q→R→P∧Q∧R.
Proof.
 repeat split; assumption.
Qed.

Theorem disj4_3' : ∀P Q R S:Prop, R→P∨Q∨R∨S.
Proof.
```

```
  right; right; left; assumption.
Qed.
```

En fait, un seul appel à `auto` suffit pour chacun de ces théorèmes (voir chapitre 8).

Les règles d'élimination sont utiles pour utiliser des hypothèses qui sont des disjonctions ou des conjonctions. La tactique `elim` est particulièrement adaptée pour ce besoin : elle applique la règle d'élimination associée au connecteur logique.

Voici un exemple utilisant l'élimination et l'introduction de conjonctions :

```
Theorem and_commutes : ∀A B:Prop, A∧B→B∧A.
Proof.
 intros A B H.
  ...
  H : A∧B
  ============================
   B∧A
 elim H.
  ...
  H : A∧B
  ============================
  A→B→B∧A
 split; assumption.
Qed.
```

Après l'étape " `elim H` ", le nouveau but contient deux implications dont les prémisses sont les deux termes de la conjonction. L'exemple similaire pour la disjonction doit être traité avec soin. Malgré l'apparente symétrie de l'énoncé, la preuve doit commencer par une élimination de l'hypothèse $A \vee B$, suivie d'une introduction par `left` (resp. `right`) ; en effet, commencer par une règle d'introduction sur la conclusion $B \vee A$ obligerait à choisir dès le départ entre `left` et `right` et casserait la symétrie du but à prouver.

```
Theorem or_commutes : ∀A B:Prop, A∨B→B∨A.
Proof.
 intros A B H.
  ...
  H : A∨B
  ============================
   B∨A
 elim H.
 ...
  ============================
   A→B∨A

subgoal 2 is:
```

```
  B→B∨A
 auto.
 auto.
Qed.
```

L'élimination de `H` correspond bien à une preuve par cas. Cette preuve se termine automatiquement parce que les constructeurs `or_introl` et `or_intror` sont dans la base de données utilisée par `auto`.

Rappelons ici que le système *Coq* fournit des tactiques automatiques pour traiter les formules logiques composées principalement de conjonctions et de disjonctions, ce sont les tactiques `tauto` et `intuition`. Nous invitons les lecteurs à essayer ces tactiques sur les différents exercices donnés dans ces deux sections.

Exercice 6.5 Démontrer le théorème suivant :

```
∀(A:Set)(a b c d:A), a=c ∨ b=c ∨ c=c ∨ d=c
```

Exercice 6.6 Démontrer les théorèmes suivants :

- `∀A B C:Prop, A∧(B∧C) → (A∧B)∧C`
- `∀A B C D: Prop, (A→ B)∧(C→D)∧A∧C → B∧D`
- `∀A: Prop, ~(A∧~A)`
- `∀A B C: Prop, A∨(B∨C) → (A∨B)∨C`
- `∀A: Prop, ~~(A∨~A)`
- `∀A B: Prop, (A∨B)∧~A → B`

Exercice 6.7 * On considère les cinq définitions suivantes, chacune pouvant être considérée comme une caractérisation de la logique dite « classique » :

```
Definition peirce := ∀P Q:Prop, ((P→Q)→P)→P.
Definition classic := ∀P:Prop, ~~P → P.
Definition excluded_middle := ∀P:Prop, P∨~P.
Definition de_morgan_not_and_not := ∀P Q:Prop, ~(~P∧~Q)→P∨Q.
Definition implies_to_or := ∀P Q:Prop, (P→Q)→(~P∨Q).
```

Prouver que ces cinq propositions sont logiquement équivalentes.

6.2.5 À propos de `repeat`

Dans la preuve de `conj3'`, page 149, nous utilisons la tacticielle `repeat`, permettant de répéter indéfiniment une tactique jusqu'à l'échec ou la résolution totale du but. La tactique “ `repeat` *`tac`* ” applique *tac* sur le but courant et s'arrête sans échec si *tac* échoue (comme la tactique `try`). En revanche, si la tactique *tac* réussit, la tactique “ `repeat` *`tac`* ” s'applique à nouveau sur tous les buts engendrés. Ce processus s'arrête lorsqu'il ne reste plus de sous-but à résoudre, ou bien lorsque *tac* échoue sur tous les sous-buts engendrés. Dans le cas contraire, ce processus peut continuer indéfiniment.

Exercice 6.8 Que font les tactiques `repeat idtac` et `repeat fail` ?

6.2.6 La quantification existentielle

Lorsque l'on utilise la règle d'introduction pour la quantification existentielle, il est nécessaire de fournir le témoin. C'est aussi le cas si l'on utilise la tactique spécialisée pour ce connecteur, la tactique `exists`.

L'utilisation de la règle d'élimination se fait par la tactique `elim`. De façon symétrique à l'introduction, l'élimination d'une hypothèse contenant une quantification existentielle produit un témoin satisfaisant la formule quantifiée.

Dans la preuve suivante, l'élimination de l'hypothèse[1] “ `Ex P` ” fournit un témoin `(a:A)` ainsi qu'une hypothèse `(H:P a)`. Le témoin `a` peut alors servir d'argument pour la tactique `exists`.

```
Theorem ex_imp_ex :
 ∀(A:Type)(P Q:A→Prop), (ex P)→(∀x:A, P x → Q x)→(ex Q).
Proof.
  intros A P Q H H0.
  elim H; intros a Ha.
 ...
 H : ex P
 H0 : ∀x:A, P x → Q x
 a : A
 Ha : P a
 ============================
  ex Q

  exists a.
 ...
 ============================
 Q a

 apply H0; exact Ha.
Qed.
```

Exercice 6.9 * Dans un contexte déclarant `A: Type` et `P,Q: A→Prop`, montrer les formules suivantes :

```
(∃x : A | P x ∨ Q x)→(ex P)∨(ex Q)

(ex P)∨(ex  Q)→∃x : A | P x ∨ Q x

(∃x : A | (∀R:A → Prop, R x))→ 2 = 3

(∀x:A, P x)→ ~(∃y : A | ~ P y)
```

1. Rappelons les notations vues en page 130 : on peut lire la proposition “ `Ex P` ” comme “ `∃a:A|P a` ”.

La réciproque de la quatrième formule n'est pas prouvable dans la logique intuitionniste de *Coq* ; en revanche, cette réciproque est prouvée dans la bibliothèque *Classical* qui permet de travailler en logique classique ; ce résultat fait l'objet du théorème `not_ex_not_all`.

6.3 L'égalité et la réécriture

Le raisonnement sur les propositions contenant des égalités utilise deux sortes de tactiques, selon que l'égalité est le but à prouver ou bien fait partie des hypothèses.

6.3.1 Introduction de l'égalité

La tactique `reflexivity`, synonyme de “ `apply refl_equal` ”, permet de prouver l'égalité entre deux termes convertibles.

```
Lemma L36 : 6*6=9*4.
Proof.
 reflexivity.
Qed.
Print L36.
L36 = refl_equal (9*4)
   : 6*6 = 9*4
```

En revanche, le dialogue suivant montre que l'exigence de convertibilité ne permet pas de traiter tous les cas d'égalité :

```
Lemma diff_of_squares : ∀a b:Z, ((a+b)*(a-b) = a*a-b*b)%Z.
Proof.
  intros.
  reflexivity.
 Error: Impossible to unify (a * a - b * b)%Z with ((a + b) * (a - b))%Z
```

En effet, `a` et `b` étant des variables libres, aucun calcul ne peut être déclenché, et rien ne permet de réduire les deux termes de l'égalité en deux termes unifiables. La tentative d'unification confronte alors un terme de la forme “ t_1 `*` t_2 ” avec un autre de la forme “ t_3 `-` t_4 ” ; or ces deux termes ne sont pas unifiables.

Afin de ne pas laisser au lecteur cet arrière-goût d'échec, montrons comment terminer la preuve en chargeant la bibliothèque adéquate et en utilisant la tactique automatique `ring` (voir section 8.3.1).

```
 Require Import ZArithRing.
 ring.
Qed.
```

6.3.2 Tactiques de réécriture

Quand nous utilisons une égalité, nous voulons habituellement exprimer qu'une certaine valeur peut être remplacée par une autre parce qu'elles sont égales. Ce type de raisonnement est fourni par une tactique appelée `rewrite`.

Soit e un terme dont le type est “ $\forall$ `(`x_1`:`T_1`) ... (`x_n`:`T_n`),` $a = b$ ” ; la tactique “ `rewrite` e ”, appliquée à un sous-but d'énoncé “ $P\ a$ ”, engendre un nouveau sous-but d'énoncé “ $P\ b$ ”. Lorsque la conclusion du but n'apparaît pas directement sous la forme d'une propriété P appliquée à a, cette tactique remplace toutes les occurrences de a dans le but courant par b. Cette tactique peut également créer plusieurs autres sous-buts, dans le cas où certains x_i n'apparaissent pas dans le type final de e (voir la section consacrée aux réécritures conditionnelles, page 156).

Illustrons cette tactique sur un petit exemple :

```
Theorem eq_sym' : ∀(A:Type)(a b:A), a=b→b=a.
Proof.
 intros A a b e.
  ...
  A : Type
  a : A
  b : A
  e : a=b
  ============================
   b=a

 rewrite e.

  ...
  e : a=b
  ============================
   b=b

 reflexivity.
Qed.
```

Réécriture droite-gauche. Conservons les notations du paragraphe précédent ; si nous souhaitons remplacer les occurrences de b par a dans le sous-but, nous pouvons utiliser la tactique “ `rewrite <-` e ”.

Exemple. Le développement suivant montre quelques utilisations élaborées de `rewrite`. Les théorèmes utilisés dans la preuve de `Zmult_distr_1` sont des égalités universellement quantifiées sur `Z`, et trouvées dans les bibliothèques de *Coq*.

*Zmult_plus_distr_l : ∀ n m p:Z, (n+m)*p = n*p + m*p*
*Zmult_1_l : ∀ n:Z, 1*n = n*

```
Theorem Zmult_distr_1 : ∀n x:Z, n*x+x = (n+1)*x.
Proof.
 intros.
  ...
  ============================
   n*x + x = (n+1)*x

 rewrite Zmult_plus_distr_l.
  ...
  ============================
   n*x + x = n*x + 1*x

 rewrite Zmult_1_l.
   ...
============================
   n*x + x = n*x +  x

 trivial.
Qed.
```

La variante " `rewrite in` H ". Il est possible d'effectuer une réécriture dans une hypothèse H, par la tactique " `rewrite` e `in` H ". Bien sûr cette variante peut être combinée avec la variante de direction citée plus haut.

6.3.3 La tactique `pattern`

Si nous voulons utiliser `Zmult_distr_1` pour prouver l'égalité $x + x + x + x + x = 5x$, il faut réécrire la première occurrence de `x` en $1 \times x$. Il est possible en *Coq* de distinguer une ou plusieurs occurrences d'un sous-terme t par la tactique `pattern`. Cette tactique prend comme argument la liste des numéros d'occurrences à remplacer ainsi que le sous-terme t, et crée un nouveau but de la forme " $P\ t$ " où P est une abstraction. Les occurrences distinguées de t sont remplacées par la variable liée de P dans cette abstraction. Seules ces occurrences sont modifiées dans une réécriture.

```
Theorem regroup : ∀x:Z, x+x+x+x+x = 5*x.
Proof.
 intro x; pattern x at 1.
  ...
  x : Z
```

```
 ============================
  (fun z:Z ⇒ z+x+x+x+x = 5*x) x

 rewrite <- Zmult_1_l.
...
 ============================
  1*x + x + x + x + x = 5*x

 repeat rewrite Zmult_distr_1.
 ...
 ============================
   (1+1+1+1+1)*x = 5*x

 auto with zarith.
Qed.
```

Exercice 6.10 Prouver le théorème suivant :

```
Theorem plus_permute2 :
   ∀n m p:nat, n+m+p = n+p+m.
```

On se limitera dans un premier temps à utiliser les tactiques `rewrite`, `pattern`, `intros`, `apply` et `reflexivity`, sans s'aider d'automatisme. On pourra en revanche utiliser les deux théorèmes suivants de la bibliothèque `Arith` :

plus_comm : ∀ n m:nat, n+m = m+n

plus_assoc
   : ∀ n m p:nat, n+(m+p) = n+m+p

6.3.4 * Réécritures conditionnelles

Dans la définition de la tactique “ `rewrite` e ”, le type du terme e est un produit dépendant dont la conclusion est une égalité. Ce cadre général nous permet d'utiliser `rewrite` sur des égalités conditionnelles.

Prenons un exemple volontairement très simple : nous commençons par prouver dans $\mathbb{N}$ que si $n \leq p < n+1$, alors $n = p$. Ce résultat s'exprime par le lemme suivant, dont la preuve, très courte, utilise la tactique `omega` (voir section 8.3.2). Cette preuve peut être consultée sur notre site [11].

```
Check le_lt_S_eq.
le_lt_S_eq
   : ∀ n p:nat, n ≤ p → p < S n → n = p
```

Cette égalité conditionnelle est utilisée pour prouver une petite propriété arithmétique :

```
Lemma cond_rewrite_example : ∀n:nat,
   8 < n+6 →  3+n < 6 → n*n = n+n.
Proof.
 intros n  H H0.
  ...
  n : nat
  H : 8 < n+6
  H0 : 3+n < 6
  ============================
  n*n = n+n
```

Or le type de " `le_lt_S_eq 2 n` " est la proposition ci-dessous :

```
Check (le_lt_S_eq 2 n).
le_lt_S_eq 2 n : 2 ≤ n → n < 3 → 2 = n
```

L'appel de " `rewrite <- (le_lt_S_eq 2 n)` " engendre alors les trois sous-buts suivants :

1. `2 * 2 = 2 + 2`
2. `2 ≤ n`
3. `n < 3`

Voici donc la fin de la preuve, qui utilise les deux lemmes suivants :

plus_lt_reg_l : $\forall$ *n m p:nat,* $p+n < p+m \to n < m$

plus_le_reg_l : $\forall$ *n m p:nat,* $p+n \leq p+m \to n \leq m$

```
 rewrite <- (le_lt_S_eq 2 n).
3 subgoals
  ...
  ============================
   2*2 = 2+2

subgoal 2 is:

   2 ≤ n

subgoal 3 is:

   n < 3

 simpl; auto.
 apply  plus_le_reg_l with (p := 6).
 rewrite plus_comm in H; simpl; auto with arith.
 apply  plus_lt_reg_l with (p:= 3); auto with arith.
Qed.
```

Exercice 6.11 * D'abord en utilisant explicitement `eq_ind`, puis dans une seconde version avec `rewrite`, prouver le résultat suivant :

```
Theorem eq_trans : ∀(A:Type)(x y z:A), x = y → y = z → x = z.
```

6.3.5 Recherche de théorèmes pour la réécriture

La commande " `SearchRewrite` *motif* " permet de lister les théorèmes dont la conclusion est une égalité dont un des membres est une instance de *motif*. Ce dernier est un terme dont les parties inconnues sont remplacées par le symbole '_'. À titre d'exemple, le théorème `Zmult_1_l` utilisé plus haut a été trouvé par la commande suivante :

```
SearchRewrite (1 * _)%Z.
```
*Zmult_1_l: ∀ n:Z, (1*n)%Z = n*

6.3.6 Autres tactiques liées à l'égalité

Coq contient d'autres tactiques permettant de simplifier l'utilisation de l'égalité : `replace`, `cutrewrite`, `symmetry` et `transitivity`, nous conseillons de consulter le manuel de référence de *Coq* à leur sujet.

6.4 Tableau récapitulatif des tactiques

Dans les démonstrations dirigées par les buts, chaque connecteur logique dispose de deux catégories de tactiques adaptées, l'une pour les usages dans les hypothèses (ce sont les tactiques d'élimination), l'autre pour les usages dans les buts (tactiques d'introduction). Les différentes tactiques pour ces usages peuvent être résumés dans le tableau suivant :

	$\Rightarrow$	$\Leftrightarrow$	$\forall$	$\wedge$
Hypothèse	`apply`	`elim`	`apply`	`elim`
But	`intros`	`split`	`intros`	`split`

	$\vee$	$\exists$	$\sim$	$=$
Hypothèse	`elim`	`elim`	`elim`	`rewrite`
but	`left` ou `right`	`exists` v	`red`	`reflexivity`

6.5 *** Définitions imprédicatives

6.5.1 Avertissement

Afin de clore ce chapitre sur le produit dépendant, nous présentons une construction différente de quelques notions vues précédemment : le faux et le vrai, la négation, ainsi que l'égalité. Ces notions sont bien sûr prédéfinies en *Coq* et accompagnées d'outils efficaces ; il n'est donc pas nécessaire de proposer d'autres définitions que celles (bien) choisies dans le système, mais nous croyons que les développements qui suivent ont un grand intérêt pour la compréhension du produit dépendant et de sa puissance d'expression. Le lecteur peut donc, soit sauter cette section en sachant que son contenu ne sera jamais utilisé ultérieurement, soit s'y attarder en appréciant les subtilités de la logique d'ordre supérieur.

Une définition *imprédicative* utilise le fait qu'une proposition A de la forme " $\forall\, P$:`Prop,` Q " introduit une sorte de circularité. En effet, A se définit par une quantification sur toutes les propositions, y compris A elle-même. Seule une étude précise d'un système de typage autorisant cette possibilité, comme celle faite par Thierry Coquand[28], permet de montrer qu'elle ne conduit pas à un paradoxe fatal. Remarquons que le système *CoC* (*Calculus* of *Constructions*), prédécesseur de *Coq*, utilisait ce type de définitions. Lors de la mise sur pied de *Coq*, cette représentation a été remplacée par des constructions inductives (voir chapitres suivants).

Nous donnons ci-dessous deux exemples de définitions imprédicatives : le faux et l'égalité.

6.5.2 Le Vrai et le Faux

Les deux définitions suivantes proposent une version de `True` et `False` ; le préfixe "`my`" permet d'éviter toute confusion avec les constantes prédéfinies en *Coq*.

```
Definition my_True : Prop
:= ∀P:Prop, P → P.

Definition my_False : Prop
:= ∀P:Prop, P.
```

La preuve suivante, obtenue en fournissant un terme de preuve de `my_True` justifie le rôle de « proposition vraie » attribué à cette constante :

```
Theorem my_I : my_True.
Proof.
 intros P p; assumption.
Qed.
```

Il est amusant de constater que la définition de `my_False` peut se paraphraser en « tout est vrai », c'est à dire que l'on modélise *Faux* par l'incohérence. Les propriétés méta-mathématiques du Calcul des Constructions impliquent qu'il n'existe aucune preuve de `my_False` dans le contexte et l'environnement vides. Pour montrer que cette constante reflète bien le concept de fausseté, prouvons l'équivalent de `False_ind` :

```
Theorem my_False_ind : ∀P:Prop, my_False→P.
Proof.
 intros P F; apply F.
Qed.
```

Exercice 6.12 * On considère la définition suivante :

```
Definition my_not (P:Prop) : Prop := P→my_False.
```

Reprendre l'exercice 6.3 en utilisant les constantes `my_False` et `my_not` au lieu de `False` et `not`.

6.5.3 Une curiosité : l'égalité de Leibniz

L'égalité de Leibniz est la traduction de la définition informelle :

« *a* et *b* sont égaux si toute propriété de *a* est une propriété de *b* »

```
Section leibniz.
 Set Implicit Arguments.
 Unset Strict Implicit.
 Variable A : Set.

 Definition leibniz (a b:A) : Prop :=
 ∀P:A → Prop, P a → P b.
```

Nous remarquons également le caractère imprédicatif de cette définition, dans la mesure où la quantification sur `P` porte sur tout prédicat défini sur `A`, y compris " `leibniz A a` ". À titre d'exemple, nous développons une preuve de symétrie[2] de la relation " `leibniz` *A* " :

```
Require Import Relations.

Theorem leibniz_sym : symmetric A leibniz.
Proof.
 unfold symmetric.

 ...
 ============================
  ∀x y:A, leibniz x y → leibniz y x

 unfold leibniz; intros x y H Q.
  ...
```

2. Les définitions usuelles sur les relations : symétrie, transitivité, inclusion, etc. se trouvent dans le module `Relations` de la bibliothèque standard de *Coq*

```
  H : ∀ P:A → Prop, P x → P y
  Q : A→Prop
  ============================
   Q y → Q x

 apply H; trivial.
Qed.
```

Il est intéressant de remarquer l'utilisation de la tactique `apply` dans cette preuve : le sous-but courant a pour énoncé la proposition “ `Q y →Q x` ”, et l'unification de ce terme avec “ `P y` ” donne la substitution suivante :

$\sigma(\mathtt{P}) = $ `fun z:A ⇒ Q z → Q x`.

Le sous-but engendré est alors la proposition “ $\sigma(P)$ `x` ”, qui se réduit en “ `Q x → Q y` ”.

On remarquera qu'une utilisation de la tactique `intros` (sans paramètres) donnerait le but suivant, sur lequel “ `apply H` ” serait inopérante :

```
  ...
  H0 : P y
  ============================
   P x
```

Exercice 6.13 ** Compléter le développement suivant :

```
Theorem leibniz_refl : reflexive A leibniz.

Theorem leibniz_trans : transitive A leibniz.

Theorem leibniz_equiv : equiv A leibniz.

Theorem leibniz_least_reflexive :
 ∀R:relation A, reflexive A R → inclusion A leibniz R.

Theorem leibniz_eq : ∀a b:A, leibniz a b → a = b.

Theorem eq_leibniz : ∀a b:A, a = b → leibniz a b.

Theorem leibniz_ind :
 ∀(x:A)(P:A→Prop), P x → ∀y:A, leibniz x y → P y.
Unset Implicit Arguments.
End leibniz.
```

6.5.4 Quelques autres connecteurs logiques et quantificateurs

Nous proposons des définitions imprédicatives de la conjonction, de la disjonction et du quantificateur existentiel.

```
Definition my_and (P Q:Prop) :=
  ∀R:Prop, (P→Q→R)→R.

Definition my_or (P Q:Prop) :=
  ∀R:Prop, (P→R)→(Q→R)→R.

Definition my_ex (A:Set)(P:A→Prop) :=
  ∀R:Prop, (∀x:A, P x → R)→R.
```

Exercice 6.14 * Afin de vous familiariser avec ces définitions, prouver les énoncés suivants :

```
∀P Q:Prop, my_and P Q → P

∀P Q:Prop, my_and P Q → Q

∀P Q R:Prop, (P→Q→R)→ my_and P Q → R

∀P Q:Prop, P → my_or P Q

∀P Q:Prop, Q → my_or P Q

∀P Q R:Prop, (P→R)→(Q→R)→ my_or P Q → R

∀P:Prop, my_or P my_False → P

∀P Q:Prop, my_or P Q → my_or Q P

∀(A:Set)(P:A→Prop)(a:A), P a → my_ex A P

∀(A:Set)(P:A→Prop),
   my_not (my_ex A P)→ ∀a:A, my_not (P a)
```

Le cas de l'inégalité

Afin de bien comprendre les définitions imprédicatives, prenons l'exemple de la relation $\leq$ sur $\mathbb{N}$, et essayons d'en donner une telle définition. Intuitivement, on pourrait définir « être supérieur ou égal à n » (c'est à dire $\lambda p.n \leq p$) par la propriété suivante (portant sur les prédicats de type `nat→Prop`).

> « satisfaire toute propriété vraie pour n et stable par passage au successeur »

Cette définition a bien un caractère imprédicatif : en effet, nous quantifions sur tout prédicat unaire sur $\mathbb{N}$, *y compris le prédicat « être supérieur ou égal à n » que nous sommes en train de définir.*

En voici la définition *Coq* :

```
Definition my_le (n p:nat) :=
  ∀P:nat → Prop, P n →(∀q:nat, P q → P (S q))→ P p.
```

Exercice 6.15 ** Prouver les lemmes suivants, établissant des relations avec le prédicat binaire `le` défini en *Coq*.

```
Lemma my_le_n : ∀n:nat, my_le n n.

Lemma my_le_S : ∀n p:nat, my_le n p → my_le n (S p).

Lemma my_le_le : ∀n p:nat, my_le n p → n ≤ p.
```

À ce stade, il paraît clair que notre définition imprédicative est correcte. Pour en être certain, il faudra montrer la réciproque de `my_le_le`. Cette preuve utilise les techniques de preuves par récurrence abordées au chapitre 9. La définition inductive de `le` se trouve page 242. La preuve de la réciproque de `my_le_le` se trouve page 260.

Comment interpréter les définitions imprédicatives

Les exemples précédents peuvent nous éclairer sur la nature des définitions imprédicatives. Reprenons quelques-uns de ces exemples, en paraphrasant les définitions associées.

– “ `my_or` P Q ” sert à prouver n'importe quelle proposition R à partir de preuves de $P{\rightarrow}R$ et de $Q{\rightarrow}R$,
– “ $x = y$ ” sert à prouver n'importe quelle proposition “ P y ” à partir d'une preuve de “ P x ”,
– “ `my_le` n p ” sert à prouver n'importe quelle proposition “ P p ” à partir d'une preuve de “ P n ” et d'une preuve de la proposition “ $\forall q$`:nat,` P q $\rightarrow P$ $(S$ $q)$ ”.

Cette lecture montre que ces prédicats ou propositions sont définis *par leur utilisation* : prouver R à partir d'un terme t de type “ `my_or` P Q ” se fait en appliquant t à R et à deux termes de preuves $t_P : P{\rightarrow}R$ et $t_Q : Q{\rightarrow}R$.

Cette approche est la même que la « programmation par passage de continuations » (ou *CPS* : Continuation Passing Style) où les fonctions prennent comme argument l'utilisation qu'on veut faire du résultat de leur application (voir O. Danvy [34] et M. Wand [87]).

Il faut noter que la présentation dans ce livre des définitions imprédicatives répond essentiellement à un objectif pédagogique. En effet *Coq* utilise les Constructions Inductives pour coder ce genre de prédicats de façon bien plus efficace (voir chapitre 9).

Chapitre 7

Structures de données inductives

La définition de types inductifs en *Gallina* étend les différentes notions de définitions de types fournies dans les langages de programmation. On peut les comparer aux définitions de types récursifs dans les langages fonctionnels : *ML*, *OCAML*, *Haskell*. Mais la possibilité de mélanger types récursifs et produits dépendants rend les types inductifs de *Gallina* beaucoup plus précis et expressifs, au point que l'on peut aussi les utiliser pour décrire la programmation logique pure, c'est-à-dire le noyau primitif de PROLOG.

À chaque type de données inductif correspond une structure de calcul, basée sur le filtrage et la récursion. Ces structures de calcul sont le noyau de la programmation récursive en *Gallina* et nous en décrirons les fondements dans cette partie.

7.1 Types sans récursion

Avant d'aborder la récursivité proprement dite, considérons les types de données sans récursion, intéressants par leur capacité à décrire des données regroupées dans des multiplets avec variantes. Les outils de calcul associés permettent de construire de tels multiplets et d'accéder séparément aux données incluses.

Pour les programmeurs habitués au langage Pascal, ces types sans récursion permettent de représenter les enregistrements fournis par la construction `record` et les accesseurs aux champs des enregistrements (notation `a.b`). Pour les programmeurs habitués au langage C, ils permettent de représenter les types construits à l'aide de `struct` et `union` et l'accès aux champs d'une telle structure (également fourni par la notation `a.b`).

7.1.1 Types énumérés

Les types inductifs les plus simples sont les types énumérés, utilisés pour décrire des ensembles finis. L'exemple le plus fréquemment utilisé d'un tel ensemble fini est celui des valeurs booléennes, qui contient seulement deux éléments. On peut retrouver la définition de ce type en utilisant la commande `Print`.

```
Print bool.
```
Inductive bool : Set := true : bool | false : bool

Pour nos exemples, nous travaillerons plutôt avec un autre ensemble fini, l'ensemble des mois de l'année. Cet ensemble a douze éléments distincts et leurs noms en anglais sont bien connus.

La déclaration à fournir à *Coq* a la forme suivante [1] :

```
Inductive month : Set :=
  January : month | February : month | March : month
| April : month   | May : month      | June : month
| July : month    | August : month   | September : month
| October : month | November : month | December : month.
```

Dans le cas présent, où les constructeurs ont exactement le type inductif défini, la notation suivante est également possible :

```
Inductive month : Set :=
| January   | February | March    | April
| May       | June     | July     | August
| September | October  | November | December.
```

Cette déclaration introduit simultanément un type `month` dans la sorte `Set` et douze éléments de ce type, `January`, `February`, etc. Ces douze éléments sont appelés les *constructeurs* du type inductif . Le système *Gallina* ajoute automatiquement des théorèmes ou fonctions qui vont nous permettre de raisonner et de faire des calculs sur les données de ce type. Le premier de ces théorèmes a pour nom `month_ind` et nous l'appellerons également le *principe de récurrence* associé à la définition inductive. On peut vérifier son énoncé par la commande suivante :

```
Check month_ind.
```
month_ind :
$\forall$ *P:month* $\rightarrow$ *Prop,*
P January $\rightarrow$ *P February* $\rightarrow$ *P March* $\rightarrow$ *P April* $\rightarrow$
P May $\rightarrow$ *P June* $\rightarrow$ *P July* $\rightarrow$ *P August* $\rightarrow$
P September $\rightarrow$ *P October* $\rightarrow$ *P November* $\rightarrow$ *P December* $\rightarrow$
$\forall$ *m:month, P m*

1. La barre verticale '`|`' située avant le premier élément de l'énumération est facultative

L'énoncé de ce théorème est construit de façon simple à partir de la définition du type inductif. Dans ce théorème, on quantifie d'abord sur un prédicat `P` quelconque sur les mois, puis on construit une succession d'implications ayant pour prémisses successives le prédicat `P` appliqué à chacun des mois, et l'on a une conclusion indiquant que la propriété est vérifiée pour tous les mois. Ce théorème permet de démontrer une propriété pour tous les mois en réunissant les douze démonstrations élémentaires pour chaque mois.

Outre `month_ind`, le système *Coq* engendre une fonction appelée `month_rec` dont le type est similaire à l'énoncé de `month_ind`, sauf que la propriété sur laquelle on quantifie ne prend plus ses valeurs dans `Prop`, mais dans `Set` :

```
Check month_rec.
```
month_rec :
∀ P:month → Set,
P January → P February → P March → P April →
P May → P June → P July → P August →
P September → P October → P November → P December →
∀ m:month, P m

La fonction `month_rec` permet de définir une fonction par cas sur le type `month`. Cette fonction est associée à des règles de réduction sur lesquelles nous reviendrons dans la section 15.1.4 et nous donnerons un exemple d'utilisation dans la section 7.1.4.

La troisième fonction, `month_rect` est encore similaire, mais cette fois-ci le type final de la proposition `P` est `Type`. Cette fonction est plus puissante que les deux autres, qui peuvent être dérivées en la spécialisant pour les sortes `Prop` `Set`, grâce aux règles de conversion des section 3.5.2 et 4.1.1. Elle peut être aussi utilisée pour montrer que deux mois sont différents, nous l'utiliserons dans la section 7.2.3.

Exercice 7.1 Définir un type inductif représentant les saisons, puis utiliser la fonction `month_rec` pour définir une fonction qui associe à chaque mois la saison qui contient la majeure partie de ses jours.

Exercice 7.2 Quel est le type des constantes `bool_ind`, `bool_rec` et `bool_rect` engendrés par *Coq* pour le type `bool` ?

7.1.2 Raisonnements et calculs simples

Les constructions `..._ind` et `..._rec` engendrées au moment de la définition d'un type inductif sont utilisables pour effectuer des raisonnements et des calculs sur les données de ce type.

Par exemple, `month_ind` peut s'utiliser pour montrer que toute donnée de type `month` est forcément l'un des mois connus. En voici une démonstration possible :

```
Theorem month_equal :
∀m:month,
 m=January ∨ m=February ∨ m=March ∨ m=April ∨ m=May ∨ m=June ∨
 m=July ∨ m=August ∨  m=September ∨ m=October ∨ m=November ∨
 m=December.
Proof.
 induction m; auto 12.
Qed.
```

Afin de mieux comprendre les mécanismes de cette preuve, nous allons présenter une démonstration plus manuelle, qui produit exactement le même terme de preuve. Nous allons appliquer le théorème `month_ind`. Il est donc nécessaire de montrer que le but est effectivement de la forme “ P `m` ” pour une proposition P bien choisie. La tactique `pattern` (voir section 6.3.3) est fournie par *Coq* exactement pour cette opération :

```
Reset month_equal.
Theorem month_equal :
∀m:month,
 m=January ∨ m=February ∨ m=March ∨ m=April ∨
 m=May ∨ m=June ∨ m=July ∨ m=August ∨
 m=September ∨ m=October ∨ m=November ∨ m=December.
Proof.
 intro m; pattern m.
  ...
  ============================
  (fun m0 : month ⇒
   m0=January ∨ m0=February ∨ m0=March ∨ m0=April ∨
   m0=May ∨ m0=June ∨ m0=July ∨ m0=August ∨
   m0=September ∨ m0=October ∨ m0=November ∨ m0=December)
   m
```

Ceci fait bien apparaître que le but est une propriété de `m`, en précisant que cette propriété est décrite par l'abstraction ci-dessous :

```
    fun m0:month ⇒ m0=January ∨ ... ∨ m0=December
```

Nous pouvons maintenant appliquer le théorème `month_ind` et le filtrage effectué par la tactique `apply` permet de déterminer les valeurs pour les deux variables quantifiées universellement apparaissant dans la conclusion de ce théorème : `P` et `m`.

```
apply month_ind.
12 subgoals

  m : month
  ============================
```

January=*January* ∨*January*=*February* ∨*January*=*March* ∨
January=*April* ∨*January*=*May* ∨*January*=*June* ∨
January=*July* ∨ *January*=*August* ∨*January*=*September* ∨
January=*October* ∨*January*=*November*∨*January*=*December*
...

Les onze autres buts sont de la même forme, et ne diffèrent que par le mois qui apparaît en membre gauche des égalités. Chacun de ces buts se traite comme dans l'exercice 6.5 page 151. Plutôt que de résoudre un par un chacun de ces sous-buts, par une tactique allant de `auto` à “ `auto 12` ”, nous pouvons composer la tactique “ `intro m; pattern m` ” avec un appel à “ `auto 12` ”, suffisant pour résoudre les 12 sous-buts.

La preuve construite par la simple commande “ `induction m;auto 12` ” est la même que celle construite par notre approche manuelle. La tactique “ `induction m` ” effectue pour nous le travail d'introduire `m`, d'utiliser `pattern`, puis d'appliquer le théorème `month_ind`. La tactique “ `auto 12` ” se charge d'appeler les tactiques `left`, `right` et “ `apply refl_equal` ” sur les douze buts engendrés. Le lecteur intrigué pourra effectuer une preuve totalement manuelle (sans `auto` ni `induction`) et utiliser la commande `Print` pour comparer les termes de preuves associés.

Exercice 7.3 Prouver de deux façons le théorème suivant :

`bool_equal : ∀b:bool, b = true ∨ b = false`

1. En fournissant directement un terme de preuve (utilisant les théorèmes `or_introl` et `or_intror` introduits dans la section 5.3.5 et le théorème `refl_equal` introduit dans la section 5.3.4).
2. En utilisant les tactiques `pattern`, `apply`, `left`, `right`, et `reflexivity`.

7.1.3 La tactique `elim`

La tactique `elim` établit le lien entre un type inductif et le principe de récurrence associé à ce type inductif en faisant systématiquement appel à ce principe de récurrence. Cette tactique a un comportement simple et pratique pour l'utilisateur mais assez complexe à décrire car plusieurs mécanismes coopèrent pour la rendre conviviale.

Dans son comportement basique, la tactique `elim` prend un seul argument d'un type inductif `T`. Elle vérifie alors la sorte du but et choisit un principe de récurrence en fonction de cette sorte. Si la sorte du but est `Prop`, alors le principe choisi est le théorème `T_ind`, pour un but dans la sorte `Set`, c'est le principe `T_rec` qui est utilisé, enfin pour un but dans la sorte `Type` c'est le principe `T_rect` qui est utilisé.

Tous ces principes ont la même forme. Ils contiennent une quantification universelle sur une variable `P` de type `T→` s où s est une sorte et leur énoncé termine par la formule “ `∀x:T,P x` ”. La tactique “ `elim` t ” procède alors en faisant apparaître en quoi le but est une fonction du terme t, puis elle applique le principe de récurrence choisi.

Déterminer en quoi un but arbitraire est une propriété d'une certaine valeur est effectué simplement par la tactique `pattern`. Lorsque le principe de récurrence à utiliser est `T_ind`, la tactique “ `elim` t ” est en première approximation équivalente à la tactique suivante :

```
pattern t; apply T_ind.
```

On voit que la tactique `elim` effectue une première opération pour le confort de l'utilisateur : elle détermine le principe de récurrence qui va être utilisé. En fait l'utilisateur peut choisir d'imposer un autre principe de récurrence, en utilisant la directive `using`. Grâce à cette directive `using`, la tactique `elim` peut aussi être utilisée si le type `T` n'est pas un type inductif, du moment que le théorème donné en argument a la forme d'un principe de récurrence. Nous verrons en section 15.1.3 quelle est cette forme en général.

La tactique `elim` peut également être utilisée avec un argument qui est une fonction. Dans ce cas, c'est le type final de la fonction qui est utilisé (le type final du terme est déterminé comme pour la tactique `apply`, voir section 4.2.2). Lorsque le type de la fonction est un type non dépendant, les arguments à fournir à la fonction réapparaissent comme des buts supplémentaires à démontrer par l'utilisateur. Lorsque le type de fonction est un type dépendant, l'utilisateur peut préciser l'argument correspondant à l'aide de la directive `with`, comme pour la tactique `apply` (voir section 6.1.3).

La tactique `elim` peut avoir un comportement plus complexe que nous étudierons dans la section 9.5, lorsque nous aurons présenté toute la puissance des types inductifs.

La tactique `induction` est construite au dessus de la tactique `elim`. Lorsque `v` n'est pas une variable du contexte, “ `induction v` ” est similaire à la tactique “ `intros until v;elim v` ”. En fait, `induction` utilise les noms de variables apparaissant dans le but et pourra éventuellement effectuer plusieurs introductions avant de faire appel à la tactique `elim`. Lorsque `v` est une variable du contexte, la tactique “ `induction v` ” a un comportement bien plus complexe (voir section 8.1.1).

7.1.4 Construction de filtrage

La construction de filtrage permet de décrire des fonctions qui effectuent un traitement par cas sur la valeur d'une expression dont le type est inductif. Ceci se fait à l'aide d'une construction `match` dont la syntaxe la plus simple est présentée ci-dessous ; on considère un terme t dont le type inductif T est défini par les constructeurs c_1, $c_2, \ldots,$ c_l :

```
match t with
  c1 => e1
| c2 => e2
  ...
```

```
| c_l ⇒ e_l
end
```

Cette construction vaut e_1 si t vaut le constructeur c_1 de T, e_2 si t vaut le constructeur c_2, etc.

Filtrage sur les valeurs booléennes

Le filtrage sur le type `bool` fournit simplement un moyen d'écrire des expressions conditionnelles. En effet, une construction par cas sur `bool` a la forme suivante :

```
match t with true ⇒ e1 | false ⇒ e2 end
```

Cette expression prend la valeur de e_1 si t vaut `true` et la valeur de e_2 sinon, exactement ce que l'on attend de l'expression conditionnelle

```
if t then e1 else e2
```

Le système *Coq* considère ces deux écritures comme rigoureusement équivalentes, ainsi que le montre le dialogue ci-dessous :

```
Check (fun b:bool ⇒ match b with true ⇒ 33 | false ⇒ 45 end).
```

fun b:bool ⇒ if b then 33 else 45 : bool →nat

Exemple

À l'aide de filtrages sur les type `month` et `bool`, nous pouvons écrire la fonction qui calcule le nombre de jours d'un mois quelconque (l'argument booléen est associé au caractère bissextile de l'année considérée) :

```
Definition month_length (leap:bool)(m:month) : nat :=
  match m with
  | January ⇒ 31 | February ⇒ if leap then 29 else 28
  | March ⇒ 31   | April ⇒ 30    | May ⇒ 31  | June ⇒ 30
  | July ⇒ 31    | August ⇒ 31   | September ⇒ 30
  | October ⇒ 31 | November ⇒ 30 | December ⇒ 31
  end.
```

Tous les cas d'une construction par filtrage doivent être couverts, sinon le système *Coq* émet un message d'erreur indiquant au moins un des cas non couverts. Foici un exemple de définition incomplète :

```
Definition month_length : bool→month→nat :=
 fun (leap:bool)(m:month) ⇒ match m with January => 31 end.
  ...
```

Error: Non exhaustive pattern-matching: no clause found for pattern February

Le calcul décrit dans la fonction `month_length` peut aussi s'exprimer en utilisant la fonction `month_rec` qui a été engendrée par le système *Coq* à la définition du type `month`. La définition peut se faire de la façon suivante :

```
Definition month_length' (leap:bool) :=
  month_rec (fun m:month ⇒ nat)
  31 (if leap then 29 else 28) 31 30 31 30 31 31 30 31 30 31.
```

Cette écriture peut sembler plus concise, mais la lisibilité en souffre. De plus, l'écriture avec la construction de filtrage permet d'utiliser des clauses par défaut, qui sont couvertes par une simple variable. Ici nous avons choisi d'utiliser le nom `other`, mais *Coq* fournit également la variable anonyme '`_`', avec la convention que plusieurs utilisations de cette variable correspondent à des variables différentes :

```
Definition month_length'' (leap:bool)(m:month) :=
 match m with
 | February ⇒ if leap then 29 else 28
 | April ⇒ 30 | June ⇒ 30 | September ⇒ 30 | November ⇒ 30
 | other ⇒ 31
 end.
```

Évaluation des fonctions

La construction de filtrage est associée à un mécanisme systématique d'évaluation, connu sous le nom de ι-réduction (prononcer « iota-réduction »). Ces règles de conversions sont utilisées systématiquement par le vérificateur de type lorsqu'il s'agit de comparer deux expressions, au même titre que les règles de β-réduction et de δ-réduction que nous avons déjà rencontrées dans la section 3.4.2. Ainsi les règles de ι-réduction assurent les convertibilités décrites dans le tableau ci-dessous :

`month_length` *leap* `September`	30
`month_length false February`	28
`month_length true February`	29

La commande `Eval` est fournie pour tester les fonctions sur certaines valeurs, comme dans l'exemple suivant :

```
Eval compute in (fun leap ⇒ month_length leap November).
```
= fun _:bool => 30 : bool→nat

Lorsque l'on effectue des démonstrations, il arrive régulièrement que l'on fasse apparaître dans les buts des expressions de la forme " $f\ \ c$ " qui pourraient être ι-réduites. Il est alors possible de provoquer la ι-réduction en faisant appel à la tactique `simpl`. Voici un exemple minimal de session utilisant cette tactique :

```
Theorem length_february : month_length false February = 28.
Proof.
 simpl.
  ...
  ============================
   28 = 28
 trivial.
Qed.
```

La tactique `simpl` peut aussi être utilisée pour effectuer l'évaluation des fonctions dans une hypothèse, en utilisant le mot-clef `in` pour préciser le nom de l'hypothèse.

La tactique `simpl` effectue la réduction de toutes les fonctions qui s'y prêtent. Il est possible de limiter la réduction en utilisant plutôt la commande suivante, qui permet de spécifier les réductions à utiliser, tout en restreignant la δ-réduction à un ensemble de constantes précis (ici limité à `month_length`) et en suivant une stratégie paresseuse :

```
lazy iota beta zeta delta [month_length].
```

Il existe aussi une tactique `cbv` qui effectue les conversions en utilisant une stratégie d'appel par valeur.

Exercice 7.4 Écrire la fonction qui associe à chaque mois la saison qui contient le premier jour de ce mois. On utilisera le type des saisons construit à l'exercice 7.1.

Exercice 7.5 Écrire la fonction qui associe la valeur booléenne `true` à chaque mois dont le nombre de jours est pair et `false` aux autres.

Exercice 7.6 Définir les fonctions `bool_xor`, `bool_and`, `bool_or`, `bool_eq`, de type `bool` $\rightarrow$ `bool` $\rightarrow$ `bool` et la fonction `bool_not` de type `bool` $\rightarrow$ `bool`. Démontrer les théorèmes suivants :

```
∀b1 b2 :bool, bool_xor b1 b2 = bool_not (bool_eq b1 b2)

∀b1 b2 :bool, bool_not (bool_and b1 b2) =
              bool_or (bool_not b1) (bool_not b2)
```

```
∀b:bool, bool_not (bool_not b) = b
```

```
∀b:bool, bool_or b (bool_not b) = true
```

```
∀b1 b2:bool, bool_eq b1 b2 = true → b1 = b2
```

```
∀b1 b2:bool, b1 = b2 → bool_eq b1 b2 = true
```

```
∀b1 b2:bool, bool_not (bool_or b1 b2) =
             bool_and (bool_not b1) (bool_not b2)
```

```
∀b1 b2 b3:bool, bool_or (bool_and b1 b3) (bool_and b2 b3)=
                bool_and (bool_or b1 b2) b3
```

7.1.5 Types enregistrements

Les types inductifs peuvent aussi naturellement être utilisés pour représenter des agrégations de données, comme des *multiplets* en mathématiques ou des enregistrements en programmation. Dans ce cas, on utilise des définitions inductives à un seul constructeur, où ce constructeur a le type d'une fonction qui prend autant d'arguments qu'il y a de champs dans l'enregistrement. L'explication intuitive est que le constructeur est une fonction qui retourne un enregistrement si on lui donne les valeurs de tous les champs.

Par exemple, un point dans le plan est souvent décrit comme un couple de coordonnées. Si l'on s'intéresse à l'ensemble constitué de tous les points à coordonnées entières, le type correspondant pourra donc être décrit de la façon suivante :

```
Inductive plane : Set := point : Z→Z→plane.
```

Le principe de récurrence associé à ce type inductif a la forme suivante :

plane_ind
: ∀ P: plane → Prop,
(∀ z z0: Z, P (point z z0))→ ∀ p: plane, P p

Ce principe de récurrence indique que, pour vérifier une propriété sur tous les éléments du plan, il suffit de la vérifier sur tous les objets obtenus en appliquant le constructeur `point` à des coordonnées quelconques.

Lorsque l'on dispose d'un point, il peut être nécessaire de faire des calculs sur ses coordonnées. C'est encore la construction `match` qui va permettre ce genre d'opération.

Par exemple, on pourra trouver l'abscisse d'un point (sa première coordonnée) de la façon suivante :

```
Definition abscissa (p:plane) : Z :=
  match p with point x y ⇒ x end.
```

Les types enregistrements peuvent également être définis sous une forme qui donne explicitement l'interprétation que l'on donnera à chaque champ, à l'aide de la commande `Record`.

```
Reset plane.

Record plane : Set := point {abscissa : Z; ordinate : Z}.
```

Au traitement de cette définition, le système *Coq* engendre la définition inductive de `plane` donnée plus haut ainsi que les définitions des accesseurs `abscissa` et `ordinate`. Ces définitions peuvent être consultées à l'aide de `Print` :

```
Print plane.
Inductive plane  : Set :=  point : Z→Z→plane

For point : Argument scopes are [Z_scope Z_scope]
Print abscissa.
abscissa =
    fun p:plane ⇒ let (abscissa, _) := p in abscissa
    : plane→Z
```

La notation “ `let (`v_1`,..., `v_n`) = `t` in `t' ” utilisée ici est une alternative pour la construction de filtrage “ `match` t' `with` $c\ v_1\ \ldots v_n \Rightarrow t$ `end` ”, si c est l'unique constructeur du type (inductif) de t'.

De plus, la notation “ p`.(abscissa)` ” est une alternative pascalienne pour l'application “ `abscissa` p ”.

Exercice 7.7 Donner le type de `plane_rec`.

Exercice 7.8 Définir une fonction calculant la distance « de Manhattan » pour les points de `plane` (la distance de Manhattan est la somme des valeurs absolues des différences des coordonnées).

Remarque

En *Gallina*, il n'est pas possible de modifier la valeur des champs d'un enregistrement. En effet, la valeur constante de ces champs est fixée à la construction.

7.1.6 Types enregistrements avec variantes

Il est possible de mélanger dans une même définition inductive les caractéristiques des types énumérés et des types enregistrement. Ceci permet alors de décrire des données qui peuvent prendre plusieurs formes différentes.

Par exemple, on peut considérer un type de véhicule contenant des bicyclettes et des engins à moteurs. Pour les bicyclettes on précise le nombre de places (on peut envisager un tandem), et pour les engins à moteur on précise le nombre de places et le nombre de roues.

```
Inductive vehicle : Set :=
  bicycle : nat→vehicle | motorized : nat→nat→vehicle.
```

Le principe de récurrence associé à ce type inductif a la forme suivante :

```
Check vehicle_ind.
```

vehicle_ind
: ∀ P:vehicle → Prop,
(∀ n:nat, P(bicycle n))→
(∀ n n0:nat, P(motorized n n0))→
∀ v:vehicle, P v

Ce principe de récurrence indique que pour vérifier une propriété sur tous les véhicules il faut vérifier cette propriété pour toutes les bicyclettes, quel que soit leur nombre de selles, et pour tous les véhicules motorisés, quels que soient le nombre de roues et le nombre de sièges. Ici, il faut donc encore vérifier tous les cas correspondant aux différents constructeurs, et pour chaque constructeur tous les cas correspondant aux différentes valeurs possibles des paramètres.

Grâce à la construction `match`, on pourra construire des fonctions qui effectuent des calculs différents en fonction du constructeur présent dans la donnée.

Par exemple, pour les véhicules, nous obtiendrons le nombre de roues et le nombre de places de la façon suivante :

```
Definition nb_wheels (v:vehicle) : nat :=
  match v with
  | bicycle x ⇒ 2
  | motorized x n ⇒ n
  end.

Definition nb_seats (v:vehicle) : nat :=
  match v with
  | bicycle x ⇒ x
  | motorized x _ ⇒ x
  end.
```

Bien sûr, le choix d'interpréter le premier argument de `motorized` comme le nombre de places et le second comme le nombre de roues est arbitraire et n'est pas indiqué tant que les fonctions `nb_wheels` et `nb_seats` n'ont pas été construites.

Exercice 7.9 Déterminer le type de `vehicle_rec`. Utiliser cette constante pour définir sans utiliser la construction de filtrage la fonction `nb_seats`.

7.2 Preuves par cas

7.2.1 La tactique `case`

Lorsque l'on effectue des preuves sur des fonctions contenant des constructions de filtrage, il faut effectuer les mêmes traitements par cas que ceux effectués par ces constructions de filtrage, pour montrer que la propriété recherchée est bien satisfaite dans tous les cas. La tactique `elim` permet déjà d'effectuer ce genre de raisonnement, mais le système *Coq* fournit une tactique plus primitive pour ce besoin, la tactique `case`.

La tactique `case` prend en argument un terme t qui doit appartenir à un type inductif. L'effet de cette tactique est de remplacer toutes les instances de t dans le but par les différents cas possibles que peut prendre un élément de ce type, conduisant ainsi l'utilisateur à faire une preuve différente pour chaque cas possible.

Par exemple, on peut démontrer que le nombre de jours dans un mois est toujours supérieur ou égal à vingt-huit. Ceci se fait en étudiant les treize cas correspondant à chaque mois et à la possibilité d'une année bissextile.

```
Theorem at_least_28 :
 ∀(leap:bool)(m:month), 28 ≤ month_length leap m.
Proof.
```

Ce théorème se montre très rapidement en utilisant les tactiques suivantes :

```
 intros leap m; case m; simpl; auto with arith.
 case leap; simpl; auto with arith.
Qed.
```

Il peut être intéressant de reprendre cette preuve de façon plus manuelle, de façon à comprendre quels sous-buts sont masqués par l'utilisation de tactiques composées et d'automatismes. Renonçons donc provisoirement à ces outils.

```
Reset at_least_28.
Theorem at_least_28:
   ∀(leap:bool)(m:month), 28 ≤ month_length leap m.
Proof.
 intros leap m; case m.
  ...
  leap : bool
  m : month
  ============================
   28 ≤ month_length leap January

  ...
```

À ce point de la démonstration il y a douze nouveaux buts : ce sont exactement les mêmes buts que si nous avions utilisé la tactique `elim`. La tactique `simpl` nous permet de simplifier le but courant, en évaluant l'application de la fonction `month_length` :

```
 simpl.
  ...
  m : month
```

```
============================
  28 ≤ 31
```

Ce but se résout en appliquant les deux théorèmes théorèmes `le_n` et `le_S` que nous avons déjà rencontrés en section 5.2.1.1, page 110.

Nous avons indiqué dans la section 3.2.3 que la notation “ `28` ” représentait en fait l’expression

```
S (S ... (S O) ... )
```

où le constructeur `S` est répété vingt-huit fois. Ce but peut donc se résoudre manuellement avec la tactique composée suivante :

```
apply le_S; apply le_S; apply le_S; apply le_n.
```

En fait la même preuve est celle construite par la tactique automatique “`auto with arith`”. Pour les autres buts la démonstration se fait de manière analogue, mais il faut ajouter un traitement par cas pour le mois de février qui tienne compte des années bissextiles.

Imprimons le terme du Calcul des Constructions Inductives, preuve du théorème `at_least_28`. Ce terme contient deux constructions de filtrage créées par l’application de la tactique `case` (le plus interne de ces filtrages se présente sous la forme d’une expression conditionnelle.)

```
Print at_least_28.
at_least_28 =
fun (leap:bool)(m:month) ⇒
  match m as m0 return (28 ≤ month_length leap m0) with
  | January ⇒ le_S 28 30 (le_S 28 29 (le_S 28 28 (le_n 28)))
  | February ⇒
      if leap as b return  (28 ≤ (if b then 29 else 28))
      then le_S 28 28 (le_n 28)
      else le_n 28
  | March ⇒ le_S 28 30 (le_S 28 29 (le_S 28 28 (le_n 28)))
  ...
  | December ⇒ le_S 28 30 (le_S 28 29 (le_S 28 28 (le_n 28)))
  end
     : ∀ (leap:bool)(m:month), 28 ≤ month_length leap m
```

Dans l’expression ci-dessus, nous pouvons observer que l’en-tête des constructions de filtrage créées par l’application de la tactique case est plus complexe que dans les exemples précédents ; considérons l’en tête ci-dessous :

```
match m as m0 return 28 ≤ month_length leap m0 with
```

Afin de comprendre les informations contenues dans cet en-tête, étudions le type de chacune des branches de ce filtrage. Nous trouvons des propositions bien différentes, parmi : “ `28 ≤ 31` ”, “ `28 ≤ (if leap then 29 else 28)` ” et “ `28 ≤ 30` ”.

Mais considérons la fonction suivante :

```
fun m0:month ⇒ 28 ≤ month_length leap m0
```

Si nous appliquons cette fonction aux divers constructeurs de `month`, nous obtenons des propositions convertibles avec le type de la branche droite correspondante de l'expression de filtrage. Par exemple, nous obtenons avec `January` une proposition convertible avec l'inégalité “ `28 ≤ 31` ”.

Sous cette condition, la construction de filtrage est bien typée. On parle alors de filtrage *dépendant*, un concept absent des langages de programmation fonctionnels conventionnels. Ce concept est étudié plus en détail en section 15.1.4.

Il est possible de guider la tactique `case` pour construire de façon interactive une expression de filtrage dépendant. Ceci se fait à l'aide de la tactique `pattern`. Nous verrons un exemple de collaboration entre la tactique `case` et la tactique `pattern` dans la section 7.2.7.

Nous laissons au lecteur le soin d'étudier le second filtrage dépendant du terme de preuve ci-dessus. Celui-ci utilise la syntaxe

`if` t `as` v `return` T `then` e_1 `else` e_2

équivalente à la construction :

`match` t `as` v `return` T `with true ⇒` e_1 `| false ⇒` e_2 `end`

7.2.2 Égalités contradictoires

Il existe de nombreux théorèmes dont les prémisses contiennent une égalité. Lorsqu'un tel théorème fait l'objet d'une preuve par cas, l'utilisation de la tactique `case` peut faire alors apparaître des buts dont certaines hypothèses sont de la forme

$$c_1 = c_2$$

où c_1 et c_2 sont deux constructeurs distincts d'un même type inductif. Cette égalité est alors contradictoire. Le système *Coq* fournit une tactique `discriminate` qui permet de conclure une preuve dès qu'une telle égalité contradictoire apparaît dans les hypothèses d'un but.

Pour décrire le fonctionnement de cette tactique sur un exemple, nous allons définir une fonction `next_month` qui associe à chaque mois le mois qui le suit dans le calendrier et démontrer un théorème simple sur cette fonction :

```
Definition next_month (m:month) :=
  match m with
    January ⇒ February  | February ⇒ March | March ⇒ April
  | April ⇒ May         | May ⇒ June       | June ⇒ July
  | July ⇒ August       | August ⇒ September
  | September ⇒ October | October ⇒ November
  | November ⇒ December | December ⇒ January
  end.

Theorem next_august_then_july :
```

```
 ∀m:month, next_month m = August → m = July.
Proof.
 intros m; case m; simpl; intros Hnext_eq.
```

La tactique `case` fait apparaître douze buts, dont seulement le septième semble aisé à démontrer :

```
Show 7.
  ...
  Hnext_eq : August = August
  ============================
   July = July
```

Ce but se résout avec la tactique `reflexivity` . Les autres se ressemblent tous. Étudions seulement le premier d'entre eux :

```
Show 1.
  ...
  Hnext_eq : February = August
  ============================
 January = July
```

La tactique “ `discriminate Hnext_eq` ” permet d'exploiter le caractère contradictoire de l'égalité `Hnext_eq`, et résout immédiatement ce but. Les dix autres buts semblables se résolvent de la même façon.

Pour utiliser la similitude entre tous ces buts, nous pourrons condenser toute la démonstration dans les quelques lignes suivantes :

```
Restart.
 intros m; case m; simpl; intros hnext_eq;
 discriminate Hnext_eq || reflexivity.
Qed.
```

7.2.3 ** Les dessous de `discriminate`

Nous allons maintenant tâcher de comprendre comment fonctionne la tactique `discriminate` en étudiant une démonstration manuelle de l'inégalité [2] “ `January ≠ February` ”.

```
Theorem not_January_eq_February : January ≠ February.
Proof.
 unfold not; intros H.
  ...
  H : January = February
```

2. L'écriture “ $t_1 \neq t_2$ ” est une notation pour l'application “ `not (`t_1 `=` t_2`)` ” ; rappelons que le symbole ‘≠’ se tape ‘<>’.

```
============================
  False
```

Une méthode pour résoudre ce type de but consiste à considérer une fonction ϕ de type “ `month` $\rightarrow$ `Prop` ” telle que les propositions “ ϕ `January` ” et `True` soient convertibles, de même que “ ϕ `February` ” et `False`. De cette façon, l'égalité `H` permettra d'associer à toute preuve de `True` une preuve de `False` et donc de résoudre le but courant. Nous pouvons prendre pour ϕ la fonction suivante :

```
fun m ⇒ match m with January ⇒ True | _ ⇒ False end
```

Cette fonction peut également s'exprimer directement à l'aide de la constante `month_rect` (laissé en exercice.)

La tactique “ `change` B ” permet de remplacer un but d'énoncé A par B si les termes A et B sont convertibles. Cette tactique s'applique donc à notre situation :

```
change ((fun m:month ⇒
          match m with | January ⇒ True | _ ⇒ False end)
        February).
```

À ce point, nous pouvons remplacer `February` par `January`, en effectuant une réécriture à l'aide de l'hypothèse `H` :

```
 rewrite <- H.
  ...
  H : January=February
  ============================
   True

 trivial.
Qed.
```

Exercice 7.10 * Construire manuellement une preuve de “ `true` $\neq$ `false` ”.

Exercice 7.11 Pour le type `vehicle`, montrer que les bicyclettes n'ont pas de moteur.

7.2.4 Constructeurs injectifs

Une autre interaction entre les traitements par cas et les égalités apparaît lorsque l'on dispose d'une égalité entre deux objets obtenus par le même constructeur, mais que l'on veut utiliser le fait qu'alors leurs composantes sont égales. Ceci correspond au fait que l'on dispose d'une égalité de la forme

$$(c\ x_1 \cdots x_k) = (c\ y_1 \cdots y_k)$$

et que l'on veut en déduire les égalités $x_1 = y_1$, ..., $x_k = y_k$.

Le système *Coq* fournit une tactique pour ce cas de figure. Elle est appelée `injection`, car elle permet de montrer simplement que les constructeurs d'un type inductif sont des fonctions injectives.

Pour illustrer l'utilisation de cette tactique nous allons prouver le théorème suivant, où nous utilisons le type `vehicle` que nous avons introduit dans la section 7.1.6, page 175.

```
Theorem bicycle_eq_seats :
 ∀x1 y1:nat, bicycle x1 = bicycle y1 → x1 = y1.
Proof.
 intros x1 y1 H.
  ...
  H : bicycle x1 = bicycle y1
  ============================
   x1=y1
```

Dans cette situation, on utilise habituellement la tactique injection appliquée à l'hypothèse `H`.

```
injection H.
  ...
  x1 : nat
  y1 : nat
  H : bicycle x1 = bicycle y1
  ============================
   x1=y1 → x1=y1
```

Cec crée exactement l'égalité dont nous avons besoin, la démonstration se termine aisément.

7.2.5 ** Les dessous d'`injection`

Ici encore, nous allons tâcher de comprendre comment fonctionne cette tactique `injection`. Il suffit de faire apparaître dans le but la fonction qui associe à chaque bicyclette le nombre de places qu'elle fournit. La fonction `nb_seats`, définie page 176, fera l'affaire. Si nous reprenons la démonstration depuis le début, nous pouvons l'effectuer de la manière suivante :

```
Reset bicycle_eq_seats.
Theorem bicycle_eq_seats :
 ∀x1 y1:nat, bicycle x1 = bicycle y1 → x1 = y1.
Proof.
 intros x1 y1 H.
 change (nb_seats (bicycle x1) = nb_seats (bicycle y1)).
  ...
  x1 : nat
  y1 : nat
```

```
  H : bicycle x1 = bicycle y1
  ============================
   nb_seats (bicycle x1) = nb_seats (bicycle y1)
```

On peut alors rendre cette égalité prouvable en profitant de l'égalité nommée `H`, par la commande suivante.

```
 rewrite H; trivial.
Qed.
```

Il n'est pas toujours aisé de suivre ce schéma de démonstration, car une fonction analogue à `nb_seats` n'est pas toujours disponible. Dans certains cas, cette fonction doit même être écrite dans le contexte particulier du but à résoudre, en utilisant les ressources fournies par ce but. En ce sens, nous utilisons une stratégie similaire à celle décrite pour l'implémentation de la tactique `discriminate`. Pour illustrer cette difficulté, plaçons nous dans un contexte où existent deux types `A` et `B` quelconques et définissons un type inductif à deux constructeurs dont l'un utilise le type `A` et l'autre le type `B`.

```
Section injection_example.
 Variables A B : Set.

Inductive T : Set := c1 : A→T | c2 : B→T.
```

Nous voulons maintenant démontrer — sans utiliser `injection` — que le constructeur `c2` est injectif :

```
 Theorem inject_c2 : ∀x y:B, c2 x = c2 y → x = y.
 Proof.
  intros x y H.
  ...
  x : B
  y : B
  H : c2 x = c2 y
  ============================
   x = y
```

Afin d'appliquer la méthode utilisée avec le type `vehicle`, nous devons construire une fonction de type " `T → B` ". S'il est facile de préciser que l'image de " `c2` *b* " est *b*, nous devons également définir l'image des éléments de la forme " `c1` *a* ". Or le contexte courant nous fournit une expression de type `B`, par exemple la variable locale `x`.

La fonction suivante — qui n'a de sens que dans le contexte courant — répond parfaitement à nos besoins :

```
   fun (v:T) ⇒ match v with | c1 a ⇒ x | c2 b ⇒ b end.
```

Nous pouvons alors utiliser `change` , puis l'égalité " `H: c2 x = c2 y` " :

```
  change (let phi :=
            fun v:T ⇒ match v with | c1 _ ⇒ x | c2 v' ⇒ v' end
         in phi (c2 x) = phi (c2 y)).
  rewrite H; reflexivity.
 Qed.
End injection_example.
```

Lorsque l'on utilise intensivement des types inductifs dépendants, comme le type `htree` décrit en section 7.5.2, la tactique `injection` peut avoir un comportement bizarre. Il est alors utile de revenir à la simulation manuelle décrite ici (voir exercice 7.45).

7.2.6 Types inductifs et égalités

Les tactiques `discriminate` et `injection` traduisent le fait que les types inductifs jouissent de propriétés très fortes. Deux termes obtenus avec des constructeurs différents ou ayant des composantes distinctes sont forcément différents. Si pour le besoin d'abstraction on cherche à étudier le quotient d'un type de données par rapport à une relation d'équivalence, il n'est pas possible de « représenter » cette relation d'équivalence par l'égalité de Leibniz [3], car ceci mène à des incohérences.

Pour se donner quand même la possibilité de travailler avec des espaces quotients, plusieurs solutions théoriques ont été étudiées. Par exemple, on parle de « sétoïdes » pour désigner des ensembles munis d'une relation d'équivalence et de nombreux travaux ont été effectués pour comprendre le comportement des fonctions sur ces structures (une fonction bien définie doit être « compatible » avec la relation d'équivalence). Néanmoins, le travail avec de telles structures pose rapidement un problème d'efficacité et nous n'approfondirons pas ces notions dans cet ouvrage.

Exercice 7.12 ** Cet exercice, — outre l'utilisation de `discriminate` — a pour but de rendre évident le danger de l'utilisation d'axiomes.

La « théorie » suivante propose une construction des nombres rationnels positifs sous forme de fractions de dénominateur non nul. Un axiome précise que deux rationnels sont égaux dès qu'ils vérifient une condition arithmétique classique.

```
Require Import Arith.

Record RatPlus : Set :=
 mkRat {top:nat; bottom:nat; bottom_condition : bottom ≠ 0}.

Axiom eq_RatPlus :
  ∀r r':RatPlus,
     top r * bottom r' = top r' * bottom r →
```

3. C'est à dire le prédicat `eq` de *Coq*, équivalent à la définition vue en 6.5.3.

```
r = r'.
```

Montrer — en donnant une preuve de `False` — que cette présentation des rationnels est incohérente. *Une fois cet exercice résolu, nous conseillons soit de quitter la session* Coq*, soit de « nettoyer » l'environnement par la commande* “ *`Reset eq_RatPlus`* ”

7.2.7 * Conseils dans l'utilisation de la tactique `case`

La tactique `case` a tendance à perdre de l'information, car elle ne prend pas en compte les propriétés de la formule considérée apparaissant dans le contexte du but. Pour éviter cette situation, trois méthodes peuvent s'appliquer :

– retarder l'usage de `intros` pour laisser les informations pertinentes dans la conclusion du but plutôt que de les introduire dans le contexte et appliquer la tactique `case`,
– utiliser la tactique `generalize` pour transférer toutes les hypothèses pertinentes dans la conclusion et appliquer la tactique `case`,
– faire apparaître une égalité qui sera partiellement remplacée par la tactique `case` et que l'on pourra utiliser pour établir la connexion entre les hypothèses introduites par cette tactique et la formule initiale.

Pour illustrer notre propos, prenons la démonstration d'un théorème artificiel faisant intervenir plusieurs occurrences de la même variable `m1`. Nous commençons par faire une session de preuve qui mène à une impasse.

```
Theorem next_march_shorter :
 ∀(leap:bool)(m1 m2:month), next_month m1 = March →
   month_length leap m1 ≤ month_length leap m2.
Proof.
 intros leap m1 m2 H.
  ...
  m1 : month
  m2 : month
  H : next_month m1 = March
  ============================
 month_length leap m1 ≤ month_length leap m2
```

Appliquons maintenant la tactique “ `case m1` ”, qui engendre 12 buts, dont nous observons ici que le quatrième : dont le premier est le suivant :

```
 case m1.
Show 4.
  ...
  leap : bool
  m1 : month
  m2 : month
  H : next_month m1 = March
   ============================
  month_length leap April ≤ month_length leap m2
```

Comme on le voit, l'instance de `m1` qui apparaissait *dans le but* a été remplacée par `April`, mais pas l'instance qui apparaissait dans l'hypothèse `H`. Cette hypothèse n'est plus d'aucune utilité pour résoudre le but, qui devient impossible à prouver. Les trois méthodes décrites plus haut s'appliquent pour sortir de cette impasse.

Pour la première méthode, nous faisons attention de laisser dans la conclusion du but toutes les informations sur la variable qui fait l'objet d'un traitement par cas, en évitant d'utiliser trop d'appels à la tactique `intros`.

```
Restart.
 intros leap m1 m2.
  ...
  ============================
  next_month m1 = March ->
  month_length leap m1 <= month_length leap m2

 case m1.
Show 4.
  ...
  m2 : month
  ============================
  next_month April = March ->
  month_length leap April <= month_length leap m2
```

Après l'utilisation de la tactique `simpl`, ce but peut se résoudre grâce à la tactique `discriminate`.

La seconde méthode, « transférer les hypothèses pertinentes dans la conclusion », correspond à la session suivante, en considérant que l'on reprend la preuve au début :

```
Restart.
intros leap m1 m2 H.
...
leap : bool
m1 : month
m2 : month
H : next_month m1 = March
 ============================
month_length leap m1 <= month_length leap m2
```

L'utilisation de la tactique `generalize` est la suivante :

```
generalize H.
...
============================
 next_month m1 = March ->
 month_length leap m1 <= month_length leap m2
```

La troisième méthode, « faire apparaître une égalité pour établir la connexion entre les cas introduits et la formule initiale », repose également sur la tactique `generalize`, mais il n'est pas nécessaire de retrouver toutes les hypothèses pertinentes. Pour l'illustrer reprenons la preuve de `next_march_shorter` au début et appliquons la même première commande :

```
Restart.
intros leap m1 m2 H.
```

...
m1 : month
m2 : month
H : next_month m1 = March
============================
month_length leap m1 ≤ month_length leap m2

Nous introduisons l'égalité " `m1 = m1` " en prémisse de la conclusion, en utilisant la tactique `generalize` à laquelle nous fournissons une preuve de cette égalité :

```
generalize (refl_equal m1).
```

...
H : next_month m1 = March
============================
m1=m1→
month_length leap m1 ≤ month_length leap m2

Maintenant nous restreignons l'ensemble des occurrences de `m1` qui seront modifiées par la tactique `case`, en utilisant la tactique `pattern` :

```
pattern m1 at -1.
```

...
============================
(fun m:month ⇒
m1 = m → month_length leap m ≤ month_length leap m2) m1

Nous utilisons un argument négatif pour indiquer les occurrences qui *ne doivent pas* être remplacées par la tactique `case`.

Grâce à cette commande `pattern`, le but a pris la forme :

...
============================
P `m1`

La tactique `case` est ainsi faite, que ce ne sont pas toutes les occurrences de `m1` dans le but qui sont remplacées par les différents cas possibles, mais seulement l'occurrence qui apparaît en argument de P. La commande " `case m1` " va donc engendrer douze buts qui garderont tous une occurrence de `m1` (celle présente à l'intérieur de P.) Examinons l'aspect du premier de ces buts :

```
case m1.
...
H : next_month m1 = March
============================
 m1 = January → month_length leap January ≤ month_length leap m2
```

Ce but est soluble, par exemple avec les commandes suivantes :

```
intro H0; rewrite H0 in H; simpl in H; discriminate H.
```

Ce schéma de raisonnement est tellement fréquent que les auteurs préconisent de définir une tactique pour l'appliquer. En voici la définition :

```
Ltac caseEq f :=
  generalize (refl_equal f); pattern f at -1; case f.
```

Deux choix de conception rendent cette tactique plus générale. Premièrement, nous remarquons que, du fait du caractère implicite du premier argument de `refl_equal`, cette tactique peut s'appliquer à n'importe quel terme dont le type a pour sorte `Type`. Deuxièmement, nous utilisons un argument négatif pour décrire les occurrences de la formule qui *ne doivent pas* être remplacées par l'appel à la tactique `case` qui suit. Ainsi, la tactique fonctionne quel que soit le nombre d'occurrences de la formule dans l'énoncé du but.

Avec cette tactique, la démonstration précédente prend la forme suivante :

```
Abort.
Theorem next_march_shorter :
 ∀(leap:bool)(m1 m2:month),
   next_month m1 = March →
    month_length leap m1 ≤ month_length leap m2.
Proof.
 intros leap m1 m2 H.
 caseEq m1;
  try (intros H0; rewrite H0 in H; simpl in H; discriminate H).
 case leap; case m2; simpl; auto with arith.
Qed.
```

La tactique `pattern` est donc très utile en conjonction avec la tactique `case`, comme nous venons de le montrer, mais aussi avec les tactiques `elim` et `rewrite` (voir en section 6.3.3 page 155).

7.3 Types avec récursion

Les types inductifs sans récursion permettent de décrire toute sorte de données, mais toujours des données dont la taille est connue à l'avance. Il est nécessaire de pouvoir raisonner sur des structures de données dont la taille peut varier, par exemple des tableaux de taille non définie à l'avance.

La récursion fournit une solution extrêmement simple : on exprime que certaines données comportent des fragments qui sont de même nature que ces données elles-mêmes. Par exemple, si l'on considère les arbres binaires, ils peuvent être des feuilles ou des arbres composés. Si ce sont des arbres composés, alors ils contiennent deux fragments qui sont eux aussi des arbres binaires, donc des objets de même nature que l'arbre complet. Un type inductif T est récursif s'il a au moins un constructeur dont l'un des arguments est de type T. Par abus de langage, nous dirons qu'un tel constructeur est *récursif*.

Les types de données que nous étudierons dans cette partie représenteront des ensembles infinis, mais dont chaque élément reste construit en un nombre fini d'étapes. Cette caractéristique nous permettra de disposer d'un moyen de raisonnement et de calcul systématique pour chacun des types abordés : la preuve par récurrence (*proof by induction* en anglais) sera le moyen de raisonnement et la construction de fonctions récursives sera le moyen de calcul. Au fur et à mesure que nous introduirons de nouveaux types récursifs, nous tâcherons d'expliquer comment leur associer un principe de récurrence et nous verrons qu'une fois encore les outils de raisonnement et de calcul sont intimement liés. L'isomorphisme de Curry-Howard va donc nous accompagner dans notre périple autour des types inductifs.

7.3.1 Le type des entiers naturels

L'archétype de la structure de donnée récursive est le type des entiers naturels. C'est aussi celui pour lequel on apprend le plus tôt à faire des raisonnements par récurrence.

La présentation formelle la plus « naturelle » des nombres naturels est inspirée des travaux de Peano. Tout nombre naturel peut être obtenu, soit en prenant le nombre `O`, soit en appliquant la fonction successeur à un nombre déjà construit. Dans le système *Coq*, ceci va s'exprimer par la définition suivante :

```
Print nat.
```
Inductive nat : Set := O : nat | S : nat→nat

Le type des entiers naturels a donc pour habitants `O`, “ `S O` ”, “ `S (S O)` ”, et l'on voit bien qu'il sera impossible de tous les énumérer. Néanmoins, les moyens de construire de nouveaux nombres naturels restent finis, et on pourra embrasser les propriétés de l'ensemble entier par des démonstrations finies grâce au principe de récurrence.

En effet, il n'existe que deux méthodes pour construire un nombre naturel. Soit on prend le nombre `O`, soit on prend un nombre x déjà construit et l'on en construit un nouveau grâce à la fonction `S`. Pour une propriété P donnée, si l'on arrive à démontrer que cette propriété est bien satisfaite par `O` et que si l'on prend un nombre x qui la satisfait alors le nombre construit “ `S` x ” la satisfait également alors, de proche en proche, tout nombre naturel satisfait P. Ceci s'exprime à l'aide d'un principe de récurrence, dont l'énoncé mathématique est le suivant :

$$\forall P, (P(0) \wedge (\forall x. P(x) \Rightarrow P(S(x)))) \Rightarrow \forall x. P(x)$$

On reconnaît le principe de récurrence que l'on enseigne dès l'école dans les classes de mathématiques.

Dans le système *Coq*, cet énoncé s'écrit de la manière suivante :

nat_ind
: ∀ P:nat → Prop,
P 0 → (∀ n:nat, P n → P (S n)) → ∀ n:nat, P n

En pratique le système *Coq* engendre automatiquement le principe de récurrence `nat_ind` à partir de la définition inductive donnée plus haut, ce que nous verrons plus précisément dans la section 15.1.4.

7.3.2 Démonstration par récurrence sur les nombres naturels

Pour effectuer notre première démonstration par récurrence, nous allons utiliser la fonction d'addition de nombres naturels fournie dans les bibliothèques de *Coq*. Cette fonction est décrite plus précisément en section 7.3.3 page 193, mais nous n'en utilisons ici que des propriétés simples, également fournies dans les bibliothèques de *Coq* :

plus_O_n
: ∀ n:nat, 0 + n = n

plus_Sn_m
: ∀ n m:nat, S n + m = S (n + m)

Nous allons maintenant refaire une preuve déjà fournie dans le système *Coq*, celle qui montre que la fonction d'addition est associative. La démonstration ci-dessous reprend le théorème `plus_assoc_r` de *Coq*, énonçant l'associativité de l'addition des entiers naturels :

```
Theorem plus_assoc :
 ∀x y z:nat, (x+y)+z = x+(y+z).
Proof.
```

Nous allons faire cette preuve par récurrence sur `x`. En effet si `x` vaut `0`, deux utilisations de `plus_O_n` doivent permettre de simplifier les deux membres de l'égalité à la même valeur. Si `x` vaut “ `S x'` ”, alors c'est `plus_Sn_m`, utilisé trois fois et une hypothèse de récurrence sur `x'` qui vont permettre de conclure :

```
intros x y z.
elim x.
...
x : nat
y : nat
z : nat
============================
 0+y+z = 0+(y+z)
```

La tactique `elim` est la tactique utilisée pour effectuer une démonstration par récurrence. Cette tactique engendre deux buts, mais nous ne donnons que le premier ici. En appliquant la tactique " `rewrite plus_O_n` ", nous obtenons un nouveau sous-but plus facile à résoudre :

```
rewrite plus_O_n.
...
============================
 y+z = 0+(y+z)
```

Ce but peut maintenant se résoudre à l'aide d'une deuxième réécriture avec le même théorème et de la réflexivité de l'égalité.

```
(* This fragment is not in the book *)
rewrite plus_O_n; trivial.
(* end of fragment *)
```

Le deuxième but a la forme suivante :

```
...
x : nat
y : nat
z : nat
============================
 ∀ n:nat, n+y+z = n+(y+z) → S n + y + z = S n + (y+z)
```

Pour en améliorer la lisibilité, nous pouvons commencer par introduire les différentes variables et hypothèses.

```
intros x' Hrec.
...
x' : nat
Hrec : x'+y+z = x'+(y+z)
============================
 S x' + y + z = S x' + (y+z)
```

Intuitivement, ce but se comprend de la façon suivante : dans le cas où `x` est de la forme " `S x'` " et où l'on suppose (hypothèse de récurrence) que `x'` satisfait déjà la propriété recherchée, il faut démontrer la même propriété pour " `S x'` ".

A l'aide de deux réécritures utilisant le théorème `plus_Sn_m`, nous faisons apparaître dans le but le membre gauche de l'égalité de l'hypothèse de récurrence `Hrec`.

```
rewrite (plus_Sn_m x' y); rewrite (plus_Sn_m (x'+y) z).
...
Hrec : x'+y+z=x'+(y+z)
============================
  S (x'+y+z) = S x' + (y+z)
```

Une réécriture supplémentaire avec le même théorème, suivie d'une réécriture selon l'hypothèse de récurrence `Hrec`, permettent de conclure la preuve :

```
 rewrite plus_Sn_m; rewrite Hrec; trivial.
Qed.
```

7.3.3 Programmation récursive

Pour chaque type de donnée inductif, le système *Coq* engendre également les outils nécessaires pour permettre la programmation de fonctions récursives. La syntaxe est la même pour tous les types inductifs, mais nous allons montrer ici le cas particulier de la programmation de fonctions récursives sur les entiers naturels.

Une fonction se définit généralement en indiquant la valeur qu'elle retourne pour un paramètre donné. En français, on dit « *la fonction qui à x associe l'expression e* ». La variable x est autorisée à apparaître dans l'expression e et c'est ce qui permet d'assurer que la fonction n'est pas constante. Pour une fonction récursive on dit plutôt « *la fonction f qui à x associe l'expression e* », avec la possibilité que f apparaisse aussi dans l'expression e. En d'autres termes, on suppose que la fonction f est déjà *partiellement* définie lorsque l'on essaie de déterminer la valeur qu'elle retourne pour une nouvelle valeur du paramètre. Ce type de définition est intriguant : comment peut-on être sûr que la définition donnée est sensée ? Par exemple « *la fonction f qui à x associe $f\,x$* » n'est pas une définition correcte, car elle « tourne en rond » et ne permet d'obtenir aucune information sur les valeurs retournées par cette fonction.

La solution adoptée dans *Coq* est d'imposer que l'utilisateur détermine les valeurs prises par la fonction dans un ordre précis, qui suit l'ordre dans lequel on construirait les termes des types inductifs. Pour les nombres naturels, on est obligé de construire `O` avant " `S O` ", avant " `S (S O)` " et ainsi de suite. C'est cet ordre de construction des nombres qui est suivi pour la définition d'une fonction récursive f, de sorte que l'on s'autorise à utiliser la valeur " f `O` " pour définir la valeur " f `(S O)` ", la valeur " f `(S O)` " pour définir la valeur " f `(S (S O))` " et ainsi de suite. De manière générale, on s'autorise à utiliser la valeur " f n " pour définir la valeur " f `(S` n`)` ".

En pratique, ce procédé de construction est fourni dans le système *Coq* par la commande `Fixpoint` qui prend la forme suivante dans le cas des nombres naturels :

`Fixpoint` f `(n:nat) :` T `:=` *expr*.

Dans cette définition, f est le nom de la fonction que l'on est en train de définir, `n` est le nom de l'argument sur lequel la récursion s'organise, t est le type retourné, et *expr* indique comment la valeur de " f `n` " est déterminée.

La définition récursive suivante décrit la fonction de type `nat`$\rightarrow$`nat` qui retourne le double de son argument. En effet, le double de 0 est 0, et le double de $n + 1$ est égal au double de n augmenté de 2.

```
Fixpoint mult2 (n:nat) : nat :=
   match n with
     0 ⇒ 0
   | S p ⇒ S (S (mult2 p))
   end.
```

Cette fonction fait bien apparaître un appel récursif dans la sous-expression “ `mult2 p` ”. Par l'intermédiaire du traitement par cas sur `n`, cet appel récursif est utilisé lorsque l'on veut déterminer la valeur de `mult2` sur l'argument “ `S p` ”.

La construction de fonctions récursives s'accompagne de l'ajout de règles de réduction, que l'on regroupe également dans la ι-réduction, que nous avons déjà vue à l'œuvre dans la section 7.1.4. Ces règles de réduction peuvent se déclencher dès qu'une fonction récursive est appliquée à une expression dont la racine est un constructeur. Le tableau suivant fait apparaître plusieurs couples d'expressions convertibles :

`mult2 0`	`0`
`mult2 (S (S 0))`	`S (S (S (S 0)))`
`mult2 (S k)`	`S (S (mult2 k))`

En pratique, les fonctions récursives définies avec `Fixpoint` contiennent un filtrage sur l'argument de la fonction, de sorte que l'on est amené à donner d'abord la valeur de la fonction lorsque cet argument est `0`, puis la valeur de la fonction lorsque l'argument est de la forme “ `S p` ” avec la possibilité d'utiliser la valeur de la même fonction en `p`. La description de la fonction récursive est donc organisée suivant la *structure* du type inductif : il y a deux constructeurs dans le type inductif, donc on retrouve deux cas ; le deuxième constructeur a un argument dans le même type inductif, on peut donc utiliser des appels récursifs sur cet argument. On parle donc de *récursion structurelle*. Les fonctions récursives ne sont pas limitées aux fonctions à un seul argument. En effet le type T dans la commande

`Fixpoint` f `(n:nat) :` T `:=` ***expr***

peut lui-même être un type de fonction, de sorte que la fonction f prend d'autres arguments que celui sur lequel la récursion s'organise. Pour distinguer cet argument, nous l'appellerons l'*argument principal*. La syntaxe de la commande `Fixpoint` permet de préciser l'argument principal par la directive `{struct` a_i`}` dans la syntaxe suivante :

`Fixpoint f (`a_1`:`T_1`)... (`a_p`:`T_p`) {struct` a_i`}:` T `:=` *expr*

Pour la ι-conversion, c'est la présence de constructeurs dans l'argument principal qui déclenche la réduction.

Par exemple, l'addition est définie dans les bibliothèques de *Coq* par une définition récursive à deux arguments :

```
Fixpoint plus (n m:nat){struct n} : nat :=
  match n with 0 ⇒ m | S p ⇒ S (plus p m) end.
```

Les règles de ι-réduction assurent les convertibilités décrites dans le tableau suivant :

`0 + 0`	`0`
`0 + m`	`m`
`(S n) + 0`	`S (n + 0)`
`(S n) + m`	`S (n + m)`
`(S (S n)) + (S m)`	`S (S (n + (S m)))`

Les réductions décrites dans ce tableau permettent de comprendre pourquoi les théorèmes `plus_O_n` et `plus_Sn_m` décrits dans la section 7.3.2 sont simples à démontrer. Puisque “ `0 + m` ” et `m` sont convertibles, l'énoncé de `plus_O_n` est simplement une instance de la réflexivité de l'égalité. On remarque que cette définition récursive de l'addition donne un caractère dissymétrique à l'opération binaire. On peut établir que l'addition ainsi définie est commutative, mais ce résultat fait l'objet d'une démonstration plus complexe et n'est pas donné par la réduction.

Exercice 7.13 Reprendre la discussion ci-dessus, en remplaçant l'addition par la multiplication `mult:nat→nat→nat` : donner un tableau décrivant la convertibilité de cette fonction pour des schémas simples des arguments.

Exercice 7.14 Écrire une fonction de type nat → bool qui retourne `true` seulement pour les entiers naturels inférieurs à 3, c'est à dire “ `S (S (S O))` ”. Cette fonction n'est pas à proprement parler une fonction récursive et fait intervenir un filtrage imbriqué.

Exercice 7.15 Définir une fonction d'addition telle que l'argument principal de récursion soit le deuxième et non le premier.

On prouvera que cette fonction calcule les mêmes valeurs que la fonction `plus` de *Coq*.

Exercice 7.16 Écrire la fonction `sum_f` qui prend en arguments un nombre naturel `n` et une fonction `f:nat→Z` et qui retourne la somme des valeurs de `f` pour les `n` premiers nombres naturels.

Un autre exemple de fonction récursive structurelle est la fonctionnelle `iterate` présentée en section 5.2.1.1, page 111. En voici une définition récursive :

```
Fixpoint iterate (A:Set)(f:A→A)(n:nat)(x:A){struct n} : A :=
  match n with
  | 0 ⇒ x
  | S p ⇒ f (iterate A f p x)
  end.
```

Cette fonction est intéressante car elle montre qu'une fonction récursive peut prendre des fonctions en argument et retourner des fonctions. La commande `Fixpoint` permet donc d'effectuer une programmation fonctionnelle d'*ordre supérieur*. Cette fonction a déjà été utilisée dans la section 5.2.1.1 par exemple pour décrire la fonction d'Ackermann, ce qui montre que la récursion structurelle est plus puissante que la récursion primitive de premier ordre. Par ailleurs, elle peut être comparée à la construction "`for ...`" utilisée dans les langages de programmation impératifs pour décrire les calculs qui doivent être répétés un certain nombre de fois.

Exercice 7.17 Définir la fonction `two_power: nat` $\rightarrow$ `nat` de sorte que `(two_power` n`)` soit le nombre 2^n.

7.3.4 Variations dans les constructeurs

Le type des entiers naturels présente deux constructeurs, dont un seul est récursif et ce constructeur n'a qu'une composante dans le type inductif lui-même. Il est possible d'avoir d'autres configurations. Par exemple, le type inductif suivant permet de décrire des arbres binaires dont les sommets binaires sont étiquetés par des entiers :

```
Inductive Z_btree : Set :=
  Z_leaf : Z_btree | Z_bnode : Z→Z_btree→Z_btree→Z_btree.
```

Dans cette structure de données, nous avons encore un constructeur sans argument, `Z_leaf`, dont l'importance est la même que celle de `O` pour les entiers naturels. Le second constructeur est récursif : il comporte trois champs dont deux sont dans le type que l'on est en train de définir. Le principe de récurrence prend la forme suivante :

Z_btree_ind : ∀ P:Z_btree → Prop,
P Z_leaf →
(∀ (z:Z)(z0:Z_btree),
P z0 → ∀ z1:Z_btree, P z1 → P (Z_bnode z z0 z1))→
∀ z:Z_btree, P z

Ce principe de récurrence a deux prémisses, une pour chaque constructeur. Il est remarquable que la prémisse pour le second constructeur demande de démontrer la propriété cherchée pour les arbres construits avec ce constructeur sous deux hypothèses de récurrence : une pour chaque sous-terme dans le type inductif lui-même.

Un autre exemple de variation est donné lorsque l'on fournit plus de deux constructeurs, comme dans le type `positive` utilisé dans la théorie `ZArith` de *Coq* pour décrire les nombres entiers strictement positifs.

```
Print positive.
```
Inductive positive : Set :=
xI : positive→positive

| xO : positive→positive
| xH : positive
For xI: Argument scope is [positive_ scope]
For xO: Argument scope is [positive_ scope]

Le constructeur `xH` est utilisé pour représenter le nombre 1, le constructeur `xO` est utilisé pour représenter la fonction $x \mapsto 2x$, et le constructeur `xI` est utilisé pour représenter la fonction $x \mapsto 2x + 1$. Il est évident que les fonctions représentées par `xO` et `xI` sont injectives sur l'ensemble des nombres entiers strictement positifs et que leurs codomaines sont disjoints et ne contiennent pas 1. Ce type inductif fournit donc un codage canonique des nombres entiers strictement positifs, qui correspond au codage binaire (en base 2) couramment employé dans les ordinateurs. Par exemple, le nombre $5 = 2(2 \times 1) + 1$, qui s'écrit `101` en base 2, est représenté par le terme " `xI (xO xH)` ". Le nombre 13, qui s'écrit `1101` est représenté par le terme " `xI (xO (xI xH))` ".

Les entiers relatifs sont alors décrits par un type inductif à trois constructeurs :

```
Print Z.
```

Inductive Z : Set :=
Z0 : Z | Zpos : positive→Z | Zneg : positive→Z
For Zpos: Argument scope is [positive_ scope]
For Zneg: Argument scope is [positive_ scope]

Comme leur nom l'indique, ces constructeurs sont utilisés pour représenter le nombre 0, les nombres positifs, et les nombres négatifs, respectivement. Les notations `0`, `5` et `-13` de la portée `Z_scope` ne sont ensuite que des conventions syntaxiques cachant des termes construits avec le type `Z`. Ainsi `0` cache le terme `ZERO`, `5` cache le terme " `POS (xI (xO xH))` ", et `-13` le terme " `NEG (xI (xO (xI xH)))` ". Il existe également une portée `positive_scope` qui permet d'utiliser des notations numériques pour les termes de type `positive`. Ainsi le terme `xH` s'affiche par défaut `1%P`.

Le principe de récurrence associé au type inductif `positive` est le suivant :

positive_ ind
: ∀ P:positive→Prop,
(∀ p:positive, P p → P (xI p)) →
(∀ p:positive, P p → P (xO p)) →
P 1%positive →
∀ p:positive, P p

Comme dans les autres cas, on retrouve la structure de la définition inductive dans le principe de récurrence. Ici, il y a trois prémisses principales pour les trois constructeurs et le constructeur `xH` s'affiche en fait `1%N`.

Comme pour les nombres naturels, il est possible de définir des fonctions récursives sur les nouveaux types inductifs introduits dans cette section. Dans de telles fonctions récursives, on trouvera encore une construction de filtrage sur l'argument principal et les appels récursifs ne seront permis que dans les clauses de la construction de filtrage qui font apparaître des sous-termes du même type inductif que l'argument principal.

Par exemple, la fonction suivante retourne la somme des valeurs portées dans un arbre binaire à valeurs entières. Les notations utilisées reposent sur les conventions syntaxiques fournies pour les formules contenant des entiers relatifs que nous avons décrites en section 3.2.3 :

```
Fixpoint sum_all_values (t:Z_btree) : Z :=
  (match t with
   | Z_leaf ⇒ 0
   | Z_bnode v t1 t2 ⇒
       v + sum_all_values t1 + sum_all_values t2
  end)%Z.
```

La fonction suivante cherche si le nombre 0 apparaît dans un arbre binaire. Notez que l'arbre n'est pas parcouru entièrement si 0 apparaît :

```
Fixpoint zero_present (t:Z_btree) : bool :=
  match t with
  | Z_leaf ⇒ false
  | Z_bnode (0%Z)  t1 t2 ⇒ true
  | Z_bnode _ t1 t2 ⇒
      if zero_present t1 then true else zero_present t2
  end.
```

Comme dernier exemple considérons une fonction définie dans les bibliothèques de *Coq* qui retourne le successeur d'un nombre binaire strictement positif :

```
Fixpoint Psucc (x:positive) : positive :=
  match x with
  | xI x' ⇒ xO (Psucc x')
  | xO x' ⇒ xI x'
  | xH ⇒ 2%positive
  end.
```

Exercice 7.18 Quelle est la représentation dans le type `positive` des nombres 1000, 25, 512 ?

Exercice 7.19 Construire la fonction `pos_even_bool : positive → bool`, qui associe la valeur `true` à un nombre exactement quand ce nombre est pair.

Exercice 7.20 Construire la fonction `pos_div4` de type `positive→Z` qui associe à tout nombre z la partie entière de $z/4$.

Exercice 7.21 On suppose que l'on dispose d'une fonction `pos_mult` qui réalise la multiplication de deux nombres de type `positive` et retourne le produit de ces deux nombres sous la forme d'un nombre de type `positive`. Utiliser cette fonction pour construire une fonction de multiplication de type `Z→Z→Z`.

Exercice 7.22 Construire le type inductif décrivant le langage des formules logiques du Calcul des Propositions sans variables :

$$\mathcal{L} = \mathcal{L} \wedge \mathcal{L} | \mathcal{L} \vee \mathcal{L} | \sim \mathcal{L} | \mathcal{L} \Rightarrow \mathcal{L} | \mathsf{L\_True} | \mathsf{L\_False}.$$

Exercice 7.23 * Tout nombre rationnel strictement positif peut être obtenu de façon unique par une séquence d'applications des fonctions N et D sur le nombre 1, où N et D sont définis par les équations suivantes :

$$\begin{array}{rcl} N(x) & = & 1 + x \\ D(x) & = & \dfrac{1}{1 + \frac{1}{x}} \end{array}$$

On peut donc associer à tout nombre rationnel strictement positif un élément d'un type inductif construit avec une constante représentant 1, et deux constructeurs représentant N et D. Définir ce type inductif (*suite dans l'exercice 7.43*).

Exercice 7.24 Les bibliothèques de *Coq* fournissent une fonction

Zeq_ bool:Z→Z→bool

qui permet de comparer deux nombres entiers. À l'aide de cette fonction, écrire la fonction `value_present` dont le type est le suivant :

```
value_present : Z→Z_btree→bool
```

et qui détermine si une valeur donnée apparaît dans un arbre binaire.

Exercice 7.25 Définir la fonction `power: Z → nat → Z` qui calcule la puissance d'un nombre entier, puis la fonction `discrete_log : positive → nat`, qui associe à p le nombre n tel que $2^n \leq p < 2^{n+1}$.

7.3.5 ** Constructeurs prenant des fonctions en arguments

7.3.5.1 Représentation indexée

Une méthode alternative pour construire des arbres binaires est de considérer que les différentes branches sont indexées par un type à deux éléments, le plus simple étant de prendre le type des booléens. Ainsi, la définition suivante décrit aussi un type d'arbres binaires portant des valeurs entières.

```
Inductive Z_fbtree : Set :=
  Z_fleaf : Z_fbtree | Z_fnode : Z →(bool→Z_fbtree)→ Z_fbtree.
```

Pour considérer la première et la deuxième branche des arbres binaires, il est nécessaire d'établir une convention, par exemple `true` sera utilisé pour référencer la première branche, tandis que `false` sera utilisé pour référencer la deuxième branche. Pour bien comprendre les similitudes entre le type `Z_btree` et le type `Z_fbtree`, observons comment on écrirait la fonction qui associe à tout arbre son fils droit si on est dans le cas d'un nœud binaire ou lui-même si on est dans le cas d'une feuille dans ces deux types :

```
Definition right_son (t:Z_btree) : Z_btree :=
  match t with
  | Z_leaf ⇒ Z_leaf
  | Z_bnode a t1 t2 ⇒ t2
  end.

Definition fright_son (t:Z_fbtree) : Z_fbtree :=
  match t with
  | Z_fleaf ⇒ Z_fleaf
  | Z_fnode a f ⇒ f false
  end.
```

Le principe de récurrence engendré pour le type `Z_fbtree` fait apparaître les mêmes possibilités pour avoir des hypothèses de récurrence sur les sous-arbres que le principe de récurrence pour le type `Z_btree`. Si `f` est la fonction apparaissant dans le deuxième champ d'un nœud `Z_fnode`, alors les deux sous-arbres sont atteints par “ `f true` ” et “ `f false` ”. Exprimer que “ `f true` ” et “ `f false` ” doivent satisfaire l'hypothèse de récurrence peut aussi se faire en disant que *pour tout* `b` *de type* `bool`*,* `(f b)` *doit satisfaire l'hypothèse de récurrence.* C'est cette approche qui est suivie systématiquement lors de la génération du principe de récurrence par *Coq* :

```
Z_fbtree_ind
    : ∀ P:Z_fbtree → Prop,
        P Z_fleaf →
        (∀ (z:Z)(z0:bool→Z_fbtree),
           (∀ b:bool, P (z0 b))→ P (Z_fnode z z0))→
        ∀ z:Z_fbtree, P z
```

La définition de fonctions récursives pour un type inductif dont certains constructeurs reçoivent des fonctions en argument se généralise assez naturellement : les images des fonctions qui apparaissent en argument des constructeurs sont naturellement des sous-termes de l'expression complète. Si ces images sont dans le type récursif lui-même, il est alors naturel de permettre un appel récursif sur elles.

Par exemple, nous pourrons naturellement écrire la fonction qui calcule la somme des nombres présents dans un arbre binaire pour le type `Z_fbtree` :

```
Fixpoint fsum_all_values (t:Z_fbtree) : Z :=
 (match t with
  | Z_fleaf ⇒ 0
  | Z_fnode v f ⇒
     v + fsum_all_values (f true) + fsum_all_values (f false)
  end )%Z .
```

Exercice 7.26 Définir la fonction `fzero_present:Z_fbtree → bool` qui recherche si le nombre 0 apparaît dans un arbre binaire.

7.3.5.2 *** Arbres à branchement infini

On peut aller encore plus loin, les fonctions utilisées pour définir les champs n'ont pas nécessairement la contrainte de prendre leur argument dans un ensemble fini [4]. Ainsi, on pourra représenter les arbres à branchement infini en fournissant une fonction dont le domaine de départ est l'ensemble des entiers naturels. Ceci revient à dire que l'on indexe les branches à l'aide des entiers naturels. Voici la définition inductive qui permet de construire une telle structure de donnée :

```
Inductive Z_inf_branch_tree : Set :=
  Z_inf_leaf : Z_inf_branch_tree
| Z_inf_node : Z→(nat→Z_inf_branch_tree)→Z_inf_branch_tree.
```

Si l'arbre présente un branchement infini, il n'est bien sûr pas possible d'écrire une fonction qui additionnerait toutes les valeurs portées dans l'arbre. En revanche, on peut définir la fonction qui additionne toutes les valeurs accessibles uniquement par des indices inférieurs à un certain nombre, en utilisant comme fonction auxiliaire la fonction `sum_f` décrite dans l'exercice 7.16 :

```
Fixpoint n_sum_all_values (n:nat)(t:Z_inf_branch_tree){struct t}
  : Z :=
  (match t with
    | Z_inf_leaf ⇒ 0
    | Z_inf_node v f ⇒
        v + sum_f n (fun x:nat ⇒ n_sum_all_values n (f x))
    end )%Z.
```

Exercice 7.27 ** Définir la fonction qui cherche dans un arbre à branchement infini si la valeur zéro apparaît dans un sous-arbre, accessible uniquement par des indices inférieurs à `n`.

7.3.6 Raisonnement sur les fonctions récursives

La tactique `simpl` est particulièrement adaptée pour appliquer les règles de réduction sur des expressions contenant des fonctions récursives. Cette tactique est un complément naturel des tactiques `case` et `elim`, puisque ces tactiques introduisent des constructeurs dans les expressions ce qui permet ensuite de déclencher les règles de réductions associées aux fonctions récursives.

L'argument principal d'une fonction récursive structurelle joue un rôle particulier. Le calcul de la fonction suit la structure de cet argument. Pour cette raison, tout raisonnement que l'on fait sur la valeur prise par une fonction récursive structurelle s'appuie naturellement sur une démonstration par récurrence sur cet argument. Cet affirmation mérite qu'on la répète et qu'on la souligne.

4. d'ailleurs le système *Coq* ne fournit pas de moyen direct pour décrire qu'un ensemble est fini

Les raisonnements sur les fonctions récursives structurelles se font naturellement par récurrence sur l'argument principal de ces fonctions puis en suivant la structure des constructions de filtrage contenues dans ces fonctions.

La démonstration de l'associativité de l'addition de la page 190 peut être simplifiée en utilisant la tactique `simpl` :

```
Theorem plus_assoc' : ∀x y z:nat, x+y+z = x+(y+z).
Proof.
```

Dans la définition de `plus` (voir page 193), l'argument principal est le premier argument. Toutes les occurrences de `x` apparaissent comme argument principal de la fonction `plus`, il est donc judicieux d'utiliser une preuve par récurrence sur cette variable.

```
 intros x y z; elim x.
  ...
  ============================
  0+y+z = 0+(y+z)
```

Appliquée au premier but, la tactique `simpl` donne directement le but suivant :

```
 simpl.
  ...
  ============================
  y+z = y+z
```

ce qui se résout automatiquement, par la tactique `trivial`. Le second but engendré par la démonstration par récurrence a la forme suivante :

```
trivial.
  ...
  ============================
∀ n:nat,
  n+y+z = n+(y+z)→ S n + y + z = S n + (y+z)
```

L'application de la tactique `simpl` mène au but suivant :

```
 simpl.
  ...
  ============================
   ∀ n:nat,
     n+y+z = n+(y+z)→ S (n+y+z) = S (n+(y+z))
```

Ce but se résout aisément, par exemple par la commande suivante :

```
 intros n H; rewrite H; auto.
Qed.
```

Exercice 7.28 Refaire la démonstration du théorème `plus_n_O` :

plus_n_O : $\forall$ *n:nat, n = n+0*

en utilisant seulement les tactiques `intros`, `assumption`, `elim`, `simpl`, et `apply` et `reflexivity`.

Exercice 7.29 ** Cet exercice utilise les types inductifs `Z_btree` et `Z_fbtree` introduits dans les sections 7.3.4 et 7.3.5.1. Écrire les deux fonctions

`f1:Z_btree→Z_fbtree` et `f2:Z_fbtree→Z_btree`

fi qui établissent la bijection la plus naturelle entre les deux types d'arbres binaires. Démontrer le théorème suivant :

```
∀t:Z_btree, f2 (f1 t) = t.
```

Que manque-t-il pour montrer la proposition suivante (merci à J.-F. Monin et T. Heuillard) ?

```
∀t:Z_fbtree, f1 (f2 t) = t
```

Exercice 7.30 Démontrer le théorème suivant sur la fonction `mult2` (voir page 192) :

$$\forall n : \texttt{nat}.(\texttt{mult2}\ n) = n + n$$

Exercice 7.31 On définit la somme des n premiers entiers avec la fonction suivante :

```
Fixpoint sum_n (n:nat) : nat :=
  match n with
  | 0 ⇒ 0
  | S p ⇒ S p + sum_n p
  end.
```

Démontrer le théorème suivant :

```
∀n:nat, 2 * sum_n n = n + n
```

Exercice 7.32 Avec les définitions de l'exercice 7.31, montrer l'inégalité :

```
∀n:nat, n ≤ sum_n n
```

7.3.7 Fonctions récursives anonymes (`fix`)

L'abstraction permet de construire une fonction non récursive directement à l'intérieur d'un terme du Calcul des Constructions, sans lui donner de nom. La commande `Fixpoint` n'a pas cette propriété. Elle couple deux opérations : d'une part la description d'une fonction récursive, d'autre part l'introduction de cette fonction récursive dans l'environnement global. La construction `fix` permet de décomposer ces deux opérations en fournissant une fonctionnalité similaire à l'abstraction. La syntaxe de cette construction est très semblable à celle de la commande `Fixpoint` :

`fix f (`a_1`:`T_1`)... (`a_p`:`T_p`) {struct `a_i`}: `T` := `*expr*

Comme pour `Fixpoint`, la directive `{struct` a_i`}` est facultative si p=1.

Rappelons cependant que `Fixpoint` débute une commande, tandis que `fix` permet de construire un terme du Calcul des Constructions Inductives.

Pour l'instant, nous ne décrivons que le cas particulier des fonctions simplement récursives, une généralisation aux fonctions mutuellement récursives est décrite en section 15.3.1 page 442.

Cette construction anonyme a plusieurs utilités. La première est que le système l'utilise systématiquement pour imprimer la valeur d'une fonction récursive. La δ-réduction d'une fonction définie à l'aide de `Fixpoint` est toujours une fonction `fix`. Un usage abusif des tactiques de réduction comme `simpl` peut faire apparaître ce type de terme. Nous voyons également en section 15.3.3 un exemple sophistiqué où la commande `Fixpoint` ne peut pas être utilisée.

L'identificateur f sert à dénoter la fonction récursive que nous définissons, mais il ne peut être utilisé qu'à l'intérieur de l'expression *expr*. Les arguments a_1, ..., a_p décrivent les arguments de la fonction `f` et l'argument désigné par la directive `{struct` *a_i*`}` est l'argument principal de récursion. L'expression T sert à décrire le type de la fonction *f*, comme dans la commande `Fixpoint`. La similitude entre cette construction et la commande `Fixpoint` permet de comprendre l'usage des différentes parties de cette construction. Par exemple, la fonction `mult2` aurait également pu être déclarée par la commande suivante :

```
Definition mult2' : nat→nat :=
  fix f (n:nat) : nat :=
    match n with 0 ⇒ 0 | S p ⇒ S (S (f p)) end.
```

Ici nous avons volontairement changé le nom donné à la fonction à l'intérieur de la construction `fix` pour souligner le fait que l'identificateur `f` n'est lié qu'à l'intérieur de cette construction. Cet identificateur n'a donc pas de relation avec le nom sous lequel on pourra utiliser la fonction.

7.4 Types polymorphes

Parmi les opérations que l'on peut effectuer sur les arbres d'entiers relatifs, il y en a un grand nombre qui ne reposent pas sur le fait que les éléments sont des entiers, mais seulement sur la structure d'arbre. Par exemple, le calcul de la taille ou de la hauteur d'un arbre est indépendant du type des données qu'il contient. Il est raisonnable de définir un type général d'arbres, dans lequel le type des éléments n'est pas entièrement spécifié, et d'utiliser des *instances* de ce type suivant les besoins apparaissant dans nos algorithmes. Cette possibilité de laisser un type variable est déjà présente dans les langages de programmation comme Ada, où l'on parle de types génériques et dans les langages de la famille ML, où l'on parle de polymorphisme. Nous montrerons cette possibilité sur les types de listes, couples, etc. Le type des arbres polymorphes fera l'objet de l'exercice 7.42.

7.4.1 Le type des listes polymorphes

Le système *Coq* fournit une théorie des listes polymorphes, dans lequel le type inductif des listes est décrit par la définition suivante :

```
Require Import List.
Print list.
```

Inductive list (A : Set) : Set :=
nil : list A | cons : A→ list A → list A
For nil: Argument A is implicit
For cons: Argument A is implicit
For list: Argument scope is [type_scope]
For nil: Argument scope is [type_scope]
For cons: Argument scopes are [type_scope _ _]

Le système *Coq* fournit une notation pour les listes, de telle sorte que l'expression " `cons` a l " est en fait notée " a`::`l ".

Nous voyons que le type défini n'apparaît plus dans les constructeurs comme un simple identificateur, mais appliqué à un argument, qui est d'ailleurs toujours le même. Cet argument est `A`, le paramètre, qui a en fait été donné à la première ligne, dans le fragment de texte `(A:Set)`.

Tout se passe comme si l'on ne définissait pas un seul type inductif mais toute une famille de types inductifs, indexée par les éléments du type `Set`, ce qui illustre la construction de types d'ordre supérieur comme nous l'avons abordée dans le chapitre 5.

Il peut y avoir plusieurs paramètres donnés dans la définition inductive. Lorsque des paramètres sont fournis, il doivent apparaître partout où l'on fait référence au type défini inductivement. Tout se passe comme si l'on définissait un type inductif sans paramètre dans un contexte où `A` était déclaré comme une variable. Ainsi la définition plus haut est pratiquement équivalente à la suite de commandes suivantes :

```
Section define_lists.
 Variable A : Set.
 Inductive list' : Set :=
 | nil' : list'
 | cons' : A → list' → list'.
End define_lists.
```

Cette analogie entre types inductifs polymorphes et types inductifs définis dans la portée d'une variable à l'intérieur d'une section permet de comprendre comment le système *Coq* construit le principe de récurrence associé à un type inductif polymorphe. Observons d'abord quelle serait la forme du principe de récurrence dans la portée de la déclaration de `A` :

```
list'_ind0 :
   ∀P : list'→Prop,
      P nil' →
```

```
      (∀(x:A)(l:list'), P l → P (cons' x l))→
    ∀x:list', P x.
```

Au moment où la section est fermée et la variable `A` est déchargée, toute déclaration de la forme $x : B$ où le type B dépend de `A` est remplacée par " x : ∀`A:Set,` B ".

C'est le cas des deux constantes `nil'` et `cons'` dont le type devient alors " ∀`A:Set,list' A` " [respectivement " ∀`A:Set,A→list' A→list' A` "]. Le type inductif doit lui-même être adapté de façon à tenir compte des modifications ci-dessus. Ainsi le type de `list'` devient-il `Set→Set` et le principe de récurrence devient est modifié en conséquence.

```
Check list'.
```
list' : Set→Set
```
Check nil'.
```
nil' : ∀ A:Set, list' A
```
Check cons'.
```
cons' : ∀ A:Set, A → list' A → list' A
```
Check list'_ind.
```
list'_ind :
∀ (A:Set)(P:list' A → Prop),
P (nil' A) →(∀ (a:A)(l:list' A), P l → P (cons' A a l)) →
∀ l:list' A, P l

Une propriété importante du principe de récurrence des définitions paramétrées est que la quantification universelle sur `A` englobe la quantification universelle sur `P` et que la première n'est pas répétée dans les prémisses principales du principe de récurrence.

Les fonctions récursives et le filtrage sur les types polymorphes fonctionnent comme pour les types vus dans les sections précédentes. Néanmoins, il est important de noter une différence de taille : les paramètres ne doivent pas apparaître dans les expressions de filtrage. Ainsi, la fonction de concaténation de listes est définie dans les bibliothèques de *Coq* de la façon suivante :

```
Fixpoint app (A:Set)(l m:list A){struct l} : list A :=
  match l with
  | nil ⇒ m
  | cons a l1 ⇒ cons a (app A l1 m)
  end.
```

Dans le schéma de filtrage, `nil` apparaît sans argument alors que c'est l'expression " `nil (A:=nat)` " qui est de type " `list A` ". De même `(cons a l1)` apparaît également avec seulement deux arguments dans le schéma de filtrage pour la deuxième clause, alors que `cons` reçoit normalement trois arguments. La raison de cette distinction est que l'argument paramétrique n'est pas réellement lié dans la partie gauche de la règle de filtrage : sa valeur est effectivement imposée par le type de `l`. Il n'est donc pas nécessaire d'inclure une variable supplémentaire dans le schéma de filtrage. Dans la partie droite de la seconde clause, `cons` et `app` apparaissent également avec seulement deux arguments, alors que ce sont réellement des fonctions à trois arguments, mais ceci est dû à l'utilisation d'arguments implicites comme en section 3.2.3 : la valeur du premier argument est inférée à partir du type du deuxième argument. L'usage d'arguments implicites pour les types polymorphes est fréquent et permet de se rapprocher du style de programmation habituel dans les langages fonctionnels polymorphes.

Pour la fonction `app`, le système *Coq* fournit aussi une notation infixe, telle que « `app` l_1 l_2 » est actuellement noté « l_1++l_2 ».

Exercice 7.33 Construire une fonction qui prend une liste en argument et calcule la liste des deux premiers éléments lorsque ceux-ci existent.

Exercice 7.34 Construire une fonction qui prend une liste et un nombre naturel n en argument et construit la liste des n premiers éléments lorsque ceux-ci existent.

Exercice 7.35 Construire la fonction qui prend une liste d'entiers et retourne leur somme.

Exercice 7.36 Construire la fonction qui prend un nombre naturel n en argument et construit la liste des entiers de n à 1.

Exercice 7.37 * Construire la fonction qui prend un nombre naturel n en argument et construit la liste des entiers de 1 à n.

7.4.2 Le type option

Bien sûr, la possibilité de définir des types polymorphes s'applique aussi pour des types qui ne présentent pas réellement de récursivité. Un exemple fréquemment utilisé est le type `option` qui est adapté à la description de fonctions partielles. Ce type est présent également dans les principaux dialectes ML. Il a la forme suivante :

```
Print option.
```
Inductive option (A:Set) : Set :=
Some : A→option A | None : option A
For Some: Argument A is implicit
For None: Argument A is implicit
For option: Argument scope is [type_scope]
For Some: Argument scopes are [type_scope _]
For None: Argument scope is [type_scope]

Lorsque l'on doit définir une fonction qui n'est pas totale d'un ensemble `A` vers un ensemble `B`, il est parfois possible de la définir comme une fonction totale de l'ensemble `A` vers l'ensemble « `option B` » en convenant simplement de retourner la valeur « None » lorsque la fonction n'est pas définie et la valeur « `Some y` » quand la fonction aurait dû avoir la valeur `y`.

Par exemple, on peut définir une fonction qui retourne le prédécesseur d'un nombre naturel s'il existe :

```
Definition pred_option (n:nat) : option nat :=
  match n with O ⇒ None | S p ⇒ Some p end.
```

Lorsque l'on utilise une valeur de type `option` il faut régulièrement effectuer un traitement par cas pour exprimer explicitement ce qui est calculé lorsque l'on ne dispose pas vraiment de valeur. Par exemple, la fonction qui retourne le prédécesseur du prédécesseur lorsqu'il existe pourra être définie de la façon suivante :

```
Definition pred2_option (n:nat) : option nat :=
  match pred_option n with
  | None ⇒ None
  | Some p ⇒ pred_option p
  end.
```

Par ailleurs, notons que les bibliothèques de *Coq* fournissent une fonction totale `pred` dont la valeur en zéro est zéro.

Pour un deuxième exemple, nous pouvons considérer la fonction qui retourne le n-ième élément d'une liste. Les bibliothèques de *Coq* fournissent une fonction `nth` pour ce besoin, mais le programmeur a préféré retourner une valeur par défaut donnée en argument supplémentaire plutôt que d'exprimer explicitement le cas où l'élément de rang n n'existe pas. Une alternative est de définir une fonction `nth_option` qui décrit explictement le cas où il n'existe pas d'élément de rang n. Cette fonction peut se définir en effectuant un filtrage simultané sur le nombre et sur la liste. Les deux arguments décroîtront dans les appels récursifs, de sorte que l'argument principal de récursion peut être l'un ou l'autre. Voici une des deux versions possibles.

```
Fixpoint nth_option (A:Set)(n:nat)(l:list A){struct l}
  : option A :=
  match n, l with
  | O, cons a tl ⇒ Some a
  | S p, cons a tl ⇒ nth_option A p tl
  | n, nil ⇒ None
  end.
```

Exercice 7.38 * Définir la fonction `nth_option'` récursive sur son seul argument numérique, de même spécification que `nth_option`. Prouver en *Coq* que ces deux fonctions retournent toujours les mêmes valeurs sur les mêmes entrées.

Exercice 7.39 * Montrer le théorème suivant :

```
∀ (A:Set)(n:nat)(l:list A),
     nth_option A n l = None → length l ≤ n.
```

Nous décrirons dans le chapitre 9 les moyens de prouver la réciproque.

Exercice 7.40 * Définir une fonction qui prend en arguments un type `A` de sorte `Set`, une fonction de type `A→bool`, et une liste d'éléments de `A` et renvoie, s'il existe, le premier élément de la liste pour lequel la fonction vaut `true`.

7.4.3 Le type des couples

La construction de couples de valeurs est un autre exemple caractéristique de type polymorphe : lorsque l'on regroupe deux données de types respectifs `A` et `B` dans un couple, on obtient une donnée de type `A*B`. Ce type est donc paramétré par les types `A` et `B`. La définition inductive des couples est donc donnée par le type suivant :

```
Inductive prod (A:Set)(B:Set) : Set := pair : A→B→(prod A B).
```

Outre la définition inductive, le système fournit quelques fonctions annexes, en particulier les fonctions `fst` et `snd` qui sont les deux projecteurs permettant de récupérer les composantes d'un couple. La description de ces fonctions peut être obtenue en utilisant la commande `Print` :

```
Print fst.
fst =
fun (A B:Set)(p:A*B) ⇒ let (x, _) := p in x
     : ∀ A B:Set, A*B→A
Arguments A, B are implicit
Argument scopes are[type_scope type_scope _]
```

En plus des fonctions annexes, *Coq* fournit également des conventions syntaxiques. Premièrement le produit “ `prod A B` ” peut également être écrit `A*B`, ce qui apparaît déjà dans la réponse du système au deux commandes `Print` ci-dessus.

Deuxièmement, l'expression “ `pair (A:=A) (B:=B) a b` ” peut également s'écrire `(a,b)`, en bénéficiant du mécanisme d'arguments implicites.

Exercice 7.41 Définir deux fonctions `split` et `combine` ayant le type suivant

```
split : ∀A B:Set, list A*B→list A * list B
combine : ∀A B:Set, list A → list B → list A*B
```

CoÉcrire et démontrer un théorème reliant ces deux fonctions.

Exercice 7.42 Construire le type des arbres binaires `btree` polymorphes. Définir deux fonctions de traduction de types “ `Z_btree→btree Z` ” et “ `btree Z→Z_btree` ”. Prouver qu'elles sont inverses l'une de l'autre.

Exercice 7.43 Cet exercice fait suite à l'exercice 7.23 de la page 198. Construire la fonction qui prend un élément du type inductif représentant les nombres rationnels et retourne le couple constitué du numérateur et du dénominateur de la fraction réduite correspondante.

Exercice 7.44 *** Le but de cet exercice est d'implémenter une fonction de crible pour retrouver tous les nombres premiers inférieurs à un nombre donné. On commence par définir un type de valeurs de comparaison

```
Inductive cmp : Set := Less : cmp | Equal : cmp | Greater : cmp.
```

Ensuite le lecteur doit définir trois fonctions :

1. `three_way_compare : nat→nat→cmp`, qui compare deux nombres naturels.
2. `update_primes : nat→list nat*nat→(list nat*nat)*bool` qui reçoit en argument un nombre k et une liste de couples d'entiers de la forme (p, m) tels que m est le plus petit multiple de p supérieur ou égal à k et retourne une liste de couples d'entiers (p, m') où m' est le plus petit multiple de p supérieur ou égal à $k+1$ et une valeur booléenne vraie si k était égal à l'un des m dans la liste reçue en entrée.
3. `prime_sieve : nat→list nat*nat` qui associe à k la liste composée des couples (p, m) où p est un nombre premier inférieur ou égal à k et m est le plus petit multiple de p supérieur ou égal à $k+1$.

Démontrer que `prime_sieve` permet bien de retrouver tous les nombres premiers inférieurs à un nombre donné.

7.4.4 Le type des sommes disjointes

Le type des couples joue le rôle des types enregistrement, en laissant le type des différentes composantes sous forme de paramètre. Le type `option` permet d'avoir des variantes, mais l'une des variantes doit être vide. Pour fournir un jeu complet de types paramétrés simples, le système *Coq* fournit aussi un type de somme disjointe autorisant deux variantes, dont chacune peut contenir une donnée. Ce type est décrit par la définition inductive suivante :

```
Inductive sum (A:Set)(B:Set):Set :=
  inl : A→sum A B | inr : B→sum A B.
```

En outre le système *Coq* fournit une notation abrégée pour ce type, de sorte que `(sum A B)` s'écrit `A+B` dans la portée `type_scope` .

```
Check (sum nat bool).
```
(nat+bool)%type
: Set

```
Check (inl bool 4).
```
inl bool 4
: (nat+bool)%type

```
Check (inr nat false).
```
inr nat false
: (nat+bool)%type

7.5 * Types inductifs dépendants

7.5.1 Paramètres du premier ordre

Les paramètres d'une définition inductive ne sont pas nécessairement des types, il peuvent aussi être des valeurs. Cette forme de paramétrisme permet d'établir des contraintes sur les données incluses dans le type considéré, où les contraintes sont exprimées à l'aide de prédicat. Par exemple, on peut définir un type d'arbre dont toutes les valeurs portées sont limitées par une borne fixée n :

```
Inductive ltree (n:nat) : Set :=
  | lleaf : ltree n
  | lnode : ∀p:nat, p ≤ n → ltree n → ltree n → ltree n.
```

Le type du constructeur `lnode` exprime bien que la valeur portée à cet endroit devra être accompagnée d'une preuve que `p` est inférieur ou égal à `n`. Comme c'est le cas ici, de tels types paramétrés font souvent intervenir des constructeurs dont le type est un produit dépendant.

Les types inductifs paramétrés par une valeur sont également utilisés de façon extensive pour décrire des types informatifs. Ainsi, la racine carrée certifiée d'un nombre `n` n'est pas seulement un nombre, mais un nombre accompagnée d'un certificat, c'est-à-dire une preuve que ce nombre satisfait bien la définition de la racine carrée. On pourra donc définir un type inductif pour regrouper toutes ces informations, paramétré par le nombre dont on extrait la racine :

```
Inductive sqrt_data (n:nat) : Set :=
  sqrt_intro : ∀x:nat, x*x ≤ n → n <  (S x)*(S x)→sqrt_data n.
```

Une fonction de type “ `∀n:nat, sqrt_data n` ” construit à la fois la valeur de la racine carrée de son argument et le certificat associé. Le chapitre 10 est consacré aux moyens de construire de telles fonctions.

7.5.2 Constructeurs à type dépendant variable

Dans les types paramétrés, l'argument dépendant est toujours utilisé avec la même valeur. Il ne varie pas d'un terme à un sous-terme. Ainsi, les termes du type “ `list A` ” sont construits avec d'autres termes du même type “ `list A` ”.

Cette uniformité n'est pas générale ; ainsi il nous arrivera de définir toute une famille de types indexée par des paramètres pouvant varier au sein de la définition. Ceci se produit fréquemment dans le cas de types de donnée récursifs dépendants.

Par exemple, l'algorithme de transformée de Fourier rapide étudié par V. Capretta dans [21] utilise des arbres binaires équilibrés, où toutes les branches sont contraintes à avoir la même hauteur. Voici comment on définit une telle structure de donnée dans le système *Coq*.

```
Inductive htree (A:Set) : nat→Set :=
```

```
| hleaf : A→htree A 0
| hnode : ∀n:nat, A → htree A n → htree A n → htree A (S n).
```

Dans cette définition il faut comprendre que le type " `htree A n` " représente le type des arbres de hauteur `n`. Le premier constructeur indique que tous les arbres de hauteur `0` sont des feuilles (ou réciproquement que les feuilles sont des arbres de hauteur `0`), tandis que le deuxième constructeur indique explicitement que l'on obtient un arbre de hauteur `n`+1 en réunissant une valeur et deux arbres de hauteur `n`.

Le principe de récurrence engendré pour cette définition inductive est encore obtenu de façon systématique, mais il est nécessaire de généraliser la technique. En effet, la propriété à démontrer porte non sur un type de donnée spécifique (en tant que type de sorte `Set`) mais sur toute la famille de types considérée. Il s'agit de démontrer une propriété sur tous les arbres de hauteur n, et pour tout n. Pour construire une expression bien typée, il faut donc que la propriété ait un type dépendant, portant sur un nombre naturel n et sur un arbre de hauteur n. De plus, le principe de récurrence va faire référence à des arbres de hauteurs différentes, de sorte que l'on ne peut pas introduire n une fois pour toutes à l'extérieur du principe de récurrence, comme c'était le cas pour les types polymorphes. Ainsi, l'en-tête du principe de récurrence pour le type `htree` a la forme suivante :

```
∀(A:Set)(P:∀n:nat, htree A n → Prop)
```

Si le type de la proposition `P` était " `htree A n→Prop` ", la valeur de n serait fixée dans le type de cette proposition. On serait alors contraint de n'utiliser que des hypothèses de récurrence sur des arbres de hauteur n pour démontrer des propriétés d'arbres de hauteur n. Ce serait visiblement inadapté, puisque les sous-arbres d'un arbre de hauteur n sont des arbres de hauteur différente.

Il est ensuite nécessaire d'adapter le reste du principe de récurrence pour tenir compte de cette nouvelle situation où la propriété considérée est un prédicat à plusieurs arguments. Le principe de récurrence complet a la forme suivante :

```
Check htree_ind.
```

htree_ind
  : ∀ (A:Set)(P:∀ n:nat, htree A n → Prop),
    (∀ a:A, P 0 (hleaf A a))→
    (∀ (n:nat)(a:A)(h:htree A n),
      P n h →
      ∀ h0:htree A n, P n h0→P (S n)(hnode A n a h h0))→
    ∀ (n:nat)(h:htree A n), P n h

Lorsque l'on décrit des fonctions récursives sur cette catégorie de types inductifs, les arguments dépendants des constructeurs sont des arguments à part entière et doivent alors apparaître dans les schémas de filtrage.

Par exemple, la fonction qui associe à tout arbre binaire équilibré à valeurs entières un arbre binaire de type `Z_btree` (sans en certifier l'équilibre) s'écrira de la façon suivante :

```
Fixpoint htree_to_btree (n:nat)(t:htree Z n){struct t}
  : Z_btree :=
  match t with
  | hleaf x ⇒ Z_bnode x Z_leaf Z_leaf
  | hnode p v t1 t2 ⇒
      Z_bnode v (htree_to_btree p t1)(htree_to_btree p t2)
  end.
```

Lorsque l'on définit des fonctions récursives produisant des termes dans un type dépendant, la construction de filtrage doit également être adaptée, car le type des données construites dans chaque branche peut varier suivant les cas. Le système *Coq* impose alors de préciser le type de la valeur retournée en fonction de la valeur filtrée, ceci au moyen d'une directive

match t in T a_1 a_2 ...a_n return T' with ...

Dans cette notation T est un type dépendant des n arguments a_1, ..., a_n, et T' le type de chaque branche de l'expression de filtrage en fonction des valeurs prises par les a_i dans le terme filtré t. Par exemple, la fonction qui retourne le miroir d'un arbre binaire équilibré a la forme suivante :

```
Fixpoint invert (A:Set)(n:nat)(t:htree A n){struct t}
   : htree A n :=
  match t in htree _ x return htree A x with
  | hleaf v ⇒ hleaf A v
  | hnode p v t1 t2 ⇒
    hnode A p v (invert A p t2)(invert A p t1)
  end.
```

Dans la construction de filtrage utilisée pour `invert`, la valeur retournée pour la première clause a le type “ `htree A 0` ” tandis que la valeur retournée pour la seconde clause a le type “ `htree A (S p)` ”, ce sont clairement des types différents. De plus, nous ne pouvons pas fixer à l'avance la hauteur des arbres qui apparaissent dans la deuxième clause, car pour retourner un arbre de hauteur n, il faut retourner des arbres de hauteur $n-1$, $n-2$, et ainsi de suite jusqu'à 0.

Exercice 7.45 ** Démontrer l'un des lemmes d'injection pour le constructeur `hnode` :

```
∀(n:nat)(t1 t2 t3 t4:htree nat n),
     hnode nat n 0 t1 t2 = hnode nat n 0 t3 t4→t1 = t3
```

La tactique `injection` est ineffective, utiliser la méthode décrite en section 7.2.5.

Exercice 7.46 Définir une fonction qui prend un entier n et construit un arbre équilibré de hauteur n contenant des valeurs entières.

Exercice 7.47 * Définir inductivement le type `binary_word` utilisé dans la section 5.1.1 ainsi que la fonction `binary_word_concat`.

Exercice 7.48 ** Définir la fonction `binary_word_or` calculant le « ou » bit à bit de deux mots booléens de même longueur (comme l'opérateur `|` de *C*).

Exercice 7.49 ** Définir une fonction à type dépendant, qui retourne `true` pour les nombres naturels de la forme $4n + 1$, `false` pour les nombres de la forme $4n + 3$, et n pour les nombres de la forme $2n$.

7.6 * Types vides

7.6.1 Types vides non dépendants

Lorsque l'on décrit un type de données inductif en *Coq* on oublie trop souvent que l'on risque de définir un type vide. Parfois, la construction d'un type vide est volontaire, mais parfois non. Il peut arriver qu'un utilisateur effectue des preuves compliquées sur la valeur retournée par une fonction prenant ses arguments dans un certain type inductif, sans s'apercevoir que ce type est vide.

Le type de données vide est défini dans les bibliothèques de *Coq* par la déclaration suivante :

```
Inductive Empty_set:Set :=.
```

En résumé, si l'on peut produire un élément de `Empty_set`, alors on peut prouver n'importe quelle propriété.

Le type `Empty_set` est évidemment construit pour être vide et personne ne sera trompé par cette définition, qui « exhibe son vide au grand jour ». En revanche, la situation est moins évidente pour le type suivant :

```
Inductive strange : Set :=  cs : strange→strange.
```

Ce type a bien un constructeur et pourtant il est vide. La raison est simple : pour construire un élément de ce type, il faut déjà en avoir un. Pour construire ce dernier, il faudrait en avoir un autre, pour construire cet autre,... La raison en est simple mais la démonstration semble infinie. En fait, cet argument se résume à une preuve assez simple si nous utilisons le principe de récurrence associé à ce type inductif :

```
Check strange_ind.
```

strange_ind
    : ∀ P:strange→Prop, (∀ s:strange, P s → P (cs s))→∀ s:strange, P s

Montrons maintenant que nous pouvons déduire une contradiction de l'existence d'un élément dans le type `strange`.

```
Theorem strange_empty : ∀x:strange, False.
Proof.
```

Nous allons simplement effectuer une preuve par récurrence sur `x` :

```
 intros x; elim x.
...
x : strange
============================
 strange→False→False

 trivial.
Qed.
```

Pourquoi la même preuve n'est elle pas possible pour un type récursif bien construit ? Observons par exemple ce qui se passe si nous tentons la même preuve pour le type `nat`, pourtant très similaire puisqu'il ne présente qu'un constructeur de plus :

```
Theorem nat_not_strange : ∀n:nat, False.
Proof.
```

Effectuons le même premier pas de démonstration :

```
intros x; elim x.
 ...
 x : nat
 ============================
  False

subgoal 2 is:
 nat→False→False
```

Le deuxième but est toujours aussi facile à démontrer, mais le premier est insoluble, ce qui est rassurant.

Remarque. Le type `strange` est un exemple de type dont tous les éléments, s'ils existaient, seraient infinis. On peut avoir l'usage d'une telle structure de donnée. Les outils présentés dans le chapitre 14 permettent de considérer un type ayant le même constructeur mais contenant des objets infinis.

Exercice 7.50 Prouver les deux propositions suivantes, *apparemment* contradictoires :
- `∀x y : Empty_set, x=y`
- `∀x y : Empty_set, x ≠ y`

7.6.2 Dépendance et types vides

Pour un type inductif dépendant, il peut arriver qu'il n'existe pas d'éléments dans le type pour certaines valeurs des arguments. Suivant l'isomorphisme de Curry-Howard, le fait qu'un type soit vide ou non est le témoin d'une information logique importante, puisque ce type représente alors une formule fausse ou non.

Nous verrons dans le chapitre 9 tout le bénéfice qui peut être tiré de cette capacité à représenter des informations logiques.

Dans cette section, nous allons introduire les concepts progressivement, en nous reposant sur une définition inductive particulière. En fait, une telle définition inductive n'est jamais utilisée dans le système *Coq*, car l'utilisation qui y est faite des sortes `Prop` ou `Set` va à l'encontre de l'utilisation intuitive du type obtenu. Néanmoins, cette présentation a un intérêt pédagogique dans la mesure où elle montre la continuité qui existe naturellement entre les types récursifs de ce chapitre et les propriétés inductives du chapitre 9.

Nous définissons ici un type inductif dépendant d'une variable `n` qui ne contient des éléments que pour certaines valeurs de `n`.

```
Inductive even_line : nat→Set :=
  | even_empty_line : even_line 0
  | even_step_line : ∀n:nat, even_line n → even_line (S (S n)).
```

Si l'on observe certains des habitants de ce type, on s'aperçoit que les plus simples ont la forme suivante [5] :

```
Check even_empty_line.
even_empty_line : even_line 0
Check (even_step_line _ even_empty_line).
even_step_line 0 even_empty_line : even_line 2

Check (even_step_line _ (even_step_line _ even_empty_line)).
even_step_line 2 (even_step_line 0 even_empty_line)
    : even_line 4
```

Apparemment, si “ `even_line` n ” possède un habitant, alors n est nécessairement un nombre pair. Cette remarque indique que les types inductifs permettent d'exprimer des propriétés logiques, ce que nous approfondirons dans le chapitre 9.

5. On notera en passant comment l'inférence de type permet de résoudre les arguments non fournis (jokers.)

Chapitre 8

Tactiques et automatisation

Ce chapitre contient deux parties. La première partie présente plusieurs catégories de tactiques spécialisées dans des domaines variés de la logique et des mathématiques : des tactics spécialisées dans le raisonnement sur les types inductifs, les tactiques automatiques principales, qui reposent sur une recherche de preuve à la *Prolog*, les tactiques sur l'égalité et la réécriture, les tactiques pour les raisonnements numériques, et les procédures de décision pour des fragments restreints de la logique.

Dans une second partie du chapitre, nous présentons un langage pour programmer de nouvelles tactiques. En fait, certaines des procédures de décisions ont été programmées à l'aide de ce langage. Nous verrons d'autres exemples d'utilisation dans le chapitre 17.

8.1 Les tactiques des types inductifs

Dans le chapitre précédent, nous avons introduit les types inductifs en leur associant seulement cinq tactiques. La tactique `case` permet d'effectuer des traitement par cas, la tactique `simpl` permet de provoquer des conversions, et la tactique `elim` permet d'effectuer des raisonnements par récurrence. Les tactiques `discriminate` et `injection` permettent de vérifier que les constructeurs construisent des termes distincts et sont injectifs.

Ces tactiques fournissent un pouvoir expressif suffisant pour mener à bien la majorité des démonstrations portant sur les types inductifs. Néanmoins, le système fournit un certain nombre de tactiques plus élaborées qui sont en fait bâties au-dessus de ces tactiques. Nous allons énumérer ici plusieurs de ces tactiques, surtout parce qu'elles permettent de rendre les démonstrations plus concises et plus génériques.

8.1.1 Traitement par cas et récursion

La tactique `induction` peut être utilisée pour exprimer que l'on effectue une preuve par récurrence sur une expression qui n'est pas encore dans le contexte. En fait " `induction` *id* " est souvent équivalente à la tactique composée " `intros until` *id*`; elim` *id* ". Il est justifié d'utiliser cette tactique lorsque l'on veut rendre explicite le fait que l'on fait une démonstration par récurrence par rapport à une variable quantifiée universellement : le script de la preuve est alors très proche du texte que l'on écrirait pour exprimer comment la preuve s'effectue.

Par exemple, la démonstration que zéro est élément neutre à droite pour l'addition des nombres naturels est effectuée aisément de la façon suivante :

```
Open Scope nat_scope.
Theorem plus_n_O' : ∀n:nat, n + 0 = n.
 Proof.
  induction n; auto.
Qed.
```

Les avis sur l'utilisation de `induction` sont partagés, la lisibilité des scripts de démonstration est accrue, mais cette tactique introduit des hypothèses dans le contexte sans laisser l'utilisateur choisir le nom de ces hypothèses. Cela mène à un script contenant des informations partielles, plus difficile à maintenir (voir section 4.7). Nous conseillons de faire précéder cette tactique par une tactique `intros` pour donner explicitement les noms des hypothèses lorsque la démonstration exige de faire référence aux hypothèses dans les étapes ultérieures.

Lorsque *id* est un nom déjà présent dans le contexte du but, la tactique `induction` a un comportement encore plus complexe, qui conduit à introduire encore plus d'hypothèses dans le but. Résumé en quelques mots, la tactique " `induction` *id* " effectue encore un raisonnement par récurrence sur *id* (comme " `elim` *id* "), mais les hypothèses créées par le raisonnement par récurrence (qui correspondent donc aux prémisses pour chaque cas) sont également introduites systématiquement. Ce comportement complexe est intéressant car il permet de distinguer entre les hypothèses qui proviennent du raisonnement par récurrence et les hypothèses qui proviennent des prémisses du but, mais il est trop complexe pour que nous le décrivions ici, le lecteur intéressé devra se reporter au manuel de référence de *Coq*.

Pour illustrer cette variante de la tactique, nous pouvons observer un début de démonstration d'un théorème simple choisi pour l'exemple :

```
Theorem le_plus_minus' : ∀n m:nat, m ≤ n → n = m+(n-m).
 Proof.
 intros n m H.
 ...
 n : nat
 m : nat
 H : m ≤ n
```

```
============================
 n = m+(n-m)

induction n.
...
H : m ≤ 0
============================
0 = m+(0-m)

 rewrite <- le_n_O_eq with (1 := H); simpl; trivial.
...
n : nat
m : nat
IHn : m ≤ n → n = m+(n-m)
H : m ≤ S n
============================
 S n = m+(S n - m)
```

Cet exemple montre que dans les deux buts engendrés par `induction`, non seulement la conclusion du but est modifiée, mais aussi le contexte, de plus l'hypothèse de récurrence est nommée de façon à être reconnue.

Dans les exemples de cet ouvrage, nous utilisons également une tactique `destruct` qui est à `case` ce qu'`induction` est à `elim`.

8.1.2 Conversions

8.1.2.1 Tactiques de base

Nous avons vu que les types inductifs fournissent également des moyens de calcul, qui peuvent être dirigés à l'aide des tactiques `simpl` et `change`. La tactique `simpl` n'effectue que des réductions, et seulement lorsque ces réductions suivent le schéma récursif des fonctions récursives. La tactique `change` permet de remplacer l'énoncé du but par un énoncé convertible, ce qui est beaucoup plus puissant mais également plus contraignant parce que l'utilisateur doit fournir ce nouvel énoncé (voir un exemple en section 7.2.3).

La tactique `simpl` peut prendre des paramètres pour spécifier la sous-expression du but que l'on veut réduire ; ce paramètre est construit de la même manière que l'argument de la tactique `pattern` que nous avons déjà utilisée dans les sections 6.3.3 page 155 et 7.2.7 page 185. Voici un exemple d'utilisation de cette variante :

```
Theorem simpl_pattern_example :  3*3 + 3*3 = 18.
Proof.
 simpl (3*3) at 2.
 ...
 ============================
```

*3*3 + 9 = 18*

8.1.2.2 Stratégies de conversion

Les tactiques `lazy` et `cbv` permettent de provoquer une réduction systématique du but, comme la tactique `simpl`, mais elle peuvent recevoir d'autres paramètres qui permettent de préciser quelles réductions sont effectuées. Tout d'abord, ces deux tactiques diffèrent par la stratégie employée. La tactique `lazy` utilise la stratégie paresseuse, où seules les expressions réellement nécessaires pour déterminer le résultat final sont réduites. La tactique `cbv` utilise la stratégie d'appel par valeur, qui veut que tout argument d'une fonction est réduit avant la réduction correspondant à l'appel de cette fonction. Aucune des deux stratégies n'est toujours meilleure que l'autre. La stratégie paresseuse peut être avantageuse lorsque l'on cherche à réduire une expression qui contient un terme important mais inutile. C'est souvent le cas lorsque l'on veut utiliser une fonction qui construit à la fois une valeur et un certificat. Nous donnons un exemple en section 16.1.

Les paramètres donnés à `lazy` et `cbv` permettent d'indiquer le type de réduction à effectuer : ces paramètres peuvent être les suivants :

- `beta` pour réduire les expressions de la forme “ `(fun x:T ⇒ e) v` ”,
- `delta` pour réduire les définitions de constantes et fonctions,
- `iota` pour réduire les expressions de filtrage et les fonctions récursives,
- `zeta` pour réduire les expressions de la forme “ `let x := v in e` ”.

En outre, l'argument `delta` peut-être modifié par une liste d'identificateurs qui précise les noms des fonctions à développer. Par exemple, nous pouvons continuer la réduction de l'expression du but précédent en demandant que seule la multiplication soit réduite :

```
Show 1.
```

```
 ...
 ============================
  3*3 + 9 = 18
  lazy beta iota zeta delta [mult].
```

```
 ...
 ============================
   3+(3+(3+0))+9 = 18
```

Il arrive que `lazy` ou `cbv` développe les fonctions récursives et les remplace par les fonctions récursives anonymes obtenues à l'aide de la construction `fix` correspondantes, ce qui rend les buts illisibles. L'exemple suivant montre ce défaut :

```
Theorem lazy_example : ∀n:nat, (S n) + 0 = S n.
Proof.
 intros n; lazy beta iota zeta delta.
```

```
...
n : nat
============================
 S
   ((fix plus (n0:nat) : nat→nat :=
       fun m:nat ⇒ match n0 with
                    | O ⇒ m
                    | S p ⇒ S (plus p m)
                    end) n 0) = S n
```

Il est alors utile d'appliquer la tactique `fold`, qui replie la définition des fonctions ainsi maltraitées par la δ-réduction :

```
fold plus.
...
============================
 S (n+0) = S n
```

8.2 Les tactiques auto et eauto

La tactique `auto` est l'un des outils de démonstration automatique les plus faciles à mettre en œuvre, c'est d'ailleurs la principale procédure de démonstration automatique utilisée dans cet ouvrage et nous l'avons introduite dès la section 4.9.2. Le principe en est simple : `auto` utilise une base de données de tactiques, lesquelles sont appliquées au but initial, ainsi qu'à tous les sous-buts engendrés, et ce répétitivement, jusqu'à la résolution de tous les buts. Si ces buts ne peuvent être tous résolus, `auto` annule tous ses efforts en laissant le but initial inchangé. On peut imaginer que ce procédé construit effectivement des arbres, dont chaque noeud correspond à l'application d'une tactique et les sous-arbres d'un nœud donné correspondent aux tactiques appliquées pour résoudre les sous-buts de la tactique associée à ce nœud. La tactique `auto` reçoit un paramètre numérique qui permet de limiter la hauteur de l'arbre ainsi construit. Par défaut, cet argument numérique est 5.

Avant toute opération, `auto` commence par placer le but dans une forme normale, où le maximum d'introductions sans δ-réduction est effectué. Par exemple, si le but est de la forme $\Gamma \overset{?}{\vdash} B {\rightarrow} {\sim} A$ il est d'abord transformé en $\Gamma, B \overset{?}{\vdash} {\sim} A$, mais pas en $\Gamma, B, A \overset{?}{\vdash}$ `False`. La tactique `auto` considère toutes les hypothèses du contexte courant comme arguments possibles d'`apply`.

Les bases de tactiques sont nommées par un identificateur. Le système utilise quelques bases pré-définies, dont la base `core`. L'utilisation par `auto` des bases de tactiques b_1, b_2, $\ldots b_n$ se fait par la tactique " `auto with` b_1 b_2 `...`b_n ".

8.2.1 Bases de tactiques : Hints

Formellement, les éléments des bases de données pour `auto` sont des tactiques, mais la commande la plus fréquemment utilisée pour ajouter un élément mentionne des théorèmes. Il s'agit de la commande

`Hint Resolve` *thm_1* `...` *thm_k* `:` *database*.

Un coût est associé à chaque tactique, ce qui permet d'intervenir sur l'ordre dans lequel les tactiques sont appliquées. Lorsque la tactique a été ajoutée dans la base de donnée à l'aide de `Hints Resolve`, le coût associé est le nombre de prémisses du théorème, ce qui correspond en fait au nombre de buts engendrés.

Les conditions pour qu'un théorème soit effectivement appliqué via `auto` sont les mêmes que celles requises par une utilisation manuelle de `apply` (voir section 6.1.3 page 136). Voici un exemple de théorème qui satisfait les conditions nécessaires :

le_n_S : $\forall$ *n m:nat*, $n \leq m \rightarrow S\ n \leq S\ m$

En effet, ce théorème contient deux variables quantifiées universellement, `n` et `m`, qui apparaissent toutes les deux dans la tête du théorème " `S n` $\leq$ `S m` ". En revanche, le théorème suivant ne satisfait pas ces conditions :

le_trans : $\forall$ *n m p:nat*, $n \leq m \rightarrow m \leq p \rightarrow n \leq p$

En effet, ce théorème contient une variable quantifiée universellement, `m` qui n'apparaît pas dans la tête du théorème " `n` $\leq$ `p` ". Aucune valeur pour `m` ne peut donc être déterminée au moment de l'application de `le_trans`.

En pratique, ceci signifie que les théorèmes utilisés par la tactique `apply` dans le comportement de `auto` ne doivent pas faire appel à la recherche d'un témoin. Néanmoins, les théorèmes qui ne satisfont pas cette contrainte sont quand même ajoutés dans les bases de données de théorèmes et ils seront seulement utilisés par la tactique `eauto` (voir section 8.2.2). On peut introduire d'autres tactiques que la tactique `apply` dans les bases de tactiques pour `auto`. La documentation de *Coq* donne l'exemple où la tactique `discriminate` est ajoutée pour être utilisée à chaque fois que l'occasion s'en présente :

`Hint Extern 4 (_` $\neq$ `_)` $\Rightarrow$ `discriminate : core.`

Les deux premiers arguments de cette commande `Hint` s'utilisent comme pour la commande " `Hint ... Resolve` ", à savoir un nom pour l'entrée et un nom pour la base de tactiques dans laquelle cette entrée est introduite. Ensuite vient le mot-clef `Extern`, puis viennent un nombre, un motif d'expression et une tactique. Le nombre est le coût, ce qui permet à l'utilisateur de choisir si cette tactique sera utilisée en priorité ou non. Le motif indique de façon précise ceux des buts pour lesquels cette tactique sera tentée.

Cette variante de la commande `Hint` peut aussi être utilisée pour introduire des tactiques `apply`, mais dans ce cas elle permet de limiter les cas d'application du théorème, en fournissant un motif plus restrictif, par exemple, ou de changer la priorité de cette entrée. Le changement de priorité peut changer la rapidité avec laquelle `auto` termine lorsque cette tactique réussit à résoudre le but, mais ne changera pas la rapidité avec laquelle elle termine lorsque le but n'est pas résolu. En effet, dans ce cas toutes les combinaisons de tactiques sont tentées jusqu'à la profondeur requise, ce qui prendra toujours le même temps quel que soit l'ordre dans lequel les tactiques sont essayées.

**** Considérations d'efficacité**

Certains théorèmes, pourtant bien formés pour être utilisés par `auto` sont de mauvais candidats pour être ajoutés dans les bases de tactiques. Ce sont des théorèmes qui introduisent des boucles de raisonnement. Un exemple simple de mauvais candidat est le théorème suivant :

sym_equal : $\forall$ *(A:Type)(x y:A),* $x = y \to y = x$

En effet, la tactique " `apply sym_equal` " produit un but sur lequel la même tactique peut encore s'appliquer, mais cette deuxième application retourne le but initial, ce dont `auto` ne s'aperçoit pas.

Parfois, le nouveau but engendré n'est pas le même, mais pourtant l'application répétitive du théorème mène quand même à un raisonnement non concluant, ce que l'on peut voir avec cet autre mauvais candidat :

le_S_n : $\forall$ *n m:nat,* $S\ n \leq S\ m \to n \leq m$

Pourtant, il est parfois bien utile d'ajouter un tel théorème si l'on sait qu'il permettra de résoudre certains buts. Dans ce cas il est judicieux d'utiliser sa propre base de tactiques.

Supposons par exemple que nous soyons en présence du but suivant :

...
H : $S\ (S\ (S\ n)) \leq S\ (S\ (S\ m))$
============================
$n \leq m$

Trois applications du théorème `le_S_n` et l'hypothèse `H` permettent de résoudre ce but. On s'en sortira automatiquement de la façon suivante :

NEW: Merci Gérard : difficulté : le nom "le" sert à la fois pour le prédicat et la base ; trouver une autre nom, genre "le_base" "le_hints" par exemple

```
Hint Resolve le_S_n : le_base.
 auto with le_base.
```
Proof completed.

La tactique "`auto with le`" n'applique que le théorème `le_S_n` et les hypothèses du contexte, jusqu'à ce que d'autres commandes `Hint` ajoutent des tactiques dans la base de tactiques `le`, elle aura donc une complexité relativement faible. Il peut être intéressant d'utiliser cette tactique pour résoudre tous les buts engendrés par une autre tactique dans une tactique composée de la forme " *tactique*`;auto with le` ".

Il faut néanmoins être prudent lorsqu'il s'agit de combiner la base de tactiques `le` avec d'autres bases de tactiques pour `auto`, comme la base de tactiques `arith`.

Considérons par exemple les deux tactiques composées suivantes :

1. *tac* `; auto with le;auto with arith`
2. *tac* `; auto with le arith`

Dans le premier cas, les buts laissés non résolus par *tac* font l'objet d'une exploration complète d'un arbre de recherche, par “ `auto with le` ” d'abord, puis par “ `auto with arith` ”. Quand la démonstration automatique échoue, les arbres de recherche pour la première tactique `auto` auront la hauteur maximale autorisée, parce que la même tactique peut se répéter plusieurs fois, mais il n'y aura qu'une branche, ce qui implique un coût minime. Les arbres de recherche pour la tactique “ `auto with arith` ” sont de taille arbitraire, mais habituellement leur exploration prend un temps raisonnable.

Dans le second cas, les buts laissés non résolus par *tac* font encore l'objet d'une exploration complète d'un arbre de recherche, mais cet arbre de recherche a probablement une taille exponentiellement plus grande que celui exploré par la tactique “ `auto with arith` ” seule. En effet, après chaque application du théorème `le_S_n` la tactique “ `auto with le arith` ” pourra appliquer le théorème `le_n_S` pour revenir au même but pour lequel le même arbre de recherche devra être exploré, potentiellement avec un branchement si d'autres théorèmes s'appliquent pour le même but.

Voici un exemple de tentative de démonstration automatique qui montre que l'efficacité de la tactique `auto` se dégrade fortement lorsque l'on applique cette tactique sur un but insoluble en mélangeant des bases de tactiques incompatibles :

```
Lemma unprovable_le : ∀n m:nat, n ≤ m.
 Time auto with arith.
  ...
Finished transaction in  0. secs (0.u,0.s)

 Time auto with le_base arith.
  ...
Finished transaction in  1. secs (0.44u,0.s)

Abort.
```

Il ne s'agit ici que d'un but très simple et le temps passé est déjà non négligeable. Lorsque le but présente plusieurs hypothèses, ce temps peut devenir inacceptable. Quand un fait est présent parmi les hypothèses du but, il est difficile d'éviter que la tactique `auto` utilise ce fait. Dans ce cas, il est judicieux d'enlever les hypothèses gênantes à l'aide de la tactique `clear`, comme le montre l'exemple suivant :

```
Section Trying_auto.
Variable l1 : ∀n m:nat, S n ≤ S m → n ≤ m.
Theorem unprovable_le2 : ∀n m:nat, n ≤ m.

Time auto with arith.
  ...
  ============================
   ∀n m:nat, n ≤ m
```

Finished transaction in 0. secs (0.48u,0.01s)

```
Time try (clear l1; auto with arith; fail).
```

  ...
  ============================
   $\forall$ *n m:nat*, $n \leq m$
Finished transaction in 0. secs (0.01u,0.s)

L'utilisation des tactiques `try` et `fail` permet d'assurer que l'hypothèse `l1` sera conservée pour les buts que la tactique `auto with arith` laisse ouverts. L'exemple donné en section 4.6 page 91 utilise également un échec provoqué.

8.2.2 * La tactique `eauto`

Nous avons indiqué précédemment que le théorème `le_trans` n'était pas adapté pour une utilisation par la tactique élémentaire `apply` et qu'il ne serait donc pas utilisé par la tactique `auto`. Il existe une variante de la tactique `auto` qui utilise plutôt la commande `eapply` et permet donc d'utiliser les théorèmes de la même forme que `le_trans`, c'est-à-dire des théorèmes pour lesquels il peut être nécessaire de deviner des témoins (voir page 140). Pour cette raison, elle explore des espaces de recherche souvent bien plus grands que ceux explorés par la tactique `auto` et est souvent beaucoup plus lente. Elle est donc beaucoup moins utilisée.

Il y a une forte analogie entre les tactiques `auto` et `eauto` et le moteur d'inférence d'un interprète *Prolog* : le type de tête d'un théorème correspond à la tête d'une clause *Prolog*. La tactique `auto` correspond à un interprète *Prolog* qui ne permettrait pas la création de nouvelles variables logiques lors de l'application d'une clause.

8.3 Les tactiques numériques

Le système *Coq* fournit trois catégories principales de nombres : les nombres naturels, les nombres entiers et les nombres réels. Les nombres naturels sont intéressants car il fournissent une structure de récursion simple et interviennent dans la problématique de la taille des données et de la combinatoire. Les nombres entiers sont intéressants car il fournissent une structure algébrique claire, la structure d'anneau et aussi parce qu'il reposent sur un codage binaire, ce qui permet une implémentation efficace des principales opérations de base. Les nombres réels ne sont pas décrits de façon définitionnelle ou par un type inductif, mais par un ensemble d'axiomes. En fait, l'approche définitionnelle usuelle ne permettrait pas d'obtenir un ensemble de nombres réels muni d'une relation d'égalité « classique ». Pour ces trois catégories de nombres, on dispose également d'un ordre total. Plusieurs tactiques sont fournies pour décider l'égalité de certaines classes de formules et la satisfiabilité de certains systèmes d'inéquations.

8.3.1 La tactique `ring`

La tactique `ring` utilise la technique de preuve par réflexion décrite dans [15] et que nous détaillerons plus précisément dans le chapitre 17. Elle est très bien adaptée pour résoudre des équations polynômiales dans un anneau ou un semi-anneau. Pour la mettre en œuvre, il est nécessaire de déclarer la structure d'anneau à l'aide des commandes “ `Add Ring` ” ou “ `Add Semi Ring` ”. Par exemple, ces commandes sont utilisées dans les modules `ZArithRing` et `ArithRing` pour la structure d'anneau du type des entiers relatifs `Z` et pour la structure de semi-anneau du type des entiers naturels `nat`, respectivement.

La tactique `ring` marche très bien pour démontrer des égalités où les membres sont des expressions construites avec une fonction d'addition, une fonction de multiplication et des valeurs $x_1, \ldots x_n$. Elle ne marche pas si des fonctions ou des constructeurs sont insérés au milieu des expressions. Voici quelques exemples de réussite et d'échec de cette tactique, dont certains sont dûs à l'utilisation d'une fonction `square` que nous définissons pour l'exemple.

```
Open Scope Z_scope.

Theorem ring_example1 : ∀x y:Z, (x+y)*(x+y)=x*x + 2*x*y + y*y.
Proof.
 intros x y; ring.
Qed.

Definition square (z:Z) := z*z.

Theorem ring_example2 :
  ∀x y:Z, square (x+y) = square x + 2*x*y + square y.
Proof.
 intros x y; ring.
 ...
============================
   square (x+y) = square y + (2*(y*x) + square x)
 unfold square; ring.
Qed.

Theorem ring_example3 :
  (∀x y:nat, (x+y)*(x+y) = x*x + 2*x*y + y*y)%nat.
Proof.
 intros x y; ring.
Qed.
```

Lorsqu'elle ne parvient pas à démontrer l'égalité, la tactique `ring` n'échoue pas, mais elle engendre une nouvelle égalité.

```
Theorem ring_example4 :
 (∀x:nat, (S x)*(x+1) = x*x + (x+x+1))%nat.
```

```
 Proof.
 intro x; ring.
 ...
 ============================
  (x * S x + S x)%nat = (1+(2*x + x*x))%nat
```

Le problème vient ici de ce que la tactique `ring` n'a pas su reconnaître le fait que " `S x` " est bien une expression polynômiale, équivalente à " `x + 1` " ; de même il est également nécessaire que `2` soit reconnu comme un nombre équivalent à " `1 + 1` ". La tactique `ring_nat` est proposée dans les bibliothèques de *Coq* pour résoudre ce genre de problème.

```
 ring_nat.
Qed.
```

Nous expliquons en section 8.5.2.2 comment construire une tactique qui effectue la reconnaissance nécessaire. Cette tactique est combinée avec `ring` pour donner la tactique `ring_nat`

La tactique `ring` n'utilise pas les égalités présentes dans le contexte pour établir son résultat ; si l'on veut établir une démonstration qui utilise les égalités du contexte, il est nécessaire de s'assurer que les réécritures adéquates ont été effectuées avant l'appel de cette tactique.

8.3.2 La tactique omega

La tactique `omega` implémente un algorithme proposé par Pugh [80]. Elle est très puissante pour résoudre des systèmes d'équations et d'inéquations linéaires seulement sur le type `Z` des entiers relatifs et sur le type `nat`. Elle fonctionne en utilisant toutes les informations qu'elle peut trouver dans le contexte du but courant. Voici un exemple de démonstration effectuée avec cette tactique :

```
Require Import Omega.

Theorem omega_example1 :
 ∀x y z t:Z, x ≤ y ≤ z ∧  z ≤ t ≤ x → x = t.
Proof.
 intros x y z t H; omega.
Qed.
```

Ce qui est remarquable dans cet exemple, c'est que `omega` va chercher les informations pertinentes dans toutes les hypothèses, même à l'intérieur des hypothèses si celles-ci sont des conjonctions.

Les inéquations doivent être linéaires, ce qui veut dire que les variables ou expressions peuvent apparaître multipliées à des constantes mais ne doivent pas être multipliées à d'autres variables. Dans l'exemple suivant, nous utilisons la fonction `square` définie en section 8.3.1 page 226 et le terme " `square x` " est considéré comme une boîte noire, et les inéquations considérées comme linéaires :

```
Theorem omega_example2 :
 ∀x y:Z,
    0 ≤ square x → 3*(square x) ≤ 2*y → square x ≤ y.
 Proof.
 intros x y H H0; omega.
Qed.
```

Lorsque les inéquations ne sont pas linéaires, `omega` tente de les assimiler à des inéquations linéaires dont les paramètres sont les expressions non-linéaires, ce qui peut permettre de résoudre certains buts, comme dans l'exemple suivant, où l'expression non linéaire “ `x*x` ” est bien reconnue par `omega`.

```
Theorem omega_example3 :
 ∀x y:Z,
   0 ≤ x*x → 3*(x*x) ≤ 2*y → x*x ≤ y.
Proof.
 intros x y H H0; omega.
Qed.
```

Cet exemple a bien pu être résolu par `omega`, car cette tactique a pu reconnaître un facteur commun “ `x * x` ” dans l'énoncé à prouver. Si l'on remplace ce facteur par une simple variable `X`, on obtient bien une combinaison d'inéquations linéaires en `X` et `y`.

```
0 ≤ X → 3*X ≤ 2*y  →  X < y.
```

En revanche, le théorème suivant, pourtant équivalent au précédent, n'est pas résolu par `omega`. En effet, le parenthésage implicite utilisé par *Coq* fait que l'écriture “ `3*x*x` ” est une abréviation de “ `(3*x)*x` ” dans laquelle le facteur “ `x*x` ” n'apparaît pas directement ; c'est seulement grâce aux propriétés d'associativité de la multiplication que la nouvelle expression est équivalente à “ `3*(x*x)` ” et la tactique `omega` n'est pas programmée pour mettre en œuvre cette associativité.

```
Theorem omega_example4 :
 ∀x y:Z, 0 ≤ x*x → 3*x*x ≤ 2*y → x*x ≤ y.
Proof.
 intros x y H H0; omega.
```
Error: omega can't solve this system

La tactique `omega` peut être utilisée avec beaucoup de succès pour résoudre des problèmes de programmation linéaire entière. Par exemple, nous avons participé à une étude sur l'implémentation efficace en espace d'un algorithme de racine carrée où cette tactique était particulièrement utile pour résoudre des problèmes de recouvrement d'intervalles [13].

Parce qu'elle utilise potentiellement toutes les hypothèses présentes dans le contexte, la tactique `omega` peut devenir très lente lorsque le contexte contient beaucoup d'hypothèses. Il peut être judicieux pour l'utilisateur de détruire par `clear` toutes les hypothèses du contexte qui ne servent à rien avant de faire appel à la tactique `omega`. Nous donnons en section 8.5.2.2 une méthode pour construire une tactique qui effectue ce genre d'opération.

8.3.3 La tactique `field`

La tactique `field` fournit la même fonctionnalité que la tactique `ring` mais pour une structure de corps, en considérant également des opérations de division. Pour toutes les simplifications concernant les divisions, cette tactique engendre une obligation de preuve supplémentaire pour assurer que le diviseur est non nul. Voici un exemple :

```
Require Import Reals.

Open Scope R_scope.

Theorem example_for_field : ∀x y:R, y ≠ 0 →(x+y)/y = 1+(x/y).
 Proof.
 intros x y H; field.
  ...
  H : y ≠ 0
  ============================
    y ≠ 0
 trivial.
Qed.
```

8.3.4 La tactique `fourier`

La tactique `fourier` fournit la même fonctionnalité que `omega` mais pour les nombres réels. Les inéquations considérées doivent se ramener à des inéquations linéaires à coefficients rationnels [44]. En voici un exemple.

```
Require Import Fourier.

Theorem example_for_Fourier : ∀x y:R, x-y>1→x-2*y<0→x>1.
 Proof.
  intros x y H H0.
  fourier.
Qed.
```

Il est plus complexe de résoudre des systèmes d'inéquations polynomiales, mais les travaux décrits dans [63] permettent d'espérer une solution prochaine.

8.4 Procédures de décision pour la logique propositionnelle

Le système *Coq* fournit également une procédure de décision pour les tautologies propositionnelles intuitionnistes, qui s'appelle `tauto`. Cette procédure de décision permettra de prouver des formules logiques qui ne peuvent pas être prouvées par `auto` car elle utilise mieux les hypothèses du contexte. En particulier, si le contexte contient des conjonctions ou des disjonctions, la tactique `auto` ne les utilisera probablement pas, tandis que `tauto` saura les utiliser. Elle s'applique également à des formules qui ne sont pas des formules de logique propositionnelle mais qui sont des instances de formules prouvables de logique propositionnelle intuitionniste. Voici une collection de formules logiques qui sont démontrées par `tauto` et non par `auto`.

```
∀A B:Prop, A∧B→A

∀A B:Prop, A∧~A → B

∀x y:Z, x≤y → ~(x≤y) → x=3

∀A B:Prop, A∨B → B∨A

∀A B C D:Prop, (A→B)∨(A→C)→A→(B→D)→(C→D)→D
```

La tactique `intuition` *tac* permet d'enchaîner le travail fait par `tauto` en logique propositionnelle avec une tactique *tac* permettant de traiter les sous-buts laissés non résolus par `tauto`. Considérons par exemple le but suivant :

```
Open Scope nat_scope.

Theorem example_intuition :
  (∀n p q:nat,  n ≤ p ∨ n ≤ q → n ≤ p ∨ n ≤ S q).
Proof.
```

Un essai avec “ `auto with arith` ” laisse ce but inchangé, car cette tactique n'élimine pas la première disjonction. La tactique “ `intros n p q;tauto` ” échoue également, car incapable de traiter la corrélation entre les propositions “ `n ≤ q` ” et “ `n ≤ S q` ”.

En revanche, la tactique “ `intros n p q;intuition auto with arith` ” réussit immédiatement : une analyse similaire à celle de `tauto` permet de traiter l'élimination de la première disjonction et le traitement du cas “ `n ≤ p` ”; il reste un sous-but correspondant au cas “ `n ≤ q` ”, lequel est traité par “ `auto with arith` ”.

La tactique `intuition` peut s'utiliser sans paramètre ; c'est alors une abréviation de “ `intuition auto with *` ” (`auto` utilise toutes les bases de théorèmes existantes.)

8.5 ** Le langage de définition de tactiques

Le système *Coq* fournit également un langage de définition de tactiques appelé $\mathcal{L}$tac. Ce langage permet d'écrire des tactiques paramétrées et récursives sans faire appel à la programmation directe en *OCAML*, ce qui imposerait une connaissance très précise des structures de données internes du système de preuve. Le langage $\mathcal{L}$tac fournit des structures de contrôle originales et peu de structures de données, ce qui rend la programmation dans ce langage assez exotique. Ce langage est récent [37] et sa syntaxe et sa sémantique sont probablement instables. En revanche, les structures de contrôle permettent une programmation très concise d'algorithmes de recherche de démonstrations, potentiellement avec des fonctions dont la terminaison n'est pas assurée.

8.5.1 Liaison de paramètres

L'un des premiers avantages de l'utilisation du langage $\mathcal{L}$tac est la possibilité d'attacher un nom court à des opérations parfois complexes. Ainsi, nous avons vu en section 8.2.1, qu'il pouvait être nécessaire d'encapsuler la commande `auto` pour lui assurer une certaine efficacité. On écrira alors la commande suivante pour définir une nouvelle tactique qui effectue cette encapsulation :

```
Ltac autoClear h := try (clear h; auto with arith; fail).
```

L'argument `h` précise le nom de l'hypothèse à enlever du contexte courant.

L'argument donné à une tactique peut appartenir à plusieurs catégories syntaxiques. Il peut être un identificateur, comme dans le cas de la tactique `autoClear`, il peut être une expression du Calcul des Constructions, comme dans le cas de la tactique `caseEq` (section 7.2.7.), il peut aussi être une tactique. Par exemple, la tactique suivante permet de généraliser la tactique `autoClear` en laissant l'utilisateur choisir la tactique qui sera employée avec la tactique `auto`.

```
Ltac autoAfter tac := try (tac; auto with arith; fail).
```

L'exemple suivant montre une utilisation simple de la tactique autoAfter. Dans le but suivant, les hypothèses `H0` et `H1` sont inutiles, et l'on souhaite les effacer avant l'appel à auto.

...
n : nat
p : nat
H : $n < p$
H0 : $n \leq p$
H1 : $0 < p$
============================
$S\ n < S\ p$

```
 autoAfter ltac:(clear H0 H1).
Qed.
```

La tactique “ `autoAfter '(clear H0 H1)` ” résout immédiatement ce but. On remarquera la présence de l’apostrophe en tête de l’argument fourni à autoAfter ; cette convention permet d’éviter la confusion entre un argument qui est une « tactique » et un argument qui est un terme du Calcul des Constructions. Dans ce dernier cas, on n’utilise pas l’apostrophe. Par exemple, un appel de `caseEq` avec pour argument le terme “ `S n` ” s’écrit simplement “ `caseEq (S n)` ”.

Les tactiques ainsi définies peuvent également être récursives. La tactique suivante applique répétitivement les théorèmes `le_n` et `le_S` pour démontrer qu’un nombre naturel arbitraire est inférieur à un autre nombre :

```
Open Scope nat_scope.

Ltac le_S_star := apply le_n || (apply le_S; le_S_star).

Theorem le_5_25 : 5 ≤ 25.
Proof.
 le_S_star.
Qed.
```

8.5.2 Constructions de filtrage

Filtrage dans le but

Pour simplifier l’écriture de tactiques automatiques, il est souvent utile de choisir l’argument d’une tactique en retrouvant cet argument dans le but.

Par exemple, on peut s’intéresser au problème de la démonstration qu’un nombre est premier. Afin de simplifier le travail de mise au point, nous déclarons comme hypothèses des propriétés arithmétiques que nous pourrons démontrer ultérieurement (voir exercice 8.1.)

Voir correction dans version anglaise

```
Section primes.

Definition divides (n m:nat) := ∃p:nat, p*n = m.

Hypotheses
  (divides_0 : ∀n:nat, divides n 0)
  (divides_plus : ∀n m:nat, divides n m → divides n (n+m))
  (not_divides_plus : ∀n m:nat, ~divides n m →
                                ~divides n (n+m))
  (not_divides_lt : ∀n m:nat, 0<m → m<n → ~divides n m)
  (not_lt_2_divides : ∀n m:nat, n≠1 → n<2 → 0<m → ~divides n m)
  (le_plus_minus : ∀n m:nat, le n m → m = n+(m-n))
  (lt_lt_or_eq : ∀n m:nat, n < S m →  n<m ∨ n=m).
```

Pour vérifier qu'un nombre entier n n'est pas divisé par un nombre p, nous pourrons effectuer la soustraction de p autant que possible, jusqu'à ce que p soit plus grand que le résultat. Si le résultat est non nul, cela permet rapidement de conclure. Cette méthode est décrite par la tactique suivante :

```
Ltac check_not_divides :=
  match goal with
  | [ |- (~divides ?X1 ?X2) ] ⇒
      cut (X1≤X2);[ idtac | le_S_star ]; intros Hle;
       rewrite (le_plus_minus _ _ Hle); apply not_divides_plus;
       simpl; clear Hle; check_not_divides
  | [ |- _ ] ⇒ apply not_divides_lt; unfold lt; le_S_star
  end.
```

La construction qui indique que l'on va effectuer un filtrage sur le but est reconnue par les mots-clefs “ `match context with` ”. Les schémas de filtrage sont de la forme

$$\texttt{[}\ h_1 : t_1\texttt{,}\ h_2 : t_2\ \ \texttt{...|-}\ C\ \texttt{]}\ \Rightarrow\ tac$$

Les expressions t_i et C sont des motifs de la même forme que ceux utilisés dans `SearchPattern` et `SearchRewrite` (voir section 6.1.3). Le filtrage n'est pas linéaire et la même variable numérotée, de la forme `?X`i peut apparaître plusieurs fois. Les hypothèses ainsi filtrées sont prises *parmi* les hypothèses du but courant et le motif de filtrage ne doit pas nécessairement décrire toutes les hypothèses du contexte. Les noms h_i peuvent être utilisés dans la tactique tac au même titre que les variables `X`i (sans le point d'interrogation) apparaissant dans les motifs t_i.

La première clause de cette tactique s'applique lorsque l'on veut vérifier qu'un nombre `n` ne divise pas un nombre `m` et que `n` est plus petit que `m`. La ligne

```
cut (X1≤X2);[ idtac| le_S_star]
```

permet d'assurer que le reste de la tactique ne sera appliqué qu'après que l'on aura vérifié que la proposition “ `X1 ≤ X2` ” est bien prouvable par `le_S_star`. Les lignes qui suivent (à partir de “ `intros Hle` ”) ne sont appliquées que sur une hypothèse dont on sait qu'elle a pu être prouvée par ailleurs. Par comparaison on aurait pu remplacer la tactique “ `idtac` ” qui ne fait rien par toute la tactique allant de “ `intros` ” à “ `check_not_divides` ”, mais ceci mènerait à un appel récursif indéfini de la tactique, sans jamais finir par le cas de base et en laissant une infinité de buts annexes à vérifier de la forme “ `n ≤ 0` ”.

La deuxième clause de l'expression de filtrage décrit le cas de base de notre tactique récursive.

Exercice 8.1 * Démontrer les théorèmes correspondant aux hypothèses de la section `primes`.

8.5.2.1 Utilisation des noms d'hypothèses

Il est également possible d'effectuer un filtrage parmi les hypothèses, en retrouvant le nom de l'hypothèse recherchée. Ceci peut être très utile pour retrouver une valeur utilisable parmi les faits du contexte sans faire appel à `eauto`. Un premier exemple utilisant cette possibilité est la tactique suivante, qui peut être utilisée pour faire des raisonnements par contraposée, c'est à dire de passer d'un but dont la conclusion est $\sim A$ et dont une hypothèse est de la forme $\sim B$ à un but dont la conclusion est B et une hypothèse affirme A ; le paramètre `H` est le nom à donner à l'hypothèse à introduire.

```
Ltac contrapose H :=
  match goal with
  | [id:(~_) |- (~_) ] ⇒ intro H; apply id
  end.
```

Voici un exemple utilisant cette tactique.

```
Theorem example_contrapose :
  ∀x y:nat, x ≠ y → x ≤ y → ~y ≤ x.
Proof.
 intros x y H H0.
 ...
 H : x ≠ y
 H0 : x ≤ y
 ============================
  ~y ≤ x

 contrapose H'.
 ...
 H : x≠y
 H0 : x ≤ y
 H' : y ≤ x
 ============================
  x=y

 auto with arith.
Qed.
```

Nous définissons également une tactique pour prouver que tout nombre m inférieur à un certain n ne divise pas p. Pour prouver que p est premier, il suffit d'appliquer cette tactique pour $n = p$. Cette tactique a une structure récursive et utilise `check_not_divides`.

```
Ltac check_lt_not_divides :=
  match goal with
  | [Hlt:(lt ?X1 2%nat) |- (~divides ?X1 ?X2) ] ⇒
```

```
      apply not_lt_2_divides; auto
  | [Hlt:(lt ?X1 ?X2) |- (~divides ?X1 ?X3) ] ⇒
      elim (lt_lt_or_eq _ _ Hlt);
       [clear Hlt; intros Hlt; check_lt_not_divides
        | intros Heq; rewrite Heq; check_not_divides]
  end.
```

Nous pouvons utiliser cette tactique pour construire la preuve qu'un nombre est bien premier, comme dans les sessions suivantes :

```
Definition is_prime : nat→Prop :=
  fun p:nat ⇒ ∀n:nat, n ≠ 1 → lt n p → ~divides n p.

Hint Resolve lt_O_Sn.

Theorem prime37 : is_prime 37.
Proof.
 Time (unfold is_prime; intros; check_lt_not_divides).
```

Proof completed
Finished transaction in 13. secs (13.05u,0.1s)

```
Time Qed.
```

...
prime37 is defined
Finished transaction in 13. secs (12.56u,0.02s)

```
Theorem prime61 : is_prime 61.
Proof.
 Time (unfold is_prime; intros; check_lt_not_divides).
```

Proof completed
Finished transaction in 68. secs (67.59u,0.23s)

```
Time Qed.
```

...
prime61 is defined
Finished transaction in 94. secs (94.51u,0.09s)

Le temps d'exécution de cette tactique montre que l'on ne peut pas envisager de l'utiliser pour un test de primalité. Il s'agit plutôt d'un exemple de construction d'arguments complets de primalité dans une théorie arithmétique simple.

Cet exemple montre la puissance expressive du langage $\mathcal{L}$tac dans la mesure où nous avons utilisé seulement 15 lignes pour définir les deux tactiques nécessaires pour construire une procédure de construction de preuve de primalité de nombres entiers et construit une preuve l'exprimant. Bien sûr il faut ajouter la démonstration des différentes hypothèses utilisées dans la section, mais ces démonstrations sont faciles. Nous verrons dans le chapitre 17 une technique qui permet d'atteindre des performances plus élevées.

8.5.2.2 *** Retour arrière dans le filtrage

Voici un exemple simple qui effectue un filtrage trivial sur les hypothèses (c'est-à-dire que toutes les hypothèses sont acceptées) et qui utilise également le fait que l'échec de la partie droite pour une hypothèse donnée provoque un retour arrière et un nouvel essai de la *même clause de filtrage* pour une autre hypothèse. Cette tactique permet de détruire toutes les hypothèses inutiles pour le typage de la conclusion d'un but [1].

```
Ltac clear_all :=
  match goal with
  | [id:_ |- _ ] ⇒ clear id; clear_all
  | [ |- _ ] ⇒ idtac
  end.
```

Cette tactique peut être utilisée pour réduire le nombre d'hypothèses du but en préparation pour des tactiques dont le temps d'exécution dépend de la taille du contexte. Par exemple, si l'on sait que la tactique `omega` pourra résoudre un but en utilisant seulement les hypothèses `H1`, `H2` et `H3`, on pourra envoyer la tactique suivante :

```
 generalize H1 H2 H3; clear_all; intros; omega.
```

Dans une expression de filtrage, on passe d'une clause à une autre lorsqu'on a essayé tous les cas décrits par le motif de cette clause. Ce comportement est différent du comportement couramment rencontré dans les langages de programmation fonctionnels.

Filtrage en profondeur et expressions conditionnelles

Dans les motifs de filtrage, le langage $\mathcal{L}$tac permet également d'inclure des expressions pour rechercher la présence d'un schéma dans l'expression filtrée, sans préciser la position réelle de cette expression. Ceci peut être utilisé pour contrôler l'utilisation de tactiques qui transforment le but, comme les tactiques de réécriture.

Un exemple frappant est fourni dans les bibliothèques de *Coq* dans la tactique `ring_nat` qui simplifie les expressions arithmétiques de type `nat` en utilisant les propriétés de semi-anneau de ce type muni de l'addition et de la multiplication. Cette tactique repose sur la tactique `ring` décrite en section 8.3.1. Il est possible d'utiliser cette tactique pour raisonner sur des égalités polynômiales de nombres naturels :

```
Theorem ring_example5 :
 ∀n m:nat, n*0 + (n+1)*m = n*n*0 + m*n + m.
Proof.
 intros; ring.
Qed.
```

1. Merci à Nicolas Magaud pour cette suggestion.

En revanche, le constructeur `S`, souvent utilisé à la place de la fonction qui additionne une unité à un nombre, n'est pas bien reconnu par cette tactique :

```
Theorem ring_example6 :
 ∀n m:nat, n*0 + (S n)*m = n*n*0 + m*n + m.
Proof.
 intros; ring.
  ...
  n : nat
  m : nat
  ============================
   m * S n = m + m*n
```

Pourtant les énoncés des théorèmes `Ring_example5` et `Ring_example6` sont équivalents si l'on reconnait que " `S n` " et " `n+1` " sont égaux. Ceci indique comment l'on pourrait rendre la tactique `ring` plus efficace : il suffit de commencer par remplacer toute instance de " `S n` " par " `1 + n` ". On peut le faire avec le théorème suivant :

```
Theorem S_to_plus_one : ∀n:nat, S n = n+1.
Proof.
  intros; rewrite plus_comm; reflexivity.
Qed.
```

On pourrait croire qu'il suffit maintenant d'utiliser ce théorème pour effectuer répétitivement la réécriture de toute instance de `S` en l'application correspondante de " `plus 1` ", par exemple avec la tactique suivante :

```
 repeat rewrite S_to_plus_one.
```

Cette solution est trop naïve : la tactique boucle, et il est facile de comprendre pourquoi lorsque l'on se rappelle que " `1` " est une notation pour l'expression " `S O` ". En effectuant la réécriture avec `S_to_plus_one`, on a fait réapparaître une nouvelle instance de `S`, et celle-ci est le candidat pour une nouvelle réécriture ...

Pour éviter ce comportement, il est nécessaire d'indiquer que l'on ne veut effectuer la réécriture que si l'argument de `S` n'est pas déjà `O`. On peut alors permettre l'appel récursif de la tactique. Voici une tactique qui implémente ce comportement :

```
Ltac S_to_plus_simpl :=
  match goal with
  | [ |- context [(S ?X1)] ] ⇒
      match X1 with
      | 0%nat ⇒ fail 1
      | ?X2 ⇒ rewrite (S_to_plus_one X2); S_to_plus_simpl
      end
  | [ |- _ ] ⇒ idtac
  end.
```

Cette tactique contient plusieurs caractéristiques qui méritent que l'on s'y attarde. En premier lieu, le schéma de filtrage qui apparaît sur la troisième ligne indique la syntaxe qu'il faut utiliser pour décrire la recherche d'une certaine expression à une profondeur quelconque dans le but.

En deuxième lieu, cette tactique donne également un exemple de filtrage sur une expression arbitraire, et non sur le but. Ici on effectue un filtrage sur l'expression qui apparaissait dans le but comme argument de la fonction `S` à l'aide de la construction de filtrage suivante :

```
  match X1 with
| 0%nat ⇒ fail 1
| ?X2 ⇒ rewrite (S_to_plus_one X2); S_to_plus_simpl
end
```

En troisième lieu, observons que la tactique `fail` reçoit un argument numérique. Cet argument numérique est très important, car il permet de contrôler les appels récursifs de la tactique. Nous étudions dans la section suivante.

Arguments numériques de la tactique `fail`

Les constructions `match` et "`match context with`" introduisent des *points de choix* dans le comportement des tactiques. En fait, deux niveaux de points de choix sont fournis. Le premier niveau correspond à la construction de filtrage complète, car il est possible de choisir parmi plusieurs règles de filtrage. Le second niveau correspond à la règle de filtrage, car il est possible de choisir parmi plusieurs instanciations des variables.

Chaque tactique est ainsi incluse à l'intérieur d'un certain nombre de de points de choix. Dans l'exemple de la tactique `S_to_plus_simpl`, la tactique " `fail 1` " est incluse à l'intérieur de quatre points de choix : la clause commençant par " `[0%N] ⇒ fail 1` ", la construction de filtrage commençant par " `Match X1` ", la clause de filtrage sur le contexte commençant par " `[ |- [(S ?X1)] ] ⇒` " et la construction de filtrage commençant par " `match context` ".

Lorsqu'une tactique échoue, le point de choix englobant se charge de trouver une autre instanciation des paramètres et de tenter une nouvelle exécution. C'est seulement si aucune autre instanciation ne permet un comportement sans échec que le contrôle est transféré au point de choix suivant.

L'argument numérique de la tactique `fail` permet de transférer le contrôle à un autre point de choix que le point de choix immédiatement englobant, indiquant quel point de choix englobant échoue. Si l'argument numérique est `0` c'est le point de choix immédiatement englobant, si l'argument numérique est `1`, c'est le suivant, et ainsi de suite.

Pour la tactique `S_to_plus_simpl`, la tactique "`fail 1`" permet d'indiquer que ce n'est seulement la règle commençant par " `[ 0%N ] ⇒` " qui échoue, mais également la construction " `match X1` " qui doit échouer. Ceci permet d'éviter que la deuxième règle de cette construction de filtrage soit appliquée. Le choix revient alors à la règle commençant par " `[ |- [(S ?X1)] ] =>` "qui trouve une autre instance.

Pour conclure, la construction de filtrage peut sembler familière au programmeur *ML*, mais c'est une illusion. Son pouvoir expressif est très différent car c'est un filtrage non linéaire et ambigu qui est effectué. De plus, la gestion des échecs de tactiques y est particulière, puisque la construction de filtrage tient en même temps le rôle de récupération d'exceptions. Dans le filtrage de *ML* on sait que si le filtrage a réussi pour entrer dans l'une des règles de filtrage, l'autre règle ne sera pas utilisée si la première règle échoue. Les arguments numériques de la tactique `fail` permettent de rétablir un comportement proche du filtrage du langage *ML*. Ainsi on pourra être amené à construire des tactiques composées de la forme " *tac* `|| fail 1` " si l'on veut que l'échec de la tactique *tac* ne provoque pas l'exécution d'une règle sœur de la même construction de filtrage.

Exercice 8.2 * Les bibliothèques de *Coq* fournissent deux théorèmes dont les énoncés

```
Check Zpos_xI.
```
*Zpos_xI : ∀ p:positive, Zpos (xI p) = (2 * Zpos p + 1)%Z*

```
Check Zpos_xO.
```
*Zpos_xO : ∀ p:positive, Zpos (xO p) = (2 * Zpos p)%Z*

Sachant que le nombre `2%Z` correspond en fait au terme " `POS (xO xH)` ", construire la tactique qui réécrit avec ces deux théorèmes autant que possible sans entrer dans une boucle.

8.5.3 Interactions avec la réduction

Les tactiques définies avec le langage $\mathcal{L}$tac peuvent également faire appel aux mécanismes de $\beta\delta\iota$-réduction pour effectuer certains calculs, de façon à calculer la forme réduite de certaines sous-expressions. Un exemple d'utilisation est celui qui permet d'implémenter la variante à un argument de la tactique `simpl` en n'utilisant que la tactique `simpl` sans argument. La tactique suivante simplifie une expression particulière `e` et remplace toutes les instances de cette expression par la valeur simplifiée.

```
Ltac simpl_on e :=
  let v := eval simpl in e in
  match goal with
  | [ |- context [e] ] => replace e with v; [idtac | auto]
  end.
```

Nous pouvons tester cette tactique sur un exemple artificiel :

```
Theorem simpl_on_example :
```

```
 ∀n:nat, ∃m : nat | (1+n) + 4*(1+n) = 5*(S m).
Proof.
 intros n; simpl_on (1+n).
...
============================
  ∃ m:nat | S n + 4 * S n = 5 * S m
```

La tactique `simpl_on` est très pratique si l'on veut exécuter *pas-à-pas* une fonction pour la mettre au point, mais bien sûr la variante de la tactique `simpl` avec des arguments permet déjà ce comportement (voir page 219).

Chapitre 9

Prédicats inductifs

La richesse des types inductifs dans le Calcul des Constructions inductives provient surtout de leur interaction avec le produit dépendant. Le pouvoir expressif atteint permet de formuler de nombreuses propriétés des données et des programmes simplement par typage. Les types inductifs dépendants permettent de couvrir aisément l'ensemble de la logique usuelle : les connecteurs logiques, la quantification existentielle et l'égalité peuvent être définis à l'aide de types inductifs et on retrouve uniformément les mêmes outils pour raisonner sur les connecteurs logiques et par récurrence sur les entiers naturels. Enfin, le pouvoir expressif des types inductifs permet également de décrire la programmation logique, c'est à dire les langages de la famille de Prolog.

Pour la description de programmes, les propriétés inductives fournissent des moyens de documentation et de vérification de cohérence qui représentent un saut qualitatif par rapport aux types fournis dans les langages de programmation usuels. Nous disposons maintenant de suffisamment d'outils pour construire des programmes certifiés, c'est-à-dire des programmes dont le type spécifie exactement le comportement.

Dans la section 7.6.2, nous avons vu un type dépendant d'un argument qui pouvait être vide ou non suivant la valeur de l'argument. Ici l'isomorphisme de Curry-Howard va permettre d'exploiter ce genre d'information et nous allons systématiquement construire des types inductifs dépendants pour décrire des prédicats. Un point important est que ces types inductifs n'auront aucun intérêt comme type de données, et pour cette raison il seront définis dans la sorte `Prop` plutôt que dans la sorte `Set` que nous avons utilisée dans le chapitre précédent. La non-pertinence des preuves se retrouvera dans le fait que la forme exacte d'un élément de ces nouvelles *propriétés* inductives n'aura pas d'importance. Le choix entre `Set` et `Prop` interfère avec les outils d'extraction de programmes exécutables à partir de développements *Coq*.

9.1 Quelques propriétés inductives

9.1.1 Quelques exemples

NEW: Merci Gérard : dire en quoi c'est différent d'une définition sans Inductive ; Voir phrase ajoutée le 20/10 en V.A.

L'exemple suivant utilise le type `plane` défini dans la section 7.1.5. Le prédicat “ `south_west` a b ” signifie « le point a est au sud-ouest de b ».

```
Inductive south_west : plane→plane→Prop :=
  south_west_def :
  ∀a1 a2 b1 b2:Z, (a1 ≤ b1)%Z → (a2 ≤ b2)%Z →
        south_west (point a1 a2)(point b1 b2).
```

On remarquera que, comme dans le chapitre 7, les types inductifs peuvent ne pas présenter de caractère récursif.

On peut également définir le prédicat `even` sur les nombres naturels qui est satisfait si et seulement si l'argument est pair :

```
Inductive even : nat→Prop :=
  | O_even : even 0
  | plus_2_even : ∀n:nat, even n → even (S (S n)).
```

Cette définition inductive est pour ainsi dire la jumelle de la définition de `even_line` vue en section 7.6.2. La seule distinction est la sorte utilisée pour le type de `even`. Cette sorte souligne l'intention logique dans l'utilisation de cette définition inductive. L'outil *Coq* réagit différemment en réponse à cette indication, principalement dans la forme donnée au principe de récurrence associé.

Un autre exemple issu de la certification d'algorithmes est celui de la définition suivante du prédicat « être une liste ordonnée ». Il s'agit ici d'une définition inductive paramétrée par le type des éléments de la liste et la relation d'ordre considérée.

```
Inductive sorted (A:Set)(R:A→A→Prop) : list A → Prop :=
  | sorted0 : sorted A R nil
  | sorted1 : ∀x:A, sorted A R (cons x nil)
  | sorted2 :
      ∀(x y:A)(l:list A),
        R x y →
        sorted A R (cons y l)→ sorted A R (cons x (cons y l)).

Implicit Arguments sorted [A].
```

Le prédicat `le` que nous avons rencontré en section 5.2.1.1 page 110 est aussi un prédicat défini par induction. Il s'agit en fait d'une définition paramétrée, qui repose sur la description d'une famille de prédicats unaires « être supérieur ou égal à n », donc paramétrée par n.

```
Inductive le (n:nat) : nat→Prop :=
  | le_n : le n n
  | le_S : ∀m:nat, le n m → le n (S m).
```

Nous pouvons également considérer la clôture transitive d'une relation binaire R définie sur un type A. On remarquera dans les définitions suivantes — extraites de la bibliothèque `Relations` de *Coq* `clos_trans` est paramétrée par A et R.

```
Definition relation (A:Type) := A → A → Prop.

Inductive clos_trans (A:Type)(R:relation A) : A→A→Prop :=
  | t_step : ∀x y:A, R x y → clos_trans A R x y
  | t_trans :
    ∀x y z:A, clos_trans A R x y → clos_trans A R y z →
        clos_trans A R x z.
```

Exercice 9.1 * Définir la propriété inductive " `last A a l` " vraie si et seulement si `l` est une liste de valeurs de type `A` et `a` est le dernier élément de `l`. Définir également une fonction `(last_fun : list A → option A)` qui retourne ce dernier élément s'il existe. Exprimer et démontrer la cohérence entre les deux définitions. Comparer la difficulté de programmation entre les deux définitions.

Exercice 9.2 * Définir de façon inductive la propriété « être un palindrome », c'est à dire une liste se lisant de la même façon dans les deux sens. On pourra définir un prédicat auxiliaire (également inductif) généralisant `last`, en ce sens qu'un de ses arguments précise ce qu'il reste de la liste `l` après avoir ôté son dernier élément.

Exercice 9.3 Définir de façon inductive la clôture transitive et réflexive d'une relation binaire quelconque.

On remarquera que le module `Rstar` de la bibliothèque standard donne une définition imprédicative de cette clôture (constante `Rstar`); vous devrez alors montrer l'équivalence entre votre définition et celle de la bibliothèque de Coq.

9.1.2 Propriétés inductives et programmation logique

La description d'une propriété inductive suit souvent la même structure qu'un programme de programmation logique, comme ceux que l'on écrit dans le langage *Prolog* (sans utiliser le coupe-choix, aussi appelé *Cut* en anglais). Bien sûr, le langage *Prolog* n'est pas typé et on trouve donc plus d'information dans une définition de propriété inductive que dans un programme *Prolog*, mais on peut quand même établir une correspondance entre les constructeurs d'une propriété inductive et les clauses qui définissent un prédicat *Prolog*.

Par exemple, on peut représenter les nombres naturels en *Prolog* en utilisant un atome `o` et un symbole de fonction [1] `s`. Le prédicat `even` qui indique si un nombre naturel est pair peut être défini par les clauses suivantes :

1. Nous écrivons ces atomes en minuscules parce que les conventions syntaxiques de *Prolog* préconisent que les identificateurs commençant par une majuscule représentent des variables.

```
even(o).

even(s(s(N))) :- even(N).
```

Nous pouvons également définir en *Prolog* le prédicat `le` :

```
le(N,N).

le(N, s(M)) :- le(N,M).
```

Enfin, le prédicat `sorted`, lorsqu'il est instancié pour une relation d'ordre particulière, correspond également à un jeu de clauses *Prolog* :

```
sorted([]).

sorted([X]).

sorted([N1;N2|L]) :- le(N1,N2), sorted([N2|L]).
```

Chacune des clauses *Prolog* pour les prédicats `le` et `sorted` correspondent à l'un des constructeurs du type inductif de même nom.

En revanche, il n'est pratiquement pas possible de définir un prédicat général pour `sorted` et `clos_trans` sans les instancier pour une relation particulière. En général, *Prolog* ne permet pas de représenter des définitions inductives où certains constructeurs prennent des fonctions en argument car *Prolog* travaille au premier ordre alors que les définitions inductives peuvent travailler à l'ordre supérieur.

Bien que les constructions inductives de *Coq* aient un pouvoir expressif supérieur à *Prolog*, on peut mettre à profit la similitude entre ces deux formalismes et utiliser *Coq* pour raisonner sur des programmes donnés en programmation logique. Par exemple, on sait utiliser la programmation logique pour décrire les langages de programmation et *Coq* peut ensuite être utilisé pour vérifier des propriétés des langages étudiés [12].

Du point de l'évaluation, un interprète *Prolog* est un outil automatique de démonstration sur des propriétés inductives. Ce comportement de recherche automatique est fourni en *Coq* par la tactique "`eauto`" (voir section 8.2.2), mais de façon beaucoup plus lente.

9.1.3 Conseils pour les définitions inductives

Voici quelques principes simples qui permettront souvent d'éviter des erreurs dans la définition de nouvelles propriétés inductives.

- Les constructeurs sont des axiomes ; à ce titre il devraient être intuitivement vrais.
- Il est préférable que les constructeurs définissent des cas mutuellement exclusifs sur les données que l'on sera amené à considérer comme les entrées. Dans le cas contraire, les démonstrations par récurrence sur ce prédicat contiendraient des duplications.

- Lorsque des arguments dépendants apparaissent toujours avec la même valeur, il est préférable de donner à ces arguments le statut de paramètre. Le principe de récurrence engendré est alors plus simple.
- Il est utile de vérifier la propriété définie sur des exemples positifs (montrer que l'on sait construire une preuve qu'un certain objet vérifie la propriété) et négatifs (montrer que l'on sait construire une preuve qu'un objet qui ne devrait pas vérifier la propriété ne la vérifie effectivement pas).

Il peut également être utile de donner plusieurs définitions pour le même concept et de montrer l'équivalence entre ces différentes définitions (voir par exemple l'exercice 9.14, page 261.)

Chaque définition d'un même concept donne un point de vue différent sur le concept considéré. En particulier, le principe de récurrence associé à la définition inductive d'un prédicat permet de structurer les démonstrations concernant ce prédicat. Disposer de plusieurs définitions inductives pour le même concept permet donc de multiplier les moyens de structurer les démonstrations. Par exemple, la série d'exercices sur les expressions bien parenthésées qui démarre avec l'exercice 9.5 montre l'intérêt de disposer de plusieurs définitions inductives pour la notion d'expression bien parenthésée : la première est une définition naturelle, facilement compréhensible pour un être humain (les autres définitions, proposées dans les exercices 9.19 et 9.20 page 262 donnent des définitions alternatives, mais mieux adaptées pour vérifier la correction d'un analyseur syntaxique).

9.1.4 L'exemple des listes triées

Nous illustrons les considérations précédentes en commentant la définition du prédicat `sorted` donnée page 242. Les trois constructeurs `sorted0`, `sorted1` et `sorted2` sont mutuellement exclusifs, car s'appliquant respectivement à des listes de 0, 1 ou au moins 2 éléments. Seul le troisième constructeur est récursif ; on remarque d'autre part les paramètres `A` (type des éléments de la liste) et `R` (relation prise en compte pour les comparaisons) de la définition de `sorted`, exprimant que ces paramètres sont constants dans toute cette définition. En revanche, à l'extérieur de cette définition, les constantes voient leur type affecté suivant le mécanisme décrit en section 7.4.

sorted
  : ∀ A:Set, (A → A → Prop) → list A → Prop

sorted0
  : ∀ (A:Set) (R:A → A → Prop), sorted R nil

sorted1
  : ∀ (A:Set) (R:A → A → Prop) (x:A), sorted R (cons x nil)

sorted2
  : ∀ (A:Set) (R:A → A → Prop) (x y:A) (l:list A),

$$R\ x\ y \to sorted\ R\ (cons\ y\ l) \to sorted\ R\ (cons\ x\ (cons\ y\ l))$$

Une preuve construite par applications successives des constructeurs `sorted0`, `sorted1` et `sorted2` permet de prouver que telle liste est ordonnée. Par exemple, la preuve suivante montre que la liste `[1;2;3]` est ordonnée. Cette démonstration se fait automatiquement à l'aide de la tactique `auto`, une fois entrés les constructeurs de la définition inductive dans la base de donnée de théorèmes.

NEW: Merci Gérard : Créer une base pour ça (sinon le “auto with arith” fait bizarre, car on croit que seule l'arithmétique intervient) ; corrigé dans la version anglaise le 20/10/03

```
Hint Resolve sorted0 sorted1 sorted2 : sorted_base.
Theorem sorted_nat_123 : sorted le (1::2::3::nil).
Proof.
 auto with sorted_base arith.
Qed.
```

Le lecteur pourra constater, en imprimant le terme de preuve de ce dernier théorème, qu'il est constitué uniquement d'applications des constructeurs des types `nat`, `list`, `le` et `sorted`.

De même, nous pouvons prouver automatiquement que si $x \leq y$, alors la liste `[`x`;`y`]` est ordonnée.

```
Theorem xy_ord :
 ∀x y:nat, le x y → sorted  le (x::y::nil).
Proof.
 auto with sorted_base.
Qed.
```

En revanche, il paraît bien plus difficile de prouver des résultats « négatifs », tels que, par exemple « la liste `[1;2;3]` n'est pas ordonnée ». Il faut en effet montrer que supposer que cette liste est ordonnée conduit à une contradiction. De façon plus générale, nous verrons des techniques permettant — étant donné un prédicat P correctement choisi — de montrer que toute liste ordonnée satisfait P. Voici par exemple une preuve que si l est une liste ordonnée d'entiers naturels, alors la liste “ `cons 0` l ” est aussi ordonnée. Dans ce cas précis, le prédicat P est la fonction “ `fun l:list nat ⇒ sorted nat (cons 0 l)` ” ; de fait ce prédicat est construit automatiquement par *Coq*, ce qui le rend absent du script ci-dessous (nous décrirons plus précisément ce type de preuve en section 9.3) :

```
Theorem zero_cons_ord :
 ∀l:list nat, sorted le l → sorted le (cons 0 l).
 Proof.
 induction 1; auto with sorted_base arith.
Qed.
```

Nous verrons en section 9.5.2 des techniques nous permettant de prouver facilement des « réciproques » des constructeurs, d'où leur nom de *tactiques d'inversion*. Voici deux lemmes, dont l'application permet de déduire facilement que la liste `[1;3;2]` n'est pas ordonnée (voir l'exercice 9.28, page 280) :

```
Theorem sorted1_inv :
 ∀(A:Set)(le:A→A→Prop)(x:A)(l:list A),
   sorted  le (cons x l)→ sorted  le l.
Proof.
 inversion 1; auto with sorted_base.
Qed.

Theorem sorted2_inv :
 ∀(A:Set)(le:A→A→Prop)(x y:A)(l:list A),
   sorted  le (cons x (cons y l))→ le x y.
Proof.
 inversion 1; auto with sorted_base.
Qed.
```

Exercice 9.4 * Définir les relations suivantes sur “ `list` A ” :

- La liste l' s'obtient à partir de l en transposant deux éléments consécutifs,
- La liste l' s'obtient à partir de l en appliquant un nombre fini de fois l'opération ci-dessus. On dit alors que l' est une *permutation* de l.

Montrer que la seconde de ces relations est une relation d'équivalence.

Exercice 9.5 Le but de cet exercice est de considérer les expressions bien parenthésées. Il démarre une série d'exercices qui culminera par la construction d'un analyseur syntaxique pour les expressions bien parenthésées, c'est à dire un programme qui reconstruit la structure d'une expression (voir pages 262, 269). On considère le type de caractères suivant :

```
Inductive par : Set := open | close.
```

Dans ce type les constructeurs `open` et `close` représentent les caractères « ouvrez la parenthèse » et « fermez la parenthèse ». Nous représentons les chaînes de caractères par des objets de type “ `list par` ” où `list` est le type des listes polymorphes décrit en section 7.4.1 page 204.

Une expression bien parenthésée est :

1. soit l'expression vide,
2. soit une expression bien parenthésée entre parenthèses,
3. soit la concaténation de deux expressions bien parenthésées.

Définir la propriété inductive `(wp:(list par)→Prop)` correspondant à cette définition informelle. On pourra utiliser la fonction `app` fournie dans le module `List` pour concaténer deux listes. Démontrer les propriétés suivantes :

```
wp_oc : wp (cons open (cons close nil))

wp_o_head_c :
     ∀l1 l2:list par,
       wp l1 → wp l2 → wp (cons open (app l1 (cons close l2)))
```

```
wp_o_tail_c :
     ∀l1 l2:list par, wp l1 → wp l2 →
       wp (app l1 (cons open (app l2 (cons close nil)))).
```

Exercice 9.6 Cet exercice fait suite à l'exercice 9.5. On considère un type d'arbres binaires sans étiquettes et une fonction associant à chaque arbre binaire une liste de caractères, donnés par les définitions suivantes. Montrer que cette fonction retourne toujours une liste bien parenthésée :

```
Inductive bin : Set := L : bin | N : bin→bin→bin.

Fixpoint bin_to_string (t:bin) : list par :=
  match t with
  | L ⇒ nil
  | N u v ⇒
    cons open
      (app (bin_to_string u)(cons close (bin_to_string v)))
  end.
```

Exercice 9.7 Cet exercice fait suite à l'exercice 9.6. Montrer que la fonction suivante retourne aussi toujours une expression bien parenthésée :

```
Fixpoint bin_to_string' (t:bin) : list par :=
  match t with
  | L ⇒ nil
  | N u v ⇒
    app (bin_to_string' u)
        (cons open (app (bin_to_string' v)(cons close nil)))
  end.
```

9.2 Propriétés inductives et connecteurs logiques

Dans le système *Coq*, les connecteurs logiques usuels, à l'exception de l'implication et de la quantification universelle, déjà fournis par les types de fonctions simples ou dépendantes, sont tous donnés par des définitions inductives. Nous avons déjà rencontré et utilisé certaines constantes et connecteurs logiques : `False`, `True`, `not`, `and` et `or`, sans donner leur définition ; c'est à cette définition — inductive —, que nous nous intéressons ici.

De manière générale, les constructeurs composant la définition inductive des connecteurs logiques correspondent aux règles d'introduction de ces connecteurs en déduction naturelle [79], tandis que les principes de récurrence correspondent aux règles d'élimination. C'est l'origine du nom de la tactique `elim`. Cette tactique s'utilise pour les connecteurs logiques comme pour les autres types inductifs. De manière naturelle, on l'utilisera à chaque fois que l'on veut extraire l'information d'une hypothèse formée à l'aide d'un connecteur logique.

9.2.1 Représentation de la vérité

La proposition toujours vraie, c'est-à-dire une propriété prouvable dans n'importe quel contexte, fait l'objet d'une définition inductive `True` à un seul constructeur `I` sans argument. On dispose alors d'une preuve sans condition pré-requise. La définition est donnée dans les bibliothèques de *Coq* de la façon suivante :

```
Inductive True : Prop := I : True.
```

Le principe de récurrence associé à cette définition inductive a l'énoncé suivant :

True_ind : ∀ *P:Prop, P* → *True* → *P*

Ce principe de récurrence — engendré automatiquement — est inutile, puisqu'il ne permet de prouver `P` qu'à la condition que l'on ait déjà une preuve de `P`.

9.2.2 Représentation de la contradiction

La proposition contradictoire doit être une proposition pour laquelle il n'existe pas de preuve. Si nous voulons représenter cette proposition par un type inductif, ceci peut s'exprimer par le fait qu'il n'existe simplement pas de constructeur pour ce type [2]. C'est le choix effectué dans les bibliothèques de *Coq* :

```
Inductive False : Prop := .
```

Le principe de récurrence associé à cette définition inductive a l'énoncé suivant :

False_ind : ∀ *P:Prop, False* → *P*

Nous avons déjà observé l'importance de ce principe de récurrence en section 5.3.5 page 128. C'est le principe de raisonnement résumé par la formule latine *ex falso quodlibet* : du faux on peut déduire ce qu'on veut.

En pratique, ceci indique que tout but contenant une hypothèse `H` dont le type est `False` peut être résolu simplement par “ `elim H` ”. En effet, l'énoncé de `False_ind` montre bien qu'aucun sous-but n'est engendré par cette élimination.

La négation n'est pas fournie par une définition inductive propre : c'est une fonction unaire définie à partir de la constante `False`. Les aspects pratiques de la démonstration de formules contenant des négations ont déjà été traités dans la section 5.3.5.

9.2.3 Représentation de la conjonction

Observons maintenant les connecteurs logiques principaux. Le connecteur « et » est ainsi défini :

```
(* This fragment is not in the book *)
Module redefine_and.
(* end of fragment *)
```

2. il y a d'autres solutions, comme le suggère le type `strange` introduit en section 7.6.1.

```
Inductive and (A B:Prop) : Prop := conj : A → B → and A B.
```

Outre cette définition inductive, le système fournit une convention syntaxique pour ce connecteur logique, de sorte que " `and A B` " s'écrit en fait " `A/\B` ", que nous noterons " `A∧B` " dans cet ouvrage. De même que dans le langage mathématique usuel, le connecteur $\wedge$ a une priorité plus forte que $\rightarrow$ et plus faible que $\sim$; ainsi la formule " `~ A ∧ A → B` " se lit-elle comme " `(and (not A) A) → B` ".

Le principe de récurrence associé est le suivant :

and_ind : ∀ A B P:Prop, (A→B→P)→A∧B→P

Le constructeur `conj` sert de règle d'introduction pour la conjonction. Nous l'avons déjà vu en section 5.3.5 page 129 et il est également utilisé dans la tactique `split`, vue en section 6.2.4, page 150.

Le principe de récurrence peut quant à lui être utilisé comme règle d'élimination, c'est-à-dire qu'il permet de déduire des informations d'une conjonction.

9.2.4 Représentation de la disjonction

Le connecteur logique « ou » est obtenu avec la définition inductive suivante :

```
Inductive or (A B:Prop) : Prop :=
  | or_introl : A → or A B | or_intror : B → or A B.
```

Le premier constructeur indique que pour prouver une disjonction, on peut se contenter de prouver son membre gauche, tandis que le second indique que l'on peut se contenter de prouver le membre droit. Nous avons déjà rencontré ces constructeurs en section 5.3.5 page 129.

Le système *Coq* fournit une notation syntaxique pour ce connecteur logique : " `or A B` " sera habituellement écrit " `A\/B` " (noté " `A ∨ B` " dans cet ouvrage). La priorité de la disjonction se trouve entre celle de la conjonction et celle de l'implication ; ainsi la proposition " `A ∨ B ∧ A → A` " se lit-elle " `(or A (and B A))→A` ".

Le principe de récurrence associé à cette définition est le suivant :

or_ind : ∀ A B P:Prop, (A→P)→(B→P)→A∨B→P

Coq fournit deux tactiques `left` et `right` pour remplacer les commandes " `apply or_introl` " et " `apply or_intror` " ; le principe de récurrence `or_ind` est quant à lui appliqué lors des étapes d'élimination de la conjonction (revoir l'exemple page 150 et lire le terme de preuve de `or_commutes`.)

9.2.5 Représentation de la quantification existentielle

Le quantificateur logique « il existe » est décrit par la définition inductive suivante :

```
Inductive ex (A:Type)(P:A→Prop) : Prop :=
    ex_intro : ∀x:A, P x → ex A P.
```

Le principe de récurrence associé est le suivant :

ex_ind
: ∀ (A:Type)(P:A→Prop)(P0:Prop), (∀ x:A, P x → P0)→ ex A P → P0

Le système *Coq* fournit également des abréviations pour ce connecteur logique, décrites en section 5.3.5 page 129.

Le constructeur `ex_intro` est particulier, car il comporte une quantification universelle sur une variable `x` qui n'apparaît pas dans sa conclusion : “ `ex A P` ”. Pour cette raison, ce constructeur ne peut pas être utilisé avec la tactique `apply` sans directive `with` (voir les sections 6.1.3 et 6.1.3.) En d'autres termes, pour prouver une quantification existentielle, l'utilisateur doit fournir un *témoin*. La tactique “ `exists` *e* ” a le même effet que “ `apply ex_intro with` *e* ’.

9.2.6 Représentation inductive de l'égalité

L'égalité entre deux termes est représentée comme un type inductif paramétré, donné par la définition suivante :

```
Inductive eq (A:Type)(x:A) : A→Prop := refl_equal : eq A x x.
```

Le système *Coq* fournit une notation syntaxique pour l'égalité et la formule “ `eq A x y` ” sera écrite “ `x=y` ”.

Le principe de récurrence associé à cette définition est le suivant :

eq_ind : ∀ (A:Type)(x:A)(P:A→Prop), P x →∀ y:A, x=y → P y

La tactique “ `rewrite <-` *e* ” est en fait équivalente à la tactique “ `elim` *e* ”.

La réécriture que nous avons déjà étudiée dans la section 6.3 repose donc sur ce principe de récurrence.

9.2.7 *** Égalité dépendante

L'égalité `eq` impose que l'on ne puisse décrire que l'égalité de deux termes qui sont de même type. Selon une proposition de C.T. McBride [65], il est possible de considérer un prédicat d'égalité qui permet de parler de l'égalité d'expressions de types différents, même si cette égalité ne sera prouvable que pour des termes de même type. Le module `JMeq`[3] de la bibliothèque de *Coq* fournit la définition suivante :

```
Inductive JMeq (A:Set)(x:A) : ∀B:Set, B→Prop :=
    JMeq_refl : JMeq x x.
```

3. Les initiales « J.M. » font référence à un homme politique britannique, et sont le fruit d'une plaisanterie. Les types représentent des classes et l'introduction de cette égalité semble représenter un progrès social, puisque l'on peut énoncer la possibilité que deux individus de classe différente aspirent à être égaux, même si au fond, seuls des individus de même classe pourront effectivement être égaux.

Cette déclaration étant effectuée sous le mode d'arguments implicites, bien que la relation `JMeq` prenne en réalité quatre arguments, nous n'aurons pas besoin de donner les arguments de type correspondant à `A` et `B`. Ce prédicat d'égalité s'utilise comme le prédicat d'égalité `eq`. En particulier, une élimination sur ce prédicat permet encore de faire une réécriture de la droite vers la gauche avec une égalité. Cette égalité est intéressante par exemple pour décrire l'injectivité de constructeurs ayant un type dépendant.

Le principe de récurrence `JMeq_ind` utilisé pour cette notion d'égalité n'est pas le principe de récurrence engendré de façon systématique pour la définition inductive. C'est un autre autre principe, lui même prouvé à l'aide de l'axiome `JMeq_eq` qui affirme que l'égalité au sens `JMeq` implique l'égalité au sens `eq`, c'est à dire l'égalité usuelle.

```
Require Import JMeq.

Check JMeq_eq.
JMeq_eq : ∀ (A:Set)(x y:A), JMeq x y → x = y

Check JMeq_ind.
JMeq_ind : ∀ (A:Set)(x y:A)(P:A→Prop), P x → JMeq x y → P y
```

Par exemple, nous pouvons définir un type regroupant tous les arbres de hauteur fixée, en reprenant le type `htree` que nous avons défini en section 7.5.2. Nous considérons qu'un arbre de haiteur fixée est le couple formé d'un nombre h et d'un arbre de hauteur h :

```
Inductive ahtree : Set :=
  any_height : ∀n:nat, htree nat n → ahtree.
```

On peut exprimer que `any_height` est injectif sur sa deuxième coordonnée avec le théorème suivant :

```
Theorem any_height_inj2 :
 ∀(n1 n2:nat)(t1:htree nat n1)(t2:htree nat n2),
   any_height n1 t1 = any_height n2 t2 → JMeq t1 t2.
Proof.
 intros n1 n2 t1 t2 H.
 ...
 H : any_height n1 t1 = any_height n2 t2
============================
 JMeq t1 t2

 injection H.
 ...
 ============================
  existS (fun n:nat => htree nat n) n1 t1 =
  existS (fun n:nat => htree nat n) n2 t2 → n1 = n2 → JMeq t1 t2
```

Bien que cette démonstration soit très courte, elle comporte une difficulté particulière. La tactique “ `change ...` ” effectue une réécriture simultanée sur deux expressions différentes : `n2` (qui apparaît dans le type caché de `t2`) et `t2`. Ce comportement n'est fourni par aucune autre tactique de *Coq*.

Pour donner un exemple significatif de l'utilisation de cette nouvelle égalité, nous allons considérer la correspondance entre un type dépendant, celui des vecteurs d'une longueur fixée, et un type non dépendant, celui des listes de longueur arbitraire. Le type des vecteurs est fourni dans un module de *Coq* nommé `Bvector`.

Nous définissons une bijection entre la famille de type `vector` et le type `list`, à l'aide de deux fonctions récursives dont la définition est très simple :

```
Require Import Bvector.
Require Import List.

Section vectors_and_lists.
 Variable A : Set.
 Fixpoint vector_to_list (n:nat)(v:vector A n){struct v}
  : list A :=
  match v with
  | Vnil ⇒ nil
  | Vcons a p tl ⇒ cons a (vector_to_list p tl)
  end.

 Fixpoint list_to_vector (l:list A) : vector A (length l) :=
   match l as x return vector A (length x) with
   | nil ⇒ Vnil A
   | cons a tl ⇒ Vcons A a (length tl)(list_to_vector tl)
   end.
```

Le problème que nous nous posons maintenant est de montrer que ces deux fonctions établissent bien une bijection. Il faut d'abord indiquer que le retour se fait bien dans le type de départ. En d'autres termes, il faut que la longueur du vecteur soit conservée :

```
 Theorem keep_length :
  ∀(n:nat)(v:vector A n), length (vector_to_list n v) = n.
 Proof.
   intros n v; elim v; simpl; auto.
 Qed.
```

Pour effectuer la preuve en entier, nous allons également démontrer que l'égalité `JMeq` traverse bien le constructeur `Vcons`. Ceci s'exprime par le lemme suivant (suggéré par Christine Paulin).

```
Theorem Vconseq :
 ∀(a:A)(n m:nat),
   n = m →
```

```
  ∀(v:vector A n)(w:vector A m),
    JMeq v w → JMeq (Vcons A a n v)(Vcons A a m w).
Proof.
intros a n m Heq; rewrite Heq.

intros v w HJeq.
...
Heq : n = m
v : vector A m
w : vector A m
HJeq : JMeq v w
============================
JMeq (Vcons A a m v)(Vcons A a m w)
```

Grâce à la réécriture par l'hypothèse `Heq`, l'hypothèse `HJeq` contient maintenant une égalité homogène, où les deux expressions comparées sont de même type. Le principe de récurrence spécialisé pour l'égalité `JMeq` permet maintenant une réécriture (ici nous effectuons la réécriture directement avec la commande de base `elim`, ceci souligne que c'est bien `JMeq_ind` qui est utilisé dans la démonstration, ce que le lecteur pourra vérifier à l'aide de la commande `Print`).

```
 elim HJeq; reflexivity.
Qed.
```

Grâce au lemme `Vconseq` nous pouvons maintenant aborder la démonstration du théorème principal.

```
Theorem vect_to_list_and_back :
 ∀n (v:vector A n),
   JMeq v (list_to_vector (vector_to_list n v)).
Proof.
 intros n v; elim v.
 simpl; auto.
 intros a n' v' HJeq.
 simpl.
...
 HJeq : JMeq v' (list_to_vector (vector_to_list n' v'))
============================
 JMeq (Vcons A a n' v')
   (Vcons A a (length (vector_to_list n' v'))
     (list_to_vector (vector_to_list n' v')))
```

Nous aimerions utiliser l'égalité `HJeq` pour réécrire dans ce but, mais ceci n'est pas possible parce que cette égalité n'est pas homogène. De plus nous avons besoin de faire simultanément deux réécritures avec deux égalités différentes, d'une part pour remplacer le type

```
vector A (length (vector_to_list n' v'))
```

par le type “ `vector A n'` ” grâce au lemme `keep_length`, et d'autre part pour remplacer “ `list_to_vector (vector_to_list n' v')` ” par `v'`. Le but intermédiaire obtenu après application de l'une de ces deux réécritures serait mal typé. Ce sont ces deux remplacements simultanés que permet le lemme `Vconseq`.

```
 apply Vconseq.
 symmetry; apply keep_length.
 assumption.
Qed.
```

La difficulté rencontrée dans la démonstration ci-dessus se retrouvera probablement à chaque fois que l'on voudra utiliser une égalité obtenue à l'aide de ce théorème et l'on pourra rarement effectuer des réécritures, car l'égalité sera rarement homogène. La technique passant par un lemme auxiliaire plus général comme le lemme `Vconseq` devra alors être employée.

Exercice 9.8 Démontrer l'énoncé suivant, sans utiliser l'égalité `eq` :

```
∀x y z:nat, JMeq (x+(y+z))((x+y)+z)
```

9.2.8 Pourquoi un principe de récurrence exotique ?

Le principe de récurrence naturellement associé avec la définition inductive de `JMeq` peut être retrouvé grâce à la commande `Scheme` (voir section 15.1.6).

```
Scheme JMeq_ind2 := Minimality for JMeq Sort Prop.
Check JMeq_ind2.
```

JMeq_ind2
: ∀ (A:Set)(x:A)(P:∀ B:Set, B → Prop),
P A x → ∀ (B:Set)(b:B), JMeq x b → P B b

Ce principe de récurrence contient une quantification universelle sur toutes les prédicats `P` sur tous les types de sortes `Set`. La propriété ainsi demandée est trop difficile à atteindre et il est impossible en pratique de satisfaire les prémisses de ce principe. En particulier, il est impossible de construire une propriété `P` qui soit bien typée et corresponde à la réécriture attendue.

Par exemple, observons plus en détail la démonstration de `Vconseq`. Dans cette démonstration, `JMeq` est utilisé avec un prédicat `P` sur “ `vector A m` ” dont la définition est la suivante :

```
fun (a:A)(m:nat)(v w0:vector A m) ⇒
     JMeq (Vcons A a m v)(Vcons A a m w0)
```

Ce prédicat est instancié avec `(w:vector A m)` pour obtenir le but avant réécriture, c'est-à-dire

```
JMeq (Vcons A a m v)(Vcons A a m w0)
```

et avec `(v:vector A m)` pour obtenir le but après réécriture, c'est-à-dire

```
JMeq (Vcons A a m v)(Vcons A a m v)
```

Si nous voulons utiliser le principe de récurrence `JMeq_ind2`, il nous faut trouver une propriété `P`, qui soit bien typée et qui s'instancie bien en ces deux buts, lorsqu'elle est appliquée aux valeurs “ `vector A m` ” et `w` d'une part et “ `vector A m` ” et `v` d'autre part. Ceci n'est pas possible.

Par exemple, nous ne pouvons pas prendre pour `P` la valeur suivante :

```
fun (B:Set)(b:B) ⇒ JMeq (Vcons A a m v)(Vcons A a m b)
```

En effet, cette expression est mal typée, puisque `b` a le type `B`, alors qu'il faudrait que cette expression ait le type “ `vector A m` ” pour que “ `Vcons A a m b` ” soit bien typé. Le prédicat `P` ne peut pas être décrit, alors que ses deux instances sur `w` et `v`, tous deux de type “ `vector A m` ” sont bien typées, et ce sont les seules expressions pour lesquelles on a besoin de ce prédicat. Le principe de récurrence « exotique » proposé par C. McBride affirme que cette restriction aux objets de même type est valide.

Pour mieux comprendre l'utilisation de cette égalité hétérogène, nous invitons le lecteur à se reporter aux travaux de C. Alvarado [5].

9.3 Raisonnement sur les propriétés inductives

9.3.1 Variantes structurées de `intros`

Lorsque l'on veut à la fois introduire des hypothèses et décomposer les connecteurs logiques qu'elles contiennent, on est amené à construire des tactiques composées qui font apparaître une grande quantité de tactiques `intros` et `elim` alternées. Il est en fait possible de condenser ces tactiques composées en utilisant une variante élaborée de la tactique `intros` qui autorise à introduire une hypothèse en la décomposant à la volée. On utilise pour cette variante des crochets pour indiquer la volonté de destructurer l'hypothèse.

Un premier exemple d'utilisation fait intervenir un connecteur logique à un seul constructeur comme la conjonction ou la quantification existentielle. Il suffit alors de donner à la tactique `intros` une expression bien parenthésée qui imite la structure de l'expression logique. En voici un exemple :

```
Theorem structured_intro_example1 : ∀A B C:Prop, A∧B∧C→A.
Proof.
 intros A B C [Ha [Hb Hc]].
```

...
Ha : A
Hb : B
Hc : C
============================
A

Des exemples plus élaborés font intervenir des connecteurs logiques à plusieurs constructeurs, comme la disjonction. Dans ce cas il faut utiliser une barre verticale “|” pour séparer les structures potentiellement différentes qui apparaîtront dans chacun des cas :

```
Theorem structured_intro_example2 : ∀A B:Prop, A ∨ B∧(B→A)→A.
Proof.
 intros A B [Ha | [Hb Hi]].
```

```
  ...
  Ha : A
  ============================
   A
```

Le deuxième but engendré a la forme suivante :

```
  ...
  Hb : B
  Hi : B→A
  ============================
   A
```

9.3.2 Les tactiques `constructor`

L'application des constructeurs d'une définition inductive peut être abrégée par une tactique qui recherche le premier constructeur qui s'applique, cette tactique s'appelle `constructor`. Lorsque la définition inductive ne présente qu'un seul constructeur, la tactique `split` applique ce constructeur. Il est possible de donner un ou plusieurs arguments en utilisant la variante `with` comme pour la tactique `apply`. La tactique `exists` est une autre variante de “ `split with` ”. Les tactiques `left` et `right` sont également des variantes de `constructor`qui ne s'utilisent que si le type a exactement deux constructeurs.

9.3.3 ∗ Récurrence sur les prédicats inductifs

La démonstration par récurrence sur les propriétés inductives est généralement assez efficace car le principe de récurrence associé à une propriété inductive exprime très bien les propriétés réunies pour que chaque constructeur s'applique.

Une fois défini le prédicat inductif `even` (voir section 9.1 page 242), et si `n` est un nombre naturel pair arbitraire, nous disposons de deux principes de récurrence pour effectuer des démonstrations sur `n` : le principe de récurrence des entiers naturels et le principe de récurrence des nombres pairs. Le second est souvent beaucoup plus expressif, au sens où il permet d'éviter de prendre en compte les nombres naturels impairs.

Imaginons par exemple que nous voulons démontrer que la somme de deux nombres pairs est un nombre pair. Une démonstration reposant sur le principe de récurrence usuel des nombres naturels, `nat_ind`, nous demande d'établir la propriété que $S(x) + p$ est pair lorsque $S(x)$ et p sont pairs, en utilisant une hypothèse de récurrence sur x. Mais si le successeur de x est pair, alors x ne l'est pas et nous ne pouvons pas utiliser l'hypothèse de récurrence.

Un essai manqué

Montrons comment progresse la démonstration lorsque nous utilisons le principe de récurrence usuel des nombres naturels.

```
Theorem sum_even : ∀n p:nat, even n → even p → even (n+p).
Proof.
 intros n; elim n.
 ...
 n:nat
 ============================
  ∀ p:nat, even 0 → even p → even (0+p)
```

Ce but est facile à résoudre, car les règles de conversion pour `plus` (également décrites dans la section 7.3.3) entraînent que “ `0 + p` ” est convertible en `p`.

```
 auto.
 ...
 n : nat
 ============================
  ∀ n0:nat, (∀ p:nat, even n0 → even p → even (n0+p))→
    ∀ p:nat, even(S n0)→ even p → even (S n0 + p)
```

Pour rendre le deuxième but plus lisible, nous introduisons les variables et hypothèses :

```
 intros n' Hrec p Heven_Sn' Heven_p.
 ...
 Hrec : ∀ p:nat, even n' → even p → even (n'+p)
 p : nat
 Heven_Sn' : even (S n')
 Heven_p : even p
 ============================
  even (S n' + p)
```

Nous aboutissons ici à une impasse. Nous voulons montrer que $S(n')+p$ est pair et l'hypothèse de récurrence nous permet seulement de démontrer que $n'+p$ est pair. Or si l'une de ces deux expressions représente un nombre pair, nous savons que l'autre représente un nombre impair. La même propriété se retrouve sur n' et $S(n')$. L'hypothèse `Heven_Sn'` indique que $S(n')$ est pair, mais pour utiliser l'hypothèse de récurrence il faudrait disposer de l'information que n' est pair, ce qui n'est pas possible.

Un essai transformé

En revanche, une démonstration reposant sur le principe de récurrence de la propriété d'être pair permettra de ne considérer que les nombres pairs, puisqu'elle demande de prouver que $S(S(x))+p$ est pair x et p sont pairs, en utilisant l'hypothèse de récurrence que $x + p$ est pair si x et p le sont. Ici l'utilisation de l'hypothèse de récurrence est directe : l'application du principe d'induction fournit également l'hypothèse que x est pair.

La session suivante reproduit ce raisonnement.

```
Restart.
 intros n p Heven_n; elim Heven_n.
  ...
  n : nat
  p : nat
  Heven_n : even n
  ============================
   even p → even (0+p)
```

Ce but correspond bien sûr au premier constructeur du prédicat inductif `even`, c'est la raison pour laquelle `x` a été remplacé par `0`. Ce but se résout automatiquement car il se simplifie en une implication de la forme $A \rightarrow A$.

```
 trivial.
  ...
  ============================
   ∀ n0:nat, even n0 →(even p → even (n0+p))→ even p →
     even (S (S n0) + p)
```

Nous pouvons introduire les différentes parties du but dans le contexte en leur donnant un nom significatif, puis simplifier la conclusion, toujours en accord avec la définition de `plus`, avec les commandes suivantes :

```
 intros x Heven_x Hrec Heven_p; simpl.
  ...
  x : nat
  Heven_x : even x
  Hrec : even p → even (x+p)
  Heven_p : even p
  ============================
   even (S (S (x+p)))
```

Nous pouvons alors utiliser directement le deuxième constructeur de `even`, dont nous rappelons le type :

plus_2_even : ∀ n:nat, even n → even (S (S n))

Pour utiliser ce théorème il faut bien sûr produire une preuve que $x + p$ est pair, mais l'hypothèse de récurrence `Hrec` puis l'hypothèse `Heven_p` fournissent directement ce résultat. Les commandes suivantes permettent donc de terminer la preuve.

```
 apply plus_2_even; auto.
Qed.
```

Exercice 9.9 Démontrer qu'un nombre pair est le double d'un autre nombre.

Exercice 9.10 Démontrer que le double d'un nombre est toujours pair (le double d'un nombre sera obtenu à l'aide de la fonction `mult2` définie page 192 ; la démonstration requiert une récurrence usuelle sur les nombres naturels), puis montrer que le carré d'un nombre pair est toujours pair, cette fois-ci en utilisant une récurrence sur la propriété de parité.

9.3.4 * Récurrence sur `le`

Le principe de récurrence associé au prédicat `le` est le suivant :

```
Open Scope nat_scope.
Check le_ind.
```

le_ind
   : ∀ (n:nat)(P:nat→Prop),
      P n →
      (∀ m:nat, n ≤ m → P m → P (S m))→
      ∀ n0:nat, n ≤ n0 → P n0

C'est ce principe qui a guidé la définition imprédicative de l'ordre $\leq$ sur $\mathbb{N}$ (page 163). Les démonstrations par récurrence sur cette définition inductive sont très proches des définitions par récurrence sur la structure des nombres entiers, puisque le cas de récurrence est également une démonstration de conservation de la propriété cherchée vis-à-vis du constructeur `S`. En revanche le cas de base change, puisqu'il ne s'agit plus de commencer avec le nombre zéro, mais avec un nombre n arbitraire.

Exercice 9.11 Réécrire la preuve ci-dessous, en utilisant la tactique `apply` au lieu de `elim` :

```
Theorem lt_le : ∀n p:nat, n < p → n ≤ p.
Proof.
 intros n p H; elim H; repeat constructor; assumption.
Qed.
```

Exercice 9.12 * Prouver par récurrence sur `le` que la définition inductive de `le` en *Coq* implique notre définition imprédicative (donnée en section 6.5.4 page 163) :

```
le_my_le : ∀n p:nat, n ≤ p → my_le n p.
```

Exercice 9.13 * Sans regarder les sources de *Coq*, prouver par récurrence la transivitivité de `le` :

```
le_trans' : ∀n p q:nat, n ≤ p → p ≤ q → n ≤ q.
```

À titre de comparaison, on prouvera directement (c'est à dire sans utiliser l'équivalence entre `le` et `my_le`) le théorème :

```
my_le_trans : ∀n p q:nat, my_le n p → my_le p q → my_le n q.
```

Exercice 9.14 (J.F. Monin) ** On considère la définition suivante de $\leq$:

$n \leq m$ si $\exists x \in \mathbb{N}|\ x + n = m$

Nous pouvons l'écrire sans expliciter le quantificateur existentiel :

```
Inductive le_diff (n:nat)(m:nat) : Prop :=
    le_d : ∀x:nat, x+n = m → le_diff n m.
```

On peut interpréter x comme la hauteur d'un arbre de preuve de " $n \leq m$ ". Montrer l'équivalence entre `le` et `le_diff`.

Exercice 9.15 ** On aurait pu définir la relation d'ordre $\leq$ sur `nat` de la façon suivante :

```
Inductive le' : nat→nat →Prop :=
  | le'_0_p : ∀p:nat, le' 0 p
  | le'_Sn_Sp : ∀n p:nat, le' n p → le' (S n)(S p).
```

Montrer en *Coq* que les prédicats `le` et `le'` sont équivalents.

Exercice 9.16 ** Une deuxième définition des listes ordonnées peut être donnée par la notion suivante :

```
Definition sorted' (A:Set)(R:A→A→Prop)(l:list A) :=
  ∀(l1 l2:list A)(n1 n2:A),
    l = app l1 (cons n1 (cons n2 l2))→ R n1 n2.
```

Démontrer que les prédicats `sorted` et `sorted'` sont équivalents.

Exercice 9.17 ** Comment peut-on traduire la définition inductive d'un prédicat ou d'une proposition en une définition imprédicative ? Proposer une méthode, et confrontez-la avec les exemples déjà commentés (pages 158 et suivantes), et essayer de nouveaux exemples (comme `even`). Pour chaque cas, vous devrez prouver l'équivalence entre la définition inductive et la définition imprédicative.

Exercice 9.18 (H. Südbrock) ** Compléter le développement suivant :

```
Section weird_induc_proof.

 Variable P : nat→Prop.
 Variable f : nat→nat.
```

```
 Hypothesis f_strict_mono : ∀n p:nat, lt n p → lt (f n)(f p).
 Hypothesis f_0 : lt 0 (f 0).

 Hypothesis P0 : P 0.
 Hypothesis P_Sn_n : ∀n:nat, P (S n)→ P n.
 Hypothesis f_P : ∀n:nat, P n → P (f n).

 Theorem weird_induc : ∀n:nat, P n.

End weird_induc_proof.
```

Il est conseillé de prouver quelques lemmes avant d'entamer la preuve du théorème proprement dit. Nous dirions même plus : l'intérêt principal de cet exercice consiste dans le choix de ces lemmes.

Exercice 9.19 * Cet exercice fait suite à l'exercice 9.5 page 247. Voici une seconde définition des expressions bien parenthésées, montrer qu'elle est équivalente à la précédente :

```
Inductive wp' : list par → Prop :=
| wp'_nil : wp' nil
| wp'_cons : ∀l1 l2:list par, wp' l1 → wp' l2 →
                  wp' (cons open (app l1 (cons close l2))).
```

Exercice 9.20 * Cet exercice fait suite à l'exercice 9.19. Voici une troisième définition des expressions bien parenthésées, montrer qu'elle est équivalente aux précédentes :

```
Inductive wp'' : list par → Prop :=
| wp''_nil : wp'' nil
| wp''_cons :
  ∀l1 l2:list par, wp'' l1 → wp'' l2 →
      wp'' (app l1 (cons open (app l2 (cons close nil)))).
```

Exercice 9.21 ** Cet exercice fait suite à l'exercice 9.20. Voici une fonction qui reconnaît les expressions bien parenthésées en comptant les parenthèses ouvertes et pas encore fermées.

```
Fixpoint recognize (n:nat)(l:list par){struct l} : bool :=
  match l with
    nil ⇒ match n with O ⇒ true | _ ⇒ false end
  | cons open l' ⇒ recognize (S n) l'
  | cons close l' ⇒
    match n with O ⇒ false | S n' ⇒ recognize n' l' end
  end.
```

Démontrer le théorème suivant :

```
recognize_complete_aux :
  ∀l:list par, wp l →
  ∀(n:nat)(l':list par),
  recognize n (app l l') = recognize n l'.
```

En déduire le théorème principal suivant :

```
recognize_complete :
  ∀l:list par, wp l → recognize 0 l = true.
```

Exercice 9.22 *** Cet exercice fait suite à l'exercice 9.21. Démontrer que la fonction `recognize` ne reconnaît que les expressions bien parenthésées. Plus précisément :

```
recognize_sound : ∀l:list par, recognize 0 l = true → wp l.
```

Indication : on pourra démontrer que si “ `recognize` n l ” vaut `true` alors la chaîne “ `app` l_n l ” est bien parenthésée, où l_n est la chaîne composée de n parenthèses ouvrantes. Pour cela, on sera amené à démontrer plusieurs théorèmes sur les concaténations de listes.

Exercice 9.23 *** Cet exercice fait suite aux exercices 9.7 et 9.20. On se donne la fonction d'analyse syntaxique suivante :

```
Fixpoint parse (s:list bin)(t:bin)(l:list par){struct l}
 : option bin :=
  match l with
  | nil ⇒ match s with nil ⇒ Some t | _ ⇒ None end
  | cons open l' ⇒ parse (cons t s) L l'
  | cons close l' ⇒
    match s with
    | cons t' s' ⇒ parse s' (N t' t) l'
    | _ ⇒ None
    end
  end.
```

Démontrer que cet analyseur syntaxique est correct et complet :

```
parse_complete :
 ∀l:list par, wp l → parse nil L l ≠ None.

parse_invert:
  ∀(l:list par)(t:bin),
      parse nil L l = Some t → bin_to_string' t = l.

parse_sound:
 ∀(l:list par)(t:bin), parse nil L l = Some t → wp l.
```

Pour démontrer la complétude, il est conseillé de suivre la structure proposée par la définition inductive de `wp'`.

9.4 * Relations inductives et fonctions

Il est parfois difficile de représenter une fonction (au sens mathématique) f par un terme *Coq* ; l'exigence de terminaison forte de toute réduction issue d'un terme *Coq* conduit à une limitation du pouvoir expressif du langage *Gallina*. En particulier, les fonctions partielles sont compliquées à définir, même en utilisant le type `option`. Une possibilité de décrire en *Coq* une fonction f de A vers B est de donner une caractérisation logique de l'ensemble des couples $(x, f(x))$, par exemple sous la forme d'une définition inductive d'un prédicat de type $A \rightarrow B \rightarrow$`Prop`. Un écueil doit cependant être évité avec soin : donner une caractérisation trop lâche du graphe de f, que satisferaient des couples (x, y) où x ne serait pas dans le domaine de f, ou qui autoriseraient des couples (x, y), (x, y'), avec $y \neq y'$.

NEW: Merci Gérard : avec quelle égalité ?

L'utilisation de définitions inductives permet donc de relâcher les contraintes sur les fonctions à décrire. L'avantage principal est que l'on pourra décrire des fonctions dont la terminaison n'est pas garantie. Par exemple, cet aspect joue un rôle majeur dans la description de langages de programmation : la fonction qui prend en entrée une donnée et un programme (dans un langage de programmation Turing-complet) et retourne le résultat de l'exécution de ce programme sur cette donnée ne peut pas être écrite en *Coq*, parce que cela impliquerait que l'on sache résoudre le problème de l'arrêt.

Pour représenter une fonction f à k arguments, on introduit le prédicat inductif P_f à $k+1$ arguments qui met en relation les k valeurs en entrée avec une valeur en sortie. Ce prédicat est défini par une collection de constructeurs couvrant tous les cas apparaissant dans la fonction. Pour chacun de ces cas, on décrit la forme des données en entrée. Ces données en entrée apparaîtront dans la conclusion du constructeur, ensuite on détermine les conditions qui expriment que l'on est dans ce cas, et on les met en hypothèse du constructeur. Ensuite, on remplace les appels récursifs de la forme " $f\ t_1\ \ldots t_k$ " par des propositions " $P_f\ t_1\ \ldots t_k\ y$ " où y est une variable « fraîche » . Enfin on ajoute les quantifications universelles qui suffisent pour que le constructeur soit bien typé. Il est également possible de représenter une fonction à k arguments par un prédicat inductif à $k+p$ arguments si la valeur retournée doit être un multiplet réunissant p composantes.

9.4.1 Représentation de la fonction factorielle

Considérons par exemple la fonction factorielle. Une possibilité de l'écrire en *OCAML* est la suivante :

```
let rec fact = function
  0 -> 1
| n -> n*(fact (n-1));;
```

Remarquons que cette fonction — définie sur le type `int` — ne termine pas dès qu'on lui donne un argument négatif. Nous allons décrire cette fonction récursive par le prédicat inductif `Pfact` qui reliera deux valeurs entières. Afin de refléter au mieux les propriétés du programme *OCAML*, nous considérons une relation binaire sur `Z` et non `nat`, où les problèmes de terminaison seraient artificiellement occultés. Le premier constructeur correspond à la première clause de filtrage de la fonction, qui détermine un premier cas de calcul. Ce constructeur fait apparaître le prédicat `Pfact` avec 0 en premier argument et 1 en second argument, puisque $1 = 0!$.

```
Pfact0 : Pfact 0 1.
```

Le deuxième cas qui apparaît dans la définition de `fact` met en jeu un appel récursif. Nous allons procéder progressivement. Premièrement nous exprimons que la valeur en entrée est reliée à la valeur en sortie :

```
 Pfact n (n*(fact (n-1)))
```

Ensuite nous ajoutons les conditions dans lesquelles ce cas sera utilisé. Il faut se souvenir que l'on n'arrive à la deuxième clause de filtrage que si la première ne s'applique pas, notre constructeur s'étoffe donc de la condition suivante :

```
 n ≠ 0 → Pfact n (n*(fact (n-1)))
```

Ensuite, nous remplaçons les appels récursifs de la fonction par une prémisse reposant sur le prédicat inductif `Pfact` et utilisant une nouvelle variable pour recevoir la valeur de l'appel récursif.

```
 n ≠ 0 → Pfact (n-1) v → Pfact n (n*v)
```

Enfin, nous quantifions universellement les différentes variables, ce qui donne le constructeur suivant :

```
Pfact1 : ∀n v:Z, n≠0 → Pfact (n-1) v → Pfact n (n*v).
```

Pour finir, la relation `Pfact` sera donc décrite par la définition inductive suivante :

```
Open Scope Z_scope.
Inductive Pfact : Z→Z→Prop :=
  Pfact0 : Pfact 0 1
| Pfact1 : ∀n v:Z, n ≠ 0 → Pfact (n-1) v → Pfact n (n*v).
```

Nous pouvons attribuer un sens à la proposition “ `Pfact n m` ” : *le calcul de `fact n` termine et retourne `m`.* Pour un `n` donné, il n'est pas toujours possible de trouver une valeur `m` telle que “ `Pfact n m` ” soit satisfaite, ce qui est conforme au comportement de la fonction `fact` écrite en *OCAML*.

Le prédicat inductif `Pfact` peut être utilisé pour démontrer que la valeur de la fonction `fact` est calculable pour certaines valeurs particulières de `m`. Par exemple, on peut vérifier l'égalité $3! = 6$ par la démonstration suivante :

```
Theorem pfact3 : Pfact 3 6.
Proof.
 apply Pfact1 with (n := 3)(v := 2).
```

Cette étape engendre deux buts, le premier demande de vérifier que trois n'est pas zéro et le second demande de vérifier l'égalité $2! = 2$. Le premier des deux buts se résout par `discriminate`; pour le second, nous pouvons réitérer le procédé. Chaque fois il faut fournir les arguments du constructeur `Pfact1` parce que la tactique `apply` ne fait pas les opérations nécessaires pour accepter que 6 est le même nombre que 3×2. Pour donner ces arguments nous pouvons utiliser la variante « `apply with` » ou tout simplement appliquer une instanciation du théorème obtenue par application usuelle. Le reste de la démonstration complète de `pfact3` peut se décrire avec la séquence de tactiques suivantes :

```
 discriminate.
 apply (Pfact1 2 1).
 discriminate.
 apply (Pfact1 1 1).
 discriminate.
 apply Pfact0.
Qed.
```

On peut également utiliser le prédicat `Pfact` pour caractériser le domaine de définition de la fonction étudiée. Ainsi, nous pouvons exprimer et démontrer les théorèmes indiquant que la fonction `fact` étudiée termine exactement pour les nombres supérieurs ou égaux à zéro.

```
Theorem fact_def_pos : ∀x y:Z, Pfact x y →  0 ≤ x.
 Proof.
 intros x y H; elim H.
```

L'élimination de l'hypothèse `(H: Pfact x y)` conduit à une preuve par récurrence sur les termes de ce type. Le premier cas correspond au premier constructeur. Dans ce cas `x` vaut 0 et la démonstration est aisée.

```
 auto with zarith.
```

Dans le deuxième cas, le but correspond à l'appel récursif. Ce but est plus lisible après que l'on ait introduit les différents éléments dans le contexte.

```
intros n v Hneq0 HPfact Hrec.
```

...
H : Pfact x y
n : Z
v : Z
Hneq0 : n $\neq$ *0*
HPfact : Pfact (n-1) v
Hrec : $0 \leq n\text{-}1$
============================
$0 \leq n$

```
omega.
Qed.
```

L'hypothèse `Hrec` donne suffisamment d'information pour que la tactique `omega` sache conclure.

NEW: faire une respiration ici, sinon on ne voit qu'on change de théorème, voir nouvelle structure version anglaise

Démontrer l'autre sens, c'est-à-dire que pour tout x positif il existe un y tel que la proposition " `Pfact` x y " soit satisfaite, est plus difficile. Le type `Z` ne permet pas de faire des raisonnements par récurrence au sens usuel, parce que ce type n'est pas récursif. En revanche, le type `positive` utilisé pour représenter les nombres strictement positifs est bien récursif, mais il ne permet de prouver une propriété $P(x)$ qu'en utilisant l'hypothèse de récurrence $P(x/2)$ (pour $x > 1$). Une meilleure solution est d'utiliser une récurrence où l'on doit prouver $P(x)$ en utilisant des hypothèses de récurrence $P(y)$ pour tous les $0 \leq y < x$. Les bibliothèques de *Coq* fournissent un prédicat `Zwf` défini de la façon suivante :

```
Definition Zwf (c x y:Z) := c ≤ x ∧ c ≤ y ∧ x < y.
```

La bibliothèque `Zwf` contient également deux théorèmes monrtrant d'une part que `Zwf` est bien fondé et d'autre part que l'on peut faire des raisonnements par récurrence à l'aide des relations bien fondées (cette notion sera approfondie en section 16.2) :

```
Zwf_well_founded
   : ∀ c:Z, well_founded (Zwf c)

well_founded_ind
   : ∀ (A:Set)(R:A→A→Prop),
       well_founded R →
       ∀ P:A → Prop,
        (∀ x:A, (∀ y:A, R y x → P y)→ P x)→
        ∀ a:A, P a
```

Ce principe nous permettra de raisonner par récurrence sur l'ensemble des nombres entiers positifs en nous autorisant à utiliser l'hypothèse de récurrence sur tous les nombres strictement plus petits que le nombre étudié. Notre démonstration va donc se faire de la façon suivante :

```
Theorem Zle_Pfact : ∀x:Z, 0 ≤ x → ∃y:Z | Pfact x y.
Proof.
 intros x0.
 elim x0 using (well_founded_ind (Zwf_well_founded 0)).
 intros x Hrec Hle.
  ...
  Hrec : ∀ y:Z, Zwf 0 y x → 0 ≤ y → ∃ y0:Z | Pfact y y0
  Hle : 0 ≤ x
  ============================
```

$\exists$ *y:Z | Pfact x y*

Nous devons faire apparaître ici les deux cas présents dans la fonction factorielle. En fait, on peut décomposer l'hypothèse `Hle` en deux cas : soit `0<x`, soit `0=x`. Ceci se fait avec un théorème que l'on retrouve dans les bibliothèques à l'aide de la commande `SearchPattern` décrite en section 6.1.3.

```
SearchPattern (_ < _ ∨ _ = _).
```

Zle_lt_or_eq: $\forall$ *n m:Z,* $n \leq m \to n < m \vee n = m$

```
elim (Zle_lt_or_eq _ _ Hle).
```

Cette tactique engendre deux buts. Le second correspond bien au cas de base de `Pfact` et se démontre aisément, par exemple avec les tactiques suivantes :

```
2:intros Heq; rewrite <- Heq; exists 1; constructor.
```

Le premier but a la forme suivante :

...
x : Z
Hrec : $\forall$ *y:Z, Zwf 0 y x* $\to 0 \leq y \to$ $\exists$ *y0:Z | Pfact y y0*
Hle : $0 \leq x$
============================
$0 < x \to$ $\exists$ *y:Z | Pfact x y*

Or nous savons que si “ `Pfact x y` ” était prouvable, cette preuve utiliserait une démonstration de “ `Pfact (x-1) v` ” pour une certaine valeur `v`. Cette démonstration peut être obtenue à l'aide de l'hypothèse de récurrence.

```
intro Hlt; elim (Hrec (x-1)).
```

La tactique “ `elim (Hrec (x-1))` ” engendre trois buts, le premier fournit une valeur `x1` et une preuve de “ `Pfact (x - 1) x1` ” et nous demande de trouver une valeur `v` et de démontrer “ `Pfact x v` ”.

...
Hrec : $\forall$ *y:Z, Zwf 0 y x* $\to 0 \leq y \to$ $\exists$ *y0:Z | Pfact y y0*
Hle : $0 \leq x$
Hlt : $0 < x$
============================
$\forall$ *x1:Z, Pfact (x-1) x1* $\to$ $\exists$ *y:Z | Pfact x y*

Nous savons que la valeur pour `y` est `x*x1`. Nous exprimons ceci par la tactique suivante :

```
intros x1 Hfact; exists (x*x1); apply Pfact1; auto with zarith.
```

Le deuxième but impose de vérifier que `x-1` est bien un prédécesseur de `x` pour la relation `Zwf`. Ce but se résout aisément avec la tactique suivante :

```
unfold Zwf; omega.
```

Le troisième but impose de vérifier que `x-1` est lui aussi supérieur à zéro. Ceci se résout également par la tactique `omega`, ce qui termine la preuve.

Nous disposons donc de la relation inductive `Pfact` qui décrit de façon logique le graphe d'une fonction *au sens mathématique*. Il est maintenant possible de faire des démonstrations par récurrence sur cette relation inductive et donc de raisonner sur le graphe de cette fonction.

Exercice 9.24 *** Cet exercice fait suite à l'exercice 9.7 page 248. La définition inductive suivante donne une présentation relationnelle d'une fonction d'analyse syntaxique pour les expressions bien parenthésées. Intuitivement, la proposition " `parse` l_1 l_2 t " signifie « l'analyse syntaxique de la chaîne de caractères l_1 laisse l_2 à analyser et construit l'expression t ».

```
Inductive parse_rel : list par → list par → bin → Prop :=
  | parse_node :
    ∀(l1 l2 l3:list par)(t1 t2:bin),
      parse_rel l1 (cons close l2) t1 → parse_rel l2 l3 t2 →
      parse_rel (cons open l1) l3 (N t1 t2)
  | parse_leaf_nil : parse_rel nil nil L
  | parse_leaf_close :
      ∀l:list par, parse_rel (cons close l)(cons close l) L.
```

Démontrer les théorèmes suivants :

```
parse_rel_sound_aux :
   ∀(l1 l2:list par)(t:bin),
     parse_rel l1 l2 t → l1 = app (bin_to_string t) l2.

 parse_rel_sound :
   ∀l:list par, (∃t:bin | parse_rel l nil t)→ wp l.
```

9.4.2 ** Représentation de la sémantique d'un langage

La représentation d'une fonction f de A vers B par une définition inductive de la relation associée est particulièrement utile si l'appartenance au domaine de définition est indécidable. On ne peut plus définir d'algorithme associant à tout a de A un terme de type B, ni même de type " `option` B ".

Un cas typique est la sémantique d'un langage de programmation. Si l'on veut représenter cette sémantique par une fonction dont les entrées sont l'état initial des variables et le programme à exécuter et dont le résultat est l'état final des variables, cette fonction ne peut pas être décrite en *Coq* dès que le langage est complet au sens de Turing, car alors le problème de l'arrêt est indécidable.

En revanche, on peut parfaitement décrire la relation entre les entrées et les sorties comme un prédicat inductif.

Pour illustrer notre propos, nous allons considérer un petit langage impératif manipulant des expressions booléennes et entières, où les variables ont toujours une valeur entière et contenant uniquement quatre instructions : l'instruction `Skip` ne fait rien, l'instruction " `Assign` x e " affecte la valeur de l'expression e à la variable x, l'instruction " `Sequence` i_1 i_2 " exécute d'abord l'instruction i_1 puis l'instruction i_2, enfin l'instruction " `WhileDo` b i " exécute l'instruction `i` tant que l'expression booléenne `b` est vraie. Pour définir ces instructions dans *Coq*, nous allons supposer l'existence de types pour les expressions et les variables et décrire le type des instructions comme un type inductif :

```
Section little_semantics.
Variables Var aExp bExp : Set.
Inductive inst : Set :=
| Skip : inst
| Assign : Var→aExp→inst
| Sequence : inst→inst→inst
| WhileDo : bExp→inst→inst.
```

Pour décrire la sémantique de ce langage de programmation, nous allons également supposer que nous disposons d'un type d'états appelé `state` et de fonctions `update`, `evalA` et `evalB`. La fonction `update` retourne l'état après l'affectation d'une valeur entière à une variable ; cette fonction est partielle parce que la mise à jour de l'état pour une variable non initialisée n'est pas définie, ce que nous représentons à l'aide du type `option`. La fonction `evalA` décrit la valeur d'une expression arithmétique dans un état ; ici encore il s'agit d'une fonction partielle car l'expression arithmétique peut contenir une variable pour laquelle l'état ne prévoit pas de valeur. La function `evalB` décrit la valeur d'une expression booléenne.

```
Variables
  (state : Set)
  (update : state→Var→Z → option state)
  (evalA : state→aExp → option Z)
  (evalB : state→bExp → option bool).
```

La définition sémantique de notre langage peut alors être donnée par une définition inductive où chaque constructeur décrit l'un des comportements possibles de l'une des instructions en suivant le style de la sémantique naturelle [56]. Il y quatre instructions et cinq constructeurs pour la relation d'évaluation, car l'instruction `WhileDo` peut avoir deux comportements différents.

```
Inductive exec : state→inst→state→Prop :=
| execSkip : ∀s:state, exec s Skip s
| execAssign :
   ∀(s s1:state)(v:Var)(n:Z)(a:aExp),
     evalA s a = Some n → update s v n = Some s1 →
```

```
     exec s (Assign v a) s1
| execSequence :
   ∀(s s1 s2:state)(i1 i2:inst),
    exec s i1 s1 → exec s1 i2 s2 →
    exec s (Sequence i1 i2) s2
| execWhileFalse :
   ∀(s:state)(i:inst)(e:bExp),
    evalB s e = Some false → exec s (WhileDo e i) s
| execWhileTrue :
   ∀(s s1 s2:state)(i:inst)(e:bExp),
    evalB s e = Some true →
    exec s i s1 →
    exec s1 (WhileDo e i) s2 →
    exec s (WhileDo e i) s2.
```

9.4.3 ** Une démonstration en sémantique

Même en l'absence d'une meilleure connaissance de l'état représenté par le type `state` et des fonctions `update`, `evalA`, `evalB`, cette description sémantique permet déjà de décrire certaines propriétés du langage. Par exemple, on peut démontrer une condition suffisante pour qu'une propriété soit assurée après l'exécution d'une boucle : il suffit que cette propriété soit « invariante » et qu'elle soit satisfaite avant l'exécution de la boucle. En outre on sait qu'après l'exécution d'une boucle le test de cette boucle est nécessairement faux. Pour les connaisseurs, la propriété `P` joue le rôle des invariants utilisés en calcul de précondition la plus faible [43, 53, 39] :

```
Theorem HoareWhileRule :
 ∀(P:state→Prop)(b:bExp)(i:inst)(s s':state),
   (∀s1 s2:state,
      P s1 → evalB s1 b = Some true → exec s1 i s2 → P s2)→
   P s → exec s (WhileDo b i) s' →
   P s' ∧ evalB s' b = Some false.
```

Un essai manqué

Notre premier réflexe est de faire directement une démonstration par récurrence sur le prédicat `exec`, en introduisant dans le contexte jusqu'à la bonne hypothèse, puis en utilisant la tactique `elim`.

```
intros P b i s s' H Hp Hexec; elim Hexec.
```

La tactique `elim` engendre cinq buts, ce qui est prévisible puisque le prédicat inductif `exec` a cinq constructeurs, mais le premier but a la forme suivante :

...
H : ∀ s1 s2:state,
P s1 → evalB s1 b = Some true → exec s1 i s2 → P s2

```
Hp : P s
Hexec : exec s (WhileDo b i) s'
============================
 ∀ s0:state, P s0 ∧ evalB s0 b = Some false
```

Dans ce but, nous devons vérifier que la propriété `(P s0)` est satisfaite pour un état `s0` arbitraire. C'est visiblement impossible. On peut bien sûr rendre le problème moins difficile en faisant réapparaître les hypothèses significatives dans le but à l'aide de la tactique `generalize` comme nous l'avons déjà indiqué dans la section 7.2.7 :

```
Restart.
 intros P b i s s' H Hp Hexec; generalize H Hp; elim Hexec.
```

```
  ...
============================
 ∀ s0:state,
   (∀ s1 s2:state,
      P s1 → evalB s1 b = Some true → exec s1 i s2 → P s2)→
   P s0 → P s0 ∧ evalB s0 b = Some false
```

Si cette variante rend le membre de gauche de la conjonction facilement prouvable, nous n'avons toujours pas de solution pour le membre droit.

La première leçon à tirer de cet échec est qu'il faut laisser le maximum d'informations pertinentes dans le but au moment de faire la démonstration par récurrence à l'aide de la tactique `elim`. Ceci rejoint la recommandation déjà faite dans la section 7.2.7.

La deuxième leçon à tirer viendra d'une analyse du principe de récurrence associé au prédicat inductif. Ce principe de récurrence a la forme suivante :

```
Check exec_ind.
exec_ind
   : ∀ P:state→inst→state→Prop,
      (∀ s:state, P s Skip s)→
      ...
      ∀ (s:state)(i:inst)(s0:state), exec s i s0 → P s i s0
```

Si `Hexec` a le type “ `exec s (WhileDo b i) s'` ”, alors “ `elim Hexec` ” cherche d'abord à déterminer la propriété `P` en cherchant les instances des expressions `s`, “ `WhileDo b i` ”, et `s'` dans le but. Dans notre cas, elle trouve bien des instances de `s`, mais pas de “ `WhileDo b i` ”. En particulier, elle ne découvre pas que `b` et `i` qui sont utilisés dans le but sont en fait des « petits morceaux » de l'expression cherchée : ainsi le rôle que ces petits morceaux devraient prendre dans chacun des buts engendrés est oublié.

La conclusion de cette analyse est qu'il faut toujours éviter de faire une preuve par récurrence sur un prédicat inductif si l'un des arguments de ce prédicat inductif n'est pas une variable.

Un essai transformé

Pour permettre une démonstration par récurrence, nous allons donc d'abord faire apparaître une propriété équivalente dans laquelle le prédicat `exec` apparaît sous la bonne forme, en introduisant une variable `i'` qui a la propriété d'être égale à `(WhileDo b i)` :

```
Restart.
intros P b i s s' H.
cut
 (∀i':inst,
    exec s i' s' →
    i' = WhileDo b i → P s → P s' ∧ evalB s' b = Some false);
 eauto.
```

La tactique `eauto` utilisée à la fin de la tactique composée “ `cut ...` ” permet de montrer que notre énoncé initial est bien une conséquence du nouvel énoncé. Il ne reste donc que le nouvel énoncé à démontrer :

```
  ...
  H : ∀ s1 s2:state,
       P s1 → evalB s1 b = Some true → exec s1 i s2 → P s2
  ============================
     ∀ i':inst,
     exec s i' s' →
     i' = WhileDo b i → P s → P s' ∧ evalB s' b = Some false
intros i' Hexec; elim Hexec; try (intros; discriminate).
```

L'étape de récurrence va faire apparaître cinq buts, correspondant aux cinq constructeurs du prédicat `exec`. Dans trois de ces buts, l'instruction `i'` sera remplacée par une instruction différente de l'instruction `WhileDo` et une égalité contradictoire sera donc présente dans ces trois buts. Il est judicieux de faire suivre l'étape de démonstration par récurrence d'une tactique qui saura retrouver cette égalité contradictoire.

```
 2 subgoals
  ...
  ============================
  ∀ (s0:state)(i0:inst)(e:bExp),
     evalB s0 e = Some false →
     WhileDo e i0 = WhileDo b i →
     P s0 → P s0 ∧ evalB s0 b = Some false
```

Il ne reste donc que deux buts, correspondant aux deux cas d'exécution de l'instruction `WhileDo`. Le premier de ces deux buts correspond au cas où l'expression booléenne s'évalue à `false`.

Ce cas est facile à résoudre, mais il faut néanmoins utiliser l'injectivité du constructeur `WhileDo` pour établir l'équivalence entre la propriété à prouver : “ `evalB s0 b = Some false` ” et la prémisse “ `evalB s0 e = Some false` ”.

```
intros s0 i0 e Heval Heq; injection Heq; intros H1 H2.
match goal with
| [id:(e = b) |- _ ] ⇒ rewrite <- id; auto
end.
```

Le dernier but a la forme suivante.

...
H : $\forall$ *s1 s2:state,*
    $P\ s1 \to evalB\ s1\ b = Some\ true \to exec\ s1\ i\ s2 \to P\ s2$
i' : inst
Hexec : exec s i' s'
============================
$\forall$ *(s0 s1 s2:state)(i0:inst)(e:bExp),*
  $evalB\ s0\ e = Some\ true \to$
  $exec\ s0\ i0\ s1 \to$
  $(i0 = WhileDo\ b\ i \to P\ s0 \to P\ s1 \land evalB\ s1\ b = Some\ false)\to$
  $exec\ s1\ (WhileDo\ e\ i0)\ s2 \to$
  $(WhileDo\ e\ i0 = WhileDo\ b\ i \to$
  $P\ s1 \to P\ s2 \land evalB\ s2\ b = Some\ false)\to$
  $WhileDo\ e\ i0 = WhileDo\ b\ i \to$
  $P\ s0 \to P\ s2 \land evalB\ s2\ b = Some\ false$

Ici encore, l'égalité " `WhileDo e i0 = WhileDo b i` " joue un rôle important pour permettre la correspondance entre la conclusion et les différentes prémisses. Les commandes suivantes permettent d'introduire les hypothèses, de retrouver l'hypothèse d'égalité et de la décomposer en deux égalités plus simples `H'` et `H''`, puis d'appliquer autant que possible les hypothèses. Lorsque plus aucune hypothèse ne s'applique, la réécriture par les égalités simples permet de finir avec des buts facilement prouvables.

```
intros;
 match goal with
 | [id:(_ = _) |- _ ] ⇒ injection id; intros H' H''
 end.
repeat match goal with
       | [id:_ |- _ ] ⇒ eapply id; eauto
       end; try rewrite <- H'; try rewrite <- H''; assumption.
Qed.
```

Exercice 9.25 ** Il existe une méthode qui permet d'éviter de passer par une égalité, en faisant simplement réapparaître " `WhileDo b i` " dans le but. Trouver cette méthode et redémontrer ce théorème.

Exercice 9.26 * Démontrer que si b est une expression booléenne qui s'évalue à `true` dans l'état s, alors l'exécution de l'instruction (`WhileDo` b `Skip`) ne termine jamais dans cet état :

```
∀(s s':state)(b:bExp),
   exec s (WhileDo b Skip) s' → evalB s b = Some true → False.
```

Exercice 9.27 ** Démontrer la règle de Hoare pour la séquence.

9.5 * Comportements élaborés de la tactique elim

9.5.1 Instancier les arguments

Nous avons indiqué en section 7.1.3 que plusieurs mécanismes se succédaient dans la tactique `elim` pour la rendre conviviale. En présence de types inductifs dépendants, et en particulier en présence de prédicats inductifs, ces mécanismes sont encore plus élaborés.

Attardons-nous sur le cas où la tactique a la forme `elim` H et H a le type suivant :

$$H : \forall x_1 \dots x_k, P_1 \to \cdots P_l \to (T\ a_1 \cdots a_n).$$

Supposons que les expressions a_1, …, a_n ne sont pas des variables mais des termes complexes contenant les variables $x_1, \dots, x_k$. La tactique recherche alors des instances des expressions a_i dans le but de façon à déterminer les valeurs que doivent prendre les variables x_j. Si e_1, …, e_k sont les expressions ainsi déterminées, la tactique “ `elim` H ” est alors équivalente à la tactique “ `elim` $(H\ e_1\ \dots e_k)$ ”.

Pour comprendre ce comportement, utilisons un petit exemple basé sur une propriété inductive et une hypothèse inventées pour l'occasion :

```
Open Scope nat_scope.
Inductive is_0_1 : nat→Prop :=
  is_0 : is_0_1 0 | is_1 : is_0_1 1.
Hint Resolve is_0 is_1 .

Lemma sqr_01 : ∀x:nat, is_0_1 x → is_0_1 (mult x x).
 Proof.
  induction 1; simpl; auto.
 Qed.
```

Cherchons maintenant à démontrer la propriété suivante :

```
Theorem elim_example : ∀n:nat, n ≤ 1 → n*n ≤ 1.
Proof.
 intros n H.
  ...
  n : nat
  H : n ≤ 1
```

```
============================
```
*n*n ≤ 1*

Nous voulons étudier le comportement de la tactique “ `elim sqr_01` ”. Le type final de `sqr_01` est “ `is_0_1 (x * x)` ” qui contient l'argument $a_1 =$ “ `x * x` ” où se trouve la variable `x`, l'un des arguments dépendants du type complet. On trouve a_1 dans le but lorsque `x` vaut `n`. La tactique “ `elim sqr_01` ” est donc équivalente à la commande “ `elim (sqr_01 n)` ”. Le terme “ `sqr_01 n` ” est encore une fonction, mais avec un argument non dépendant et un but sera simplement ajouté pour cet argument. Enfin, deux buts sont engendrés pour les constructeurs de la propriété `is_0_1`. En résumé, la tactique “ `elim sqr_01` ” engendre les trois buts suivants :

```
elim sqr_01.
```
...
```
============================
```
0 ≤ 1

subgoal 2 is:
1 ≤ 1
subgoal 3 is:
is_ 0_ 1 n

Nous venons de décrire le comportement d'`elim` lorsqu'une occurrence de l'argument dépendant est effectivement trouvée. Si plusieurs occurrences sont trouvées, une est choisie (normalement la plus à gauche dans le but). Si la tactique choisit une mauvaise occurrence, l'utilisateur peut la guider à l'aide d'une instanciation des arguments où à l'aide de la tactique `pattern`.. Si aucune occurrence n'est trouvée on obtient un message d'erreur.

Si la propriété inductive a des arguments paramétriques, seuls les arguments non paramétriques font l'objet de recherche d'instance. Un exemple typique de ce comportement apparaît avec les théorèmes d'égalité, comme par exemple le théorème d'associativité de l'addition :

plus_ assoc : ∀ n m p:nat, n+(m+p) = n+m+p

Ce théorème est une fonction à trois arguments, n, m et p et le prédicat inductif qui apparaît dans le type final est la propriété `eq`. Cette propriété a elle-même trois arguments, mais les deux premiers sont paramétriques. Ce sont donc seulement les instances de `(plus (plus _ _) _)` qui sont cherchées dans le but courant.

9.5.2 Inversion

Les constructeurs d'un prédicat inductif P permettent d'effectuer des raisonnements positifs sur ce prédicat : *telle ou telle expression satisfait P parce* *que tel ou tel constructeur permet de le démontrer*. En revanche, les principes d'élimination permettent d'effectuer des raisonnements limitatifs : *si telle expression satisfaisait P, alors elle satisferait aussi tel prédicat Q*. Néanmoins une utilisation directe de `elim` n'est pas toujours la meilleure solution.

Reprenons par exemple la propriété `even` déjà utilisée dans la section 9.1 et décrite par la définition inductive suivante :

```
Print even.
Inductive even : nat→Prop :=
   O_even : even 0
  | plus_2_even : ∀ n:nat, even n → even (S (S n))
For even: Argument scope is [nat_scope]
For plus_2_even: Argument scopes are [nat_scope _]
```

Observons maintenant une démonstration que 1 n'est pas pair.

Un essai manqué

L'énoncé “ `~ even 1` ” est convertible en l'énoncé “ `(even 1) → False` ”. Nous pouvons donc considérer que nous disposons d'une hypothèse construite avec le type inductif `even`. Un premier réflexe est d'utiliser cet énoncé pour effectuer une preuve par récurrence sur le prédicat inductif, ce qui donne deux buts dont nous ne présentons que le premier.

```
Theorem not_1_even : ~even 1.
Proof.
 red; intros H; elim H.
  ...
 H : even 1
 ============================
  False
```

Ce but n'est pas plus facile à démontrer que le but existant avant la commande “ `elim H` ” : c'est le même. Cet échec est simple à comprendre : la tactique `elim` cherche avec quelle propriété `P` instancier le théorème `even_ind` engendré par la définition inductive, comme `1` n'apparaît pas dans le but, le prédicat construit par *Coq* est la fonction constante “ `fun x:nat ⇒ False` ”.

L'application de `even_ind` avec ce prédicat engendre deux buts, dont le premier est tout simplement l'application de `P` à `0`, ce qui donne le même but qu'avant, puisque la `P` est une fonction constante.

Un essai transformé

La tactique `inversion` permet d'éviter ce problème. Pour l'utiliser, il faut faire apparaître une hypothèse dont le type est inductif, ici en décomposant la négation.

```
Theorem not_1_even' : ~even 1.
Proof.
 unfold not; intros H.
 ...
  H : even 1
  ============================
   False

inversion H.
```

L'appel à “ `inversion H` ” termine cette preuve. Cette tactique applique le raisonnement suivant : une preuve arbitraire de “ `even 1` ” ne peut s'obtenir ni par le constructeur `O_even` (car $0 \neq 1$), ni par `plus_2_even`, car 1 ne peut se mettre sous la forme “ `S (S n)` ”. Cette analyse permet d'exclure toute preuve de `even 1`, ce qui permet de résoudre le but courant « par l'absurde ». Nous verrons page 279 comment cet argument intuitif est traduit dans le formalisme de *Coq*.

Dans cet exemple, la tactique `inversion` a entièrement résolu le but, mais ce n'est pas toujours le cas. En général, cette tactique ne fait que retrouver les constructeurs qui auraient pu s'appliquer et laisse un certain nombre de cas à traiter interactivement par l'utilisateur. Prenons par exemple le fragment de démonstration suivant :

```
Theorem plus_2_even_inv : ∀n:nat, even (S (S n))→ even n.
Proof.
 intros n H; inversion H.
  ...
  n : nat
  H : even (S (S n))
  n0 : nat
  H1 : even n
  H0 : n0 = n
  ============================
   even n
```

L'information ajoutée dans le but correspond au raisonnement suivant : *si la proposition “ `even (S (S n))` ” a été démontrée avec l'un des constructeurs de `even`, cela ne peut pas avoir été avec le constructeur `O_even`. Cela a nécessairement été fait avec le constructeur `plus_2_even`, instancié pour une certaine valeur `n0` et l'on doit alors avoir “ `S (S n0) = S (S n)` ”, ce qui équivaut à `n0=n` car le constructeur `S` est injectif et “ `even n0` ” doit également avoir été prouvé, ce qui est équivalent au fait que “ `even n` ” est prouvé.* Ces remarques justifient les hypothèses ajoutées dans le but. Grâce à ces hypothèses, le but devient aisé à démontrer. Si l'on observe l'énoncé `plus_2_even_inv`, on s'aperçoit qu'il a été obtenu à partir du constructeur `plus_2_even` tout simplement en inversant le sens de la flèche. C'est cette observation qui justifie le nom choisi pour la tactique.

**** Les dessous d'`inversion`** Nous allons tâcher de comprendre comment fonctionne la tactique `inversion` en reprenant de façon manuelle les démonstrations de `not_even_1` et `plus_2_even_inv`. Nous utilisons la tactique `elim`, mais avec un prédicat bien choisi.

Dans le premier cas, il nous suffit de construire un prédicat P tel que l'on puisse prouver les propositions suivantes :

- P `1` $\rightarrow$ `False`
- $\forall$`n:nat,even n` $\rightarrow$ P `n`

Un candidat naturel est la fonction « `fun (n:nat)` $\Rightarrow$ `n = 1` $\rightarrow$ `False` ». La première proposition se prouve immédiatement, la seconde par récurrence sur « `even n` ».

Ce prédicat peut en fait se construire interactivement à l'aide de `generalize` et `pattern`. Nous utilisons la possibilité de donner un argument négatif à `pattern` comme nous l'avons déjà fait en section 7.2.7.

```
Theorem not_even_1 : ~even 1.
 Proof.
 intro H.
 generalize (refl_equal 1).
  ...
  H : even 1
  ============================
   1=1 → False

 pattern 1 at -2.
  ...
  H : even 1
  ============================
  (fun n:nat ⇒ n = 1 → False) 1
```

Nous disposons alors d'un but permettant une récurrence sur `H` ; les appels à `discriminate` permettent d'évacuer des hypothèses de la forme « `0 = 1` » et « `S (S n) = 1` ».

Observons maintenant la démonstration de `plus_2_even_inv` :

```
Theorem plus_2_even_inv' : ∀n:nat, even (S (S n))→ even n.
Proof.
 intros n H.
  ...
  n : nat
  H : even (S (S n))
  ============================
   even n
```

Il est nécessaire de faire apparaître le but comme une propriété de “ `S (S n)` ”, puisque c'est ce nombre qui apparaît en argument de la propriété inductive. Nous le faisons de la même manière que dans le théorème précédent à l'aide des tactiques `generalize` et `pattern` :

```
generalize (refl_equal (S (S n))); pattern (S (S n)) at -2.
  ...
  n : nat
  H : even (S (S n))
  ============================
    (fun n0:nat ⇒ n0 = S (S n)→ even n)(S (S n))
```

Nous pouvons maintenant effectuer l'étape de récurrence sur `H` pour obtenir deux buts dont le premier a la forme suivante :

```
elim H.
...
============================
   0 = S (S n)→ even n
```

Ce but est résoluble par la tactique `discriminate`. Le deuxième but a la forme suivante (après l'introduction de variables et d'hypothèses) :

```
intros n0 H'0 H' H'1.
...
n0 : nat
H'0 : even n0
H' : n0=S (S n)→ even n
H'1 : S (S n0) = S (S n)
============================
even n
```

Par la tactique “ `injection H'1` ” nous pouvons obtenir l'égalité `n0=n`, puis — par réécriture — résoudre ce but.

Exercice 9.28 Montrer le résultat suivant :

```
~sorted le (1::3::2::nil)
```

Exercice 9.29 (Laurent Théry) * Démontrer qu'avec des timbres de 5 unités et des timbres de 3 unités, on peut payer toutes les valeurs supérieures à 8 unités. Ce problème est un cas particulier d'un problème connu sous le nom de « problème de Frobenius ».

Chapitre 10

* Les fonctions et leur spécification

Nous avons présenté de façon informelle des programmes certifiés dans le chapitre 2. Étant donnée une relation R de $A \rightarrow B \rightarrow$`Prop`, il faut construire une fonction qui à tout a de A associe un résultat b de type B accompagné d'une preuve de " $R\ a\ b$ " (un *certificat*).

Il peut y avoir deux approches pour définir des fonctions et fournir des preuves qu'elles satisfont une spécification donnée. Une approche est de définir ces fonctions avec un type qui est une *spécification faible* et de joindre des lemmes compagnons qui expriment les propriétés de la fonction. Dans cette approche, nous définition une fonction f de type $A \rightarrow B$ et nous démontrons un lemme dont l'énoncé est " $\forall a$`:`A`, R` a `(`f a`)` ". Les types que nous avons décrits dans les chapitres précédents suffisent pour cette approche. Une seconde approche est de donner directement une *spécification forte* à notre fonction : son type peut exprimer directement que son argument est de type A et que le résultat associé est la combinaison d'une valeur v de type b et d'une preuve que v satisfait " `R` a v ". Nous avons montré un tel type combiné en section 7.5.1 pour les racines carrées. Nous qualifierons également les fonctions et types rencontrés dans cette seconde approche de « fonctions bien spécifiées » et de « types riches ». Puisque le type du résultat doit indiquer comment ce résultat est relié à l'entrée, ce genre de spécification forte repose habituellement sur les types dépendants. De plus, nous utilisons naturellement des types inductifs pour décrire la combinaison de valeurs et de preuves.

Dans la première partie de ce chapitre, nous montrons comment utiliser les types inductifs pour construire des spécifications fortes. Nous montrons ensuite comment construire des fonctions fortement spécifiées. Nous étudions également l'approche alternative avec des spécifications faibles et des lemmes compagnons. Nous montrons certaines des difficultés qui peuvent êtres rencontrées dans cette approche. Dans une dernière partie, nous décrivons un exemple élaboré où nous considérons la divison euclidienne pour les nombres représentés en format binaire, en le traitant par les deux approches. Dans un premier temps nous donnons une fonction faiblement spécifée et les lemmes compagnons et dans un deuxième temps nous fournissons le développement d'une fonction bien spécifiée. Cette étude montre qu'il est aussi facile de développer des fonctions bien spécifiées que de démontrer la correction de fonction faiblement spécifiées.

10.1 Types inductifs pour les spécifications

Jusqu'à maintenant les types dépendants que nous avons construits étaient surtout de sorte `Prop`, les exceptions les plus notables étant `htree` (page 210) et `binary_word` (page 104). Dans ces deux derniers cas, tous les arguments des constructeurs étaient de sorte `Set`. Nous allons maintenant considérer des types de données (de sorte `Set`) dont les constructeurs peuvent prendre des preuves (termes de sorte `Prop`) en argument. Ces arguments-preuves ne sont pas utilisés dans les calculs mais pour exprimer que les données vérifient certaines propriétés.

10.1.1 Type « sous-ensemble »

Une forme de spécification très naturelle consiste en l'adjonction à un type d'un prédicat défini sur ce type, créant ainsi *le type des élément qui satisfont le prédicat*. Intuitivement ce nouveau type représente un « sous-ensemble » du type initial. Par exemple, la spécification « un nombre premier supérieur à n » , consiste en la donnée du type `nat` et du prédicat :

```
fun p:nat ⇒ n < p ∧ prime p
```

Une valeur certifiée répondant à cette spécification devrait contenir, sous une forme ou une autre, un composant « calculatoire » précisant comment obtenir p, et une justification, preuve que p est bien premier et supérieur à n. Cette spécification se décrit aisément à l'aide de la définition inductive suivante, fournie dans les bibliothèques de *Coq*.

```
Inductive sig (A:Set)(P:A→Prop) : Set :=
    exist : ∀x:A, P x → sig A P.
Implicit Arguments sig [A].
```

Le nom `sig` de ce type inductif fait référence à la notion théorique de type Σ [1]. Le constructeur `exist` prend deux arguments en plus des arguments paramétriques. L'argument `x` de type `A` est le composant calculatoire et l'argument non nommé de type " `P x` " est la justification. Le principe de récurrence associé à ce type inductif est le suivant :

sig_ind
    : ∀ (A:Set)(P:A→Prop)(P0:sig P → Prop),

1. Intuitivement, la quantification existentielle et les types sous-ensembles sont reliés aux types de sommes disjointes de la même manière que la quantification universelle est reliée au produit cartésien.

(∀ (x:A)(p:P x), P0 (exist P x p))→
∀ s:sig P, P0 s

Le système *Coq* fournit également une notation pour ce type inductif : une expression de la forme “ `sig (fun x : A ⇒ E)` ” s'écrit `{x : A | E}`. Par exemple, le type des nombres naturels qui sont plus grands que n et premiers peut s'écrire “ `{p:nat | n < p ∧ prime p}` ”.

Les lecteurs les plus attentifs auront reconnu l'extrême similitude entre ce type inductif et le type `ex`, utilisé pour la quantification existentielle. La seule différence importante entre les deux définitions est que `ex` est défini dans la sorte `Prop` tandis que `sig` est défini dans la sorte `Set`. Ceci a une influence sur la façon dont le système *Coq* engendre le principe de récurrence (voir section 15.1.5). La différence principale entre les deux types est qu'il est possible de construire une fonction de type “ `(sig P)→A` ” : un habitant de “ `sig P` ” « contient » un élément de `A`, que cette fonction permet de retrouver. En revanche, il est impossible de construire une telle fonction de type “ `(ex P) → A` ”. Intuitivement, si l'on dispose d'une expression de type “ `sig P` ”, on dispose d'un moyen de construire un élément du type `A` qui satisfait la propriété considérée. En revanche, si l'on dispose d'une preuve de `(ex P)` on connaît seulement l'existence d'un témoin pour la propriété `P` et on pourra utiliser un tel témoin (arbitraire) pour établir une autre proposition mais on ne dispose pas de procédé pour le construire. Cette distinction est importante : tous les objets dont le type est de sorte `Set` correspondent à des procédés de calculs et non seulement à des valeurs. Cette distinction qui renforce le concept de non-pertinence des preuves est traditionnellement utilisée pour l'extraction, comme nous le verrons dans le chapitre 11.

Par exemple, le type `sig` pourra être utilisé avec profit pour donner un type très précis à une fonction de division euclidienne `div_pair` sur les entiers relatifs qui retourne le couple du quotient et du reste avec l'assurance que ce couple satisfait la spécification [2].

```
Variable div_pair :
∀a b:Z, 0 < b →
 {p:Z*Z | a = (fst p)*b + snd p ∧ 0 ≤ snd p < b}.
```

Il s'agit donc du type d'une fonction qui prend en arguments deux valeurs entières dont la deuxième est strictement positive, et retourne le couple d'un quotient et d'un reste qui satisfont les propriétés attendues. Il faut toutefois se méfier de l'explication intuitive de « sous-type » qui est un peu fausse. Un habitant du type `{x:A | P}` n'est pas un habitant du type `A`. En revanche, on peut toujours le décomposer pour extraire un élément de type `A` qui satisfait la propriété `P`, à l'aide de la construction `match`. Par exemple, si nous voulons appliquer la fonction `div_pair` aux diviseurs de type `{b:Z | 0 < b }` nous pouvons le faire de la façon suivante :

2. Les exemples de ce chapitre utilisent tantôt le type `nat`, tantôt le type `Z` ; nous n'avons pas inclus dans le texte du livre les nombreuses ouvertures de portées `nat_scope` et `Z_scope` qui font partie des sources *Coq* de ce chapitre

```
Definition div_pair' (a:Z)(x:{b:Z | 0 < b}) : Z*Z :=
  match x with
  | exist b h ⇒ let (v, _) := div_pair a b h in v
  end.
```

Exercice 10.1 * Construire une fonction `extract` de type

```
∀(A:Set)(P:A→Prop), sig P → A
```

telle que l'on ait la propriété suivante :

```
  ∀(A:Set)(P:A→Prop)(y:{x:A | P x}),
        P (extract A (fun x:A ⇒ P x) y)
```

Démontrer cette propriété avec l'aide de *Coq*.

Exercice 10.2 * À l'aide de la fonction `extract` définie dans l'exercice précédent, décrire une fonction prenant les mêmes arguments en entrée que la fonction `div_pair'`, mais avec un résultat fortement spécifié.

Exercice 10.3 * Construire une fonction `sig_rec_simple` ayant le type suivant :

```
∀(A:Set)(P:A→Prop)(B:Set), (∀x:A, P x → B)→ sig P → B
```

10.1.2 Type sous-ensemble emboîté

Il peut être intéressant d'emboîter les types sous-ensembles, au sens où l'on veut exprimer par exemple que l'on prend les valeurs x pour lesquelles on sait construire une valeur y telle que $Q(x, y)$ soit vérifié. Cet emboîtement est symétrique de l'emboîtement de quantifications existentielles en logique. Ceci est rendu possible par le type `sigS`, décrit par la définition suivante :

```
Inductive sigS (A:Set)(P:A→Set) : Set :=
    existS : ∀x:A, P x → sigS A P.
Implicit Arguments sigS [A].
```

L'expression “ `sigS (fun x:A ⇒ B)` ” s'écrit aussi `{x :A & B}`. Par rapport au type `sig`, le type `sigS` va permettre de simplifier l'écriture lorsque l'on veut mentionner plusieurs données vérifiant une certaine propriété. Ainsi, le type de la fonction de division donnée dans la section 10.1.1 peut s'écrire plus lisiblement de la façon suivante :

```
∀a b:Z, 0 ≤ b → {q:Z &{r:Z | a=q*b + r ∧ 0 ≤ r < b}}
```

10.1.3 Somme disjointe certifiée

De même que le type `sig` est le correspondant dans la sorte `Set` du type `ex`, le type `sumbool` est le correspondant du type `or`. Ce type est décrit par la définition inductive suivante :

```
Inductive sumbool (A B:Prop) : Set :=
  left : A → sumbool A B | right : B → sumbool A B.
```

Pour ce type inductif, le système *Coq* fournit une notation syntaxique : l'expression `(sumbool A B)` s'écrit `{A}+{B}`. Ce type inductif est particulièrement adapté pour décrire les fonctions de test, qui retourneraient une valeur booléenne en programmation conventionnelle. Par exemple, les spécifications suivantes :

Z_le_gt_dec : $\forall\, x\ y{:}Z,\ \{x \leq y\}{+}\{x > y\}$
Z_lt_ge_dec : $\forall\, x\ y{:}Z,\ \{x < y\}{+}\{x \geq y\}$

sont bien plus informatives que le simple type `Z→Z→bool`, qui ne précise pas sous quelle condition tel ou tel booléen est retourné (pour un exemple d'utilisation, voir page 30 et la section 10.4). C'est une tradition dans les bibliothèques de *Coq* de nommer les fonctions retournant un type `sumbool` avec un nom portant le suffixe « `_dec` » pour exprimer que ces fonction permettent de décider entre deux cas. De telles fonctions permettent de décider entre deux cas.

L'exemple suivant montre une application du type `sumbool` ; nous voulons construire une fonction `div2_gen` qui retourne la moitié par défaut de son argument, à l'aide d'une fonction `div2_of_even` qui ne peut être utilisée que pour les nombres pairs :

```
div2_of_even : ∀n:nat, even n → {p:nat | n = p+p}
```

Supposons maintenant que nous disposions d'une fonction qui effectue le test de parité avec le type suivant :

```
test_even : ∀n:nat, {even n}+{even (pred n)}
```

On pourra alors construire une fonction de division par 2 qui accepte tous les nombres naturels grâce à une construction de filtrage simple ; on notera que le type de cette fonction utilise le type de somme disjointe introduit en section 7.4.4 page 209 et ses constructeurs `inl` et `inr`.

```
Definition div2_gen (n:nat) :
  {p:nat | n = p+p}+{p:nat | pred n = p+p} :=
  match test_even n with
  | left h ⇒ inl _ (div2_of_even n h)
  | right h' ⇒ inr _ (div2_of_even (pred n) h')
  end.
```

En pratique, l'utilisation de `sumbool` permet d'attacher à la fonction un commentaire exprimant le sens de cette fonction, avec l'avantage que ce commentaire peut être vérifié par le système de démonstration. La fonction `div2_of_even` peut seulement être appelée par `div2_gen` parce que nous avons prouvé que son argument numérique satisfait toujours les bonnes conditions.

Le résultat de la fonction `div2_gen` est construit avec le type `sum` pour les sommes disjointes parce que la valeur retournée n'est pas seulement une indication booléenne, mais un nombre entier qui doit satisfaire l'une de deux propriétés différentes. Ceci explique que le résultat soit une somme disjointe avec des types « sous-ensembles » des deux cotés.

Plus loin (section 10.2.7) la tactique `refine` nous donnera un moyen de nous faire aider dans la construction de telles définitions.

Types a égalité décidable

Dans les langages de programmation, il est possible de tester si deux valeurs d'un certain type sont égales (avec certaines restrictions s'il s'agit d'un type fonctionnel). Le pendant en *Coq* de ce test d'égalité s'exprime à l'aide de la spécification suivante :

```
Definition eq_dec (A:Type) := ∀x y:A, {x = y}+{x ≠ y}.
```

Exercice 10.4 Prouver que l'égalité sur `nat` est décidable (c'est à dire construire un terme de type “ `eq_dec nat` ”). En revanche, il est bien connu que l'égalité sur un type fonctionnel est en général indécidable, donc inutile de chercher à construire un terme de type “ `eq_dec (nat→nat)` ”.

Exercice 10.5 Utiliser la fonction demandée dans l'exercice 10.4 pour écrire une fonction comptant le nombre d'occurrences d'un entier naturel dans une liste de type “ `list nat` ”.

10.1.4 Somme disjointe hybride

Pour la description de fonctions partielles, on pourra être amené à utiliser un type intermédiaire entre `sumbool` et `sig`, le type `sumor`. Ce type effectue la somme disjointe entre un type de données et une proposition. Il est décrit par la définition inductive suivante :

```
Inductive sumor (A:Set)(B:Prop) : Set :=
  inleft : A → sumor A B | inright : B → sumor A B.
```

Le système *Coq* fournit aussi une notation pour ce type inductif : l'expression “ `sumor A B` ” s´écrit “ `A + {B}` ”. Ce type peut-être utilisé pour une fonction qui retourne ou bien une valeur dans un ensemble, ou bien une preuve qu'une certaine propriété est satisfaite. Ainsi, le type de la fonction de division euclidienne peut s'écrire également :

```
∀a b:Z,  {q:Z & {r:Z | a = q*b + r ∧ 0 ≤ r < b}}+{b ≤ 0}.
```

Il est possible de combiner la somme disjointe hybride avec la somme disjointe certifiée, en instanciant le paramètre `A` de `sumor` avec un type obtenu par `sumbool`. Les deux connecteurs utilisent le signe '+' de manière infixe, mais l'analyse syntaxique se fait naturellement comme si l'opération `+` était « associative à gauche ». Ainsi, lorsque P_1, P_2 et P_3 sont des propositions, l'expression " {P_1}+{P_2}+{P_3} " est analysée syntaxiquement comme l'expression " ({P_1}+{P_2})+{P_3} " c'est à-dire " (sumor (sumbool P_1 P_2) P_3) ".

10.2 Spécifications fortes

Ajouter des arguments de preuve aux fonctions permet de rendre leur type plus informatifs vis-à-vis de leur comportement. En revanche, construire des fonctions pour ces types est également plus complexe et nous avons souvent besoin d'utiliser des techniques de démonstration pour définir ces fonctions.

10.2.1 Fonctions bien spécifiées

Quand on observe le type de `pred_option` donné en section 7.4.2, on sait qu'elle retourne parfois un nombre naturel, mais on ne sait rien sur ce nombre. Cette fonction n'est pas bien spécifiée par son type. Une autre variante utilisant le type `sumor` peut avoir le type suivant :

```
∀n:nat, {p:nat | n = S p}+{n = 0}
```

Construire une fonction de ce type est plus difficile que pour une fonction faiblement spécifiée, puisque nous allons maintenant devoir également construire des démonstrations. Deux méthodes principales s'offrent à nous. La première est de construire directement un terme du Calcul des Constructions qui combine à la fois les aspects algorithmiques et les aspects logiques de la fonction. La seconde méthode est d'utiliser des tactiques pour construire le terme du Calcul des Constructions cherché, comme s'il s'agissait d'une preuve. Pour illustrer la première méthode, nous donnons ici une construction directe pour la fonction prédécesseur. En général, nous conseillons de privilégier la seconde méthode, que nous détaillons dans la section suivante.

```
Definition pred' (n:nat) : {p:nat | n = S p}+{n = 0} :=
  match n return {p:nat | n = S p}+{n = 0} with
  | O ⇒ inright _ (refl_equal 0)
  | S p ⇒
     inleft _
       (exist (fun p':nat ⇒ S p = S p') p (refl_equal (S p)))
  end.
```

Cette fonction fait intervenir une construction de filtrage dépendant dans le même style que les constructions fabriquées par la tactique `case` (voir section 7.2.1), puisque le type de la valeur retournée dans chaque cas est différent. En effet, les types respectifs des deux branches sont :

```
{p:nat | 0 = S p }+{0 = 0}

{p':nat | S p = S p'}+{S p = 0}
```

10.2.2 Construction de fonctions par preuves

La fonction `pred'` requiert des preuves pour satisfaire la spécification. Il est possible de bénéficier de l'aide des tactiques pour ce travail. La fonction `pred'` peut donc être redéfinie de la façon suivante :

```
Reset pred'.
Definition pred' : ∀n:nat, {p:nat | n = S p}+{n = 0}.
 intros n; case n.
 right; apply refl_equal.
 intros p; left; exists p; reflexivity.
Defined.
```

Ainsi la complexité de la définition de fonctions bien spécifiées peut être fortement réduite à l'aide des tactiques et ce travail est fait interactivement avec l'assistance du système *Coq*.

Dans la section 4.2.2, nous avons montré la correspondance entre certaines tactiques et certaines structures du Calcul des Constructions ; par exemple, la tactique `intro` construit une abstraction, tandis que la tactique `apply` construit une application de fonction. Ici nous observons que la tactique `case` permet de construire une expression de filtrage, ce que nous avions déjà observé dans la section 7.2.1. L'utilisation de tactiques au lieu de termes du calcul des constructions facilite la description des fonctions mais rend le contenu algorithmique d'une fonction difficile à déchiffrer. Nous décrirons en section 10.2.7 une tactique appelée `refine` qui permet de retrouver une description mieux structurée du terme construit.

Lorsque l'on utilise le mécanisme de preuve par tactiques pour construire une fonction, il faut être très circonspect dans l'utilisation de tactiques automatiques. En effet, ces tactiques ne laissent à l'utilisateur qu'un faible contrôle sur les termes effectivement construits. En particulier, nous conseillons d'éviter les tactiques automatiques lorsque le terme à construire a encore un contenu algorithmique important, surtout si la fonction à construire n'est que faiblement spécifiée : on n'est pas sûr d'obtenir la fonction prévue (voir la discussion page 86). Même pour une fonction assez fortement spécifiée, les choix effectués par les tactiques automatiques peuvent mener à des choix algorithmiques désastreux. Un exemple intéressant à cet égard est présenté dans le chapitre 12.

10.2.3 Fonctions partielles par précondition

Une fonction partielle de A vers B peut se spécifier par un type de la forme “ $\forall$`x:`A`,` $(P\ x) \rightarrow B$ ” où P est un prédicat qui décrit le domaine de définition de la fonction. L'application de cette fonction demande deux arguments : un terme t de type A, et une preuve de la *précondition* “ $P\ t$ ”.

Dans la construction de cette fonction on peut être amené à construire un terme de type B dans un contexte où l'on peut construire une preuve de “ $\sim P\ t$ ”. Un terme utilisant `False_rec` permet alors de résoudre ce problème. Il s'agit d'une application de l'élimination de la contradiction. Ainsi, pour la fonction prédécesseur, on pourra écrire la définition suivante (notons que cette fonction est encore faiblement spécifiée) :

```
Definition pred_partial : ∀n:nat, n ≠ 0 → nat.
 intros n; case n.
  ...
  ============================
   0 ≠ 0 → nat

intros h; elim h; reflexivity.
  ...
  ============================
   ∀ n0:nat, S n0 ≠ 0 → nat

 intros p h'; exact p.
Defined.
```

10.2.4 ** Complexité des démonstrations de préconditions

Lorsque l'on cherche à composer des fonctions avec préconditions, il faut bien sûr vérifier leurs conditions d'utilisation. Par exemple, pour composer avec elle-même la fonction `pred_partial`, il faut trouver un nouvel ensemble de définition et montrer qu'il suffit pour les différentes applications. Ici le domaine de définition est l'ensemble des `n` tels que $2 < n$. Il est nécessaire de démontrer que cet ensemble est inclus dans le domaine de définition de `pred_partial`. Nous effectuons cette démonstration dans un lemme auxiliaire.

```
Theorem le_2_n_not_zero : ∀n:nat, 2 ≤ n → n ≠ 0.
Proof.
 intros n Hle; elim Hle; intros; discriminate.
Qed.
```

La seconde démonstration porte sur l'image d'un nombre par la fonction `pred_partial`. Il est intéressant d'observer cette seconde démonstration, car elle fait apparaître une difficulté inattendue.

```
Theorem le_2_n_pred :
 ∀(n:nat)(h: 2 ≤ n), pred_partial n (le_2_n_not_zero n h) ≠ 0.
```

Le premier réflexe est d'effectuer une preuve par récurrence sur le prédicat `le` comme pour le théorème précédent. Mais ceci provoque un message d'erreur :

```
intros n h; elim h.
```
Error: Cannot solve a second-order unification problem

La raison de cet échec est que l'hypothèse `h` est utilisée comme argument dans un théorème qui parle de `n`, on ne peut donc pas remplacer toutes les instances de `n` par d'autres valeurs sans modifier également l'hypothèse `h`. Une façon de simplifier notre démonstration est d'appliquer `pred_partial` à `n` et une preuve arbitraire que `n` est non nul. Notre théorème est alors meilleur et plus facile à prouver.

```
Abort.
Theorem le_2_n_pred' :
 ∀n:nat, 2 ≤ n → ∀h:n ≠ 0, pred_partial n h ≠ 0.
Proof.
 intros n Hle; elim Hle.
 intros; discriminate.
 simpl; intros; apply le_2_n_not_zero; assumption.
Qed.

Theorem le_2_n_pred :
 ∀(n:nat)(h:2 ≤ n), pred_partial n (le_2_n_not_zero n h) ≠ 0.
 Proof.
 intros n h; exact (le_2_n_pred' n h (le_2_n_not_zero n h)).
Qed.
```

Avec les théorèmes `le_2_n_not_zero` et `le_2_n_pred`, on peut construire la fonction qui répète deux fois le calcul du prédécesseur :

```
Definition pred_partial_2 (n:nat)(h:2 ≤ n) : nat :=
  pred_partial (pred_partial n (le_2_n_not_zero n h))
               (le_2_n_pred n h).
```

Le problème que nous avons dû résoudre vient de ce que le type retourné par la fonction `pred_partial` n'est pas assez fortement spécifié : en effet nous savons seulement que la valeur retournée est un nombre naturel, mais nous n'avons pas assez d'information pour savoir que cette valeur est assez bonne pour être donnée en argument dans un second appel. L'exercice 15.4 page 433 décrit une autre technique pour démontrer le théorème `le_2_n_pred`.

10.2.5 ** Renforcement des spécifications

Un type dépendant plus expressif pour la fonction `pred` permet d'éviter ces difficultés.

Nous pouvons définir une fonction prédécesseur `pred_strong` dont le type est « `∀n:nat, n ≠ 0 → {v: nat | n = S v}` », puis utiliser `pred_strong` pour construire la fonction dont le type est le suivant :

```
∀n:nat, 2 ≤ n → {v:nat | n = S (S v)}
```

La définition de `pred_strong` est assez facile, en particulier en utilisant des tactiques :

```
Definition pred_strong : ∀n:nat, n ≠ 0 → {v:nat | n = S v}.
 intros n; case n;
  [intros H; elim H | intros p H'; exists p]; trivial.
Defined.
```

L'un des avantages de la spécification de `pred_strong` est que nous allons pouvoir raisonner sur une valeur `p` représentant le résultat de la fonction indépendamment de la précondition utilisée pour la calculer.

Il n'est donc plus nécessaire que la preuve de “ `2 ≤ n` ” soit passée en argument à une autre fonction dans le théorème qui justifie que la précondition du deuxième appel de la fonction prédécesseur est satisfaite. La preuve qui suit utilise la tactique `omega` pour raisonner sur les égalités et inégalités (voir section 8.3.2).

```
Theorem pred_strong2_th1 :
 ∀n p:nat, 2 ≤ n → n = S p → p ≠ 0.
Proof.
 intros; omega.
Qed.

Theorem pred_th1 :
 ∀n p q:nat, n = S p → p = S q → n = S (S q).
Proof.
 intros; subst n; auto.
Qed.
```

Nous disposons alors des briques pour construire notre fonction dans laquelle `pred_strong` est itérée deux fois.

```
Definition pred_strong2 (n:nat)(h:2≤n):{v:nat | n = S (S v)} :=
  match pred_strong n (le_2_n_not_zero n h) with
  | exist p h' ⇒
      match pred_strong p (pred_strong2_th1 n p h h') with
      | exist p' h'' ⇒
          exist (fun x:nat ⇒ n = S (S x))
                p' (pred_th1 n p p' h' h'')
      end
  end.
```

Ici encore, la fonction `pred_strong2` aurait pu être construite comme une preuve à l'aide de tactiques. Voici le script qui permet d'obtenir ce résultat :

```
Definition pred_strong2' :
```

```
  ∀n:nat, 2 ≤ n → {v:nat | n = S (S v)}.
 intros n h; case (pred_strong n).
 apply le_2_n_not_zero; assumption.
 intros p h'; case (pred_strong p).
 apply (pred_strong2_th1 n); assumption.
 intros p' h''; exists p'.
 eapply pred_th1; eauto.
Defined.
```

Notons bien que nous avons utilisé des tactiques automatiques à trois reprises, deux fois avec la tactique `assumption` et une fois avec les tactiques `eapply` et `eauto`. Ces utilisations n'ont pas d'impact sur le contenu algorithmique de la fonction, puisqu'il s'agissait à chaque fois de vérifier une propriété des données, et non de déterminer un calcul effectif.

10.2.6 *** Renforcement minimal de spécification

Les fonctions faiblement spécifiées sont habituellement accompagnées par des lemmes qui expriment les propriétés satisfaites par ces fonctions. L'utilisation de ces fonctions faiblement spécifiées et de leurs théorèmes compagnons en combinaison avec des fonction avec précondition est souvent difficile car elle requiert de construire des termes comportants du filtrage dépendant et des égalités. Si f est une fonction faiblement spécifiée dont le type final est un type inductif à plusieurs constructeurs $c_1, \ldots, c_l$. Les théorèmes compagnons pour f on naturellement la forme

$$\forall x\texttt{:}A\texttt{,}\ \texttt{(}f\ x\ \texttt{=}\ c_i\texttt{)}\ \to\ P_i\ x.$$

On pourrait penser que ces théorèmes pallient la faiblesse de la spécification de f ; l'exemple suivant nous montre que ce n'est pas toujours le cas, surtout quend il s'agit de composer f avec des fonctions partielles spécifiées par préconditions.

Afin d'illustrer ces problèmes, et de présenter une méthode pour les résoudre, nous travaillons sur un exemple simple. Reprenons les prédicats `prime` et `divides`, introduits en 5.1.1, page 103.

Considérons un test de primalité faiblement spécifié, c'est à dire une fonction de type `nat→bool`. Deux hypothèses permettent de spécifier la valeur retournée par ce test (faute de quoi `prime_test` resterait une fonction à valeur booléenne quelconque). Le contexte fournit également une fonction qui trouve un diviseur premier de son argument, mais ne peut être appelée que pour les nombres non premiers. Ici encore cette fonction est faiblement spécifiée et accompagnée d'une hypothèse :

```
Section minimal_specification_strengthening.

 Variable prime : nat→Prop.
 Definition divides (n p:nat) : Prop := ∃q:_ | q*p = n.
 Definition prime_divisor (n p:nat):= prime p ∧ divides p n.
```

```
Variable prime_test : nat→bool.
Hypotheses
  (prime_test_t : ∀n:nat, prime_test n = true → prime n)
  (prime_test_f : ∀n:nat, prime_test n = false → ~prime n).

Variable get_primediv_weak : ∀n:nat, ~prime n → nat.
Hypothesis get_primediv_weak_ok :
    ∀(n:nat)(H:~prime n), 1 < n →
       prime_divisor n (get_primediv_weak n H).


Theorem divides_refl : ∀n:nat, divides n n.
 Proof.
 intro n; exists 1; simpl; auto.
Qed.
Hint Resolve divides_refl.
```

Nous souhaitons composer le test `prime_test` et la fonction auxiliaire `get_primediv_weak` afin d'obtenir une fonction donnant un diviseur premier de tout nombre supérieur ou égal à 2.

Il est tentant de construire une expression conditionnelle de la forme suivante :

`fun n:nat ⇒ if prime_test n then n else` E

L'expression E ne peut être un appel de la fonction `get_primediv_weak`, car cette fonction prend entre autres arguments une preuve de “ `~prime n` ”, qui n'est pas fournie par le contexte courant.

Une solution naïve, mais erronée, consiste à utiliser la tactique `caseEq` que nous avons décrite en section 7.2.7 page 185 pour construire le terme cherché en assurant qu'un contexte différent est fourni pour chaque branche de l'expression conditionnelle. Voici un exemple de script qui permet de définir la fonction attendue :

```
Definition bad_get_prime : nat→nat.
  intro n; caseEq (prime_test n).
 ...
 ============================
  prime_test n = true → nat

subgoal 2 is:
 prime_test n = false → nat

 intro; exact n.
 intro Hfalse; apply (get_primediv_weak n); auto.
Defined.
```

```
Print bad_get_prime.
bad_get_prime =
fun n:nat ⇒
  (if prime_test n as b return (prime_test n = b → nat)
   then fun _:prime_test n = true ⇒ n
   else
    fun Hfalse:prime_test n = false ⇒
      get_primediv_weak n (prime_test_f n Hfalse))
   (refl_equal (prime_test n))
    : nat→nat
Argument scope is [nat_scope]
```

Cette technique fait donc apparaître une construction de filtrage dépendant appliquée à une égalité dont les altérations dans chaque cas permettront de conclure[3]. Cette technique n'est qu'apparemment productive parce qu'il est étonnament difficile de faire des démonstrations sur notre nouvelle fonction. Supposons en effet que nous voulions prouver que `bad_get_primediv` renvoie toujours un diviseur premier de son argument.

```
Theorem bad_get_primediv_ok :
 ∀n:nat, 1 < n → prime_divisor n (bad_get_prime n).
Proof.
 intros n H; unfold bad_get_prime.
...
n : nat
H : 1 < n
============================
  prime_divisor n
   ((if prime_test n as b return (prime_test n = b → nat)
     then fun _:prime_test n = true ⇒ n
     else
      fun Hfalse:prime_test n = false ⇒
        get_primediv_weak n (prime_test_f n Hfalse))
     (refl_equal (prime_test n)))
```

À ce point du raisonnement il est naturel d'étudier séparément les exécutions correspondant aux deux valeurs possibles de « `prime_test n` ». Ceci devrait se faire à l'aide de la tactique `case` mais elle ne peut pas être utilisée ici :

```
case (prime_test n).
Error: Cannot solve a second-order unification problem
```

La difficulté vient de ce que le terme « `refl_equal (prime_test n)` » doit à la fois avoir le type « `prime_test n = prime_test n` » (qu'il a naturellement) et le type « `prime_test n = true` » ou « `prime_test n = false` » selon le cas. Résoudre cette difficulté est très ardu.

3. Dans notre exemple, seul le cas `false` est vraiment utilisé, mais le lecteur n'aura aucune peine à généraliser cet exemple.

La solution que nous proposons pour ce type de problème est d'utiliser une fonction légèrement plus spécifiée que `prime_test`, mais dont la spécification utilise `prime_test`. L'effet obtenu est le même qu'avec la tactique `caseEq` : on peut définir le comportement de la fonction dans chaque cas en utilisant la même égalité " `prime_test n = true` " ou " `prime_test n = false` " dans le contexte, mais la fonction obtenue se prête mieux à un raisonnement sur son comportement.

Cette fonction se spécifie à l'aide de `sumbool` et se définit autour d'une traitement par cas très simple :

```
Definition stronger_prime_test :
  ∀n:nat, {(prime_test n)=true}+{(prime_test n)=false}.
 intro n; case (prime_test n);[left | right]; reflexivity.
Defined.
```

Puis la fonction cherchée est construite par un filtrage non dépendant sur le résultat de `stronger_prime_test` au lieu d'un filtrage dépendant sur le résultat de `prime_test`.

```
Definition get_prime (n:nat) : nat :=
  match stronger_prime_test n with
  | left H ⇒ n
  | right H ⇒ get_primediv_weak n (prime_test_f n H)
  end.
```

Démontrer la propriété attendue pour notre fonction est maintenant très simple [4] :

```
 Theorem get_primediv_ok :
  ∀n:nat, 1 < n → prime_divisor n (get_prime n).
 Proof.
  intros n H; unfold get_prime.
  case (stronger_prime_test n); auto.
  split; auto.
 Qed.

End minimal_specification_strengthening.
```

Nous appelons cette technique *renforcement minimal de spécification* parce que la spécification de la fonction `stronger_prime_test` est seulement que la valeur retournée est la même que pour `prime_test` mais pas une indication de la propriété assurée par `prime_test`. Le lecteur est invité à tester cette technique sur d'autres exemples de types inductifs.

4. En supposant que la base de théorèmes pour `auto` contient la reflexivité de `divides`.

10.2.7 La tactique `refine`

La tactique `refine` permet de construire les fonctions par preuve tout en conservant un bon contrôle sur la structure de la fonction et de son contenu algorithmique. Le principe utilisé par cette tactique est de laisser l'utilisateur fournir un terme du Calcul des Constructions dont certains fragments sont laissés inconnus. Ces fragments sont alors produits par la tactique comme de nouveaux buts à résoudre.

Ainsi, on pourra comparer la définition suivante à la définition de la fonction `pred_partial` donnée en section 10.2.3 :

```
Definition pred_partial' : ∀n:nat, n ≠ 0 → nat.
refine
 (fun n ⇒
    match n as x return x ≠ 0 → nat with
    | O ⇒ fun h:0 ≠ 0 ⇒ _
    | S p ⇒ fun h:S p ≠ 0 ⇒ p
    end).
```

Une partie de cette fonction est encore inconnue et est représentée par un joker '`_`'. Le système *Coq* retourne un but correspondant au type attendu à cet endroit dans la fonction :

```
...
 n : nat
 h : 0≠0
 ============================
  nat
```

Pour définir la fonction `pred_partial_2` nous pouvons de nouveau utiliser la tactique `refine`. La première tentative, trop directe, échoue :

```
Definition pred_partial_2' : ∀n:nat, le 2 n → nat.

refine (fun n h ⇒ pred_partial (pred_partial n _) _).
Error: generated subgoal (?268::nat) ≠ 0 has metavariables in it
```

Le type attendu pour le second joker dépend de la valeur représentée par le premier joker. Pour éviter cela, nous pouvons construire notre démonstration de manière nommer le premier joker (appelons-le `h'`), en nous reposant sur l'application d'une abstraction pour fournir ce nom. Le terme construit est très proche de celui construit par la tactique `cut` et correspond en fait au théorème plus général `le_2_n_pred'` que nous avons prouvé avant de démontrer `le_2_n_pred` (voir page 290).

```
refine
 (fun n h ⇒
```

```
   (fun h':n≠0 ⇒ pred_partial (pred_partial n h') _)
   _).
```

Cette commande laisse bien sûr deux buts à démontrer, mais elle permet de condenser en une tactique l'ensemble du contenu calculatoire de notre fonction : nous voyons immédiatement que la fonction `pred_partial` est composée avec elle-même pour obtenir le résultat final.

Pour définir la fonction `pred_strong2` à l'aide de la tactique `refine` nous pouvons construire l'expression suivante :

```
Definition pred_strong2'' :
   ∀n:nat, 2≤n → {v:nat | n = S (S v)}.
refine
 (fun n h ⇒
    match pred_strong n _ with
    | exist p h' ⇒
      match pred_strong p _ with
      | exist p' h'' ⇒ exist _ p' _
      end
    end).
```

L'expression passée à `refine` contient cinq points d'interrogation correspondant à cinq lacunes, mais deux de ces jokers correspondent aux paramètres du constructeur `exist` et peuvent être déduits du contexte. Il ne reste alors que trois buts, qui sont prouvables avec les théorèmes que nous avons déjà utilisés dans les autres présentations de la fonction `pred_strong2`.

Les expressions fournies à `refine` peuvent également contenir la construction `fix` utilisée pour construire une fonction récursive anonyme (voir section 7.3.7). Nous donnerons un exemple de l'utilisation conjointe de `refine` et `fix` dans la section 10.4.2.

Toutes les fonctions que l'on peut construire en donnant directement le terme du Calcul des Constructions peuvent donc être définies également à l'aide de la tactique `refine`, souvent plus simplement et avec un meilleur soutien de la part du système *Coq*. Par exemple, on peut mettre au point l'expression à donner à la tactique `refine` progressivement, en laissant de grosses lacunes que l'on remplit en utilisant les buts engendrés par la tactique comme indication du type attendu pour chaque lacune. C'est le même procédé que lorsque l'on construit une preuve pas-à-pas avec les tactiques de bases, mais le script de preuve obtenu est mieux adapté à une relecture ultérieure car la structure est plus apparente.

10.3 Variations sur la récursion structurelle

Nous avons déjà vu dans le chapitre 7 comment définir des fonctions récursives. Ici nous allons nous attarder sur les techniques pour prouver des propriétés sur ces fonctions.

10.3.1 Fonctions récursives structurelles à pas multiple

En mathématiques on trouve des suites récurrentes simples et des suites récurrentes doubles, comme la suite de Fibonacci. La récurrence double n'est qu'un cas particulier et l'on pourra rencontrer des suites récurrentes triples, quadruples, *etc.* Cette forme de multiplicité est assez difficile à traiter dans les preuves par récurrence et mérite donc qu'on s'y attarde. Nous appuierons nos descriptions sur l'exemple de la fonction de division par 2 qui peut être définie de la façon suivante :

```
Fixpoint div2 (n:nat) : nat :=
  match n with 0 ⇒ 0 | 1 ⇒ 0 | S (S p) ⇒ S (div2 p) end.
```

Pour montrer la difficulté d'établir les propriétés de cette fonction, nous allons nous intéresser à la démonstration du théorème suivant :

```
Theorem div2_le : ∀n:nat, div2 n ≤ n.
Proof.
```

Un essai manqué

La fonction `div2` est définie comme une fonction récursive sur son seul argument. Nous tentons directement une démonstration par récurrence, et traitons immédiatement le cas de base (0) :

```
Proof.
 induction n.
 ...
 ============================
  div2 0 ≤ 0
 simpl; auto.
 ...
 n : nat
 IHn : div2 n ≤ n
============================
div2 (S n) ≤ S n
```

Prouver le deuxième but doit permettre de couvrir la démonstration pour tous les entiers supérieurs ou égaux à 1, mais le nombre 1 apparaît encore comme un cas particulier, ceci donne l'idée de faire une deuxième preuve par récurrence sur `n`.

```
 induction n.
 ...
 IHn : div2 0 ≤ 0
 ============================
  div2 1 ≤ 1
```

Le premier but de cette deuxième preuve par récurrence est encore un but simple.

```
simpl.
...
IHn : div2 0 ≤ 0
============================
 0 ≤ 1
auto.
```

Mais le suivant est beaucoup plus complexe :

```
...
n : nat
IHn0 : div2 n ≤ n → div2 (S n) ≤ S n
IHn : div2 (S n) ≤ S n
============================
 div2 (S (S n)) ≤ S (S n)
```

On s'aperçoit d'abord que l'hypothèse `IHn0` est inutile puisqu'elle est une conséquence logique directe de l'hypothèse `IHn`. En outre, le but à prouver est une propriété de “ `div2 (S (S n1))` ”, c'est-à-dire une propriété de “ `div2 n` ”, mais `IHn1` n'exprime qu'une propriété de “ `div2 (S n)` ”.

Cette incohérence entre le but à prouver et les hypothèses fournies par le schéma de preuve par récurrence montre que cet essai de preuve n'est pas parti dans une bonne direction.

Un essai transformé

Pour démontrer la propriété voulue, nous devons réfléchir à la structure de la fonction `div2`. Les arguments de `div2` pour lesquels le calcul se fait en exactement p appels récursifs sont $2p$ et $2p+1$. Ceci suggère de démontrer la propriété simultanément pour un nombre et son successeur. Reprenons donc notre preuve en l'organisant sur ce principe :

```
Theorem div2_le' : ∀n:nat, div2 n ≤ n.
 Proof.
 intros n.
```

La première étape est de faire apparaître la propriété que nous voulons effectivement prouver.

```
cut (div2 n ≤ n ∧ div2 (S n) ≤ S n).
```

Nous avons alors deux buts, dont le premier se résout immédiatement :

```
...
============================
 div2 n ≤ n ∧ div2 (S n) ≤ S n → div2 n ≤ n
tauto.
```

Le second but est simplement la proposition que nous avons introduite à l'aide de la tactique `cut`. Nous effectuons notre preuve par une récurrence simple sur `n` :

```
elim n.
...
n : nat
============================
  div2 0 ≤ 0 ∧ div2 1 ≤ 1
```

La preuve de cette conjonction est aisée, par exemple à l'aide de la commande suivante (la commande `auto` profite des théorèmes déjà présents dans la mémoire du système de preuve).

```
simpl; auto.
```

Le second but prend une forme légèrement plus compliquée : l'hypothèse de récurrence et le but à prouver sont tous deux des conjonctions.

```
...
n : nat
============================
∀ n0:nat,
 div2 n0 ≤ n0 ∧ div2 (S n0) ≤ S n0 →
 div2 (S n0) ≤ S n0 ∧ div2 (S (S n0)) ≤ S (S n0)
```

```
intros p [H1 H2].
...
n : nat
p : nat
H1 : div2 p ≤ p
H2 : div2 (S p) ≤ S p
============================
 div2 (S p) ≤ S p ∧ div2 (S (S p)) ≤ S (S p)
```

On observe rapidement que le premier membre de la conjonction dans le but est le même que l'hypothèse `H2`. On peut donc aisément ce débarrasser de cette partie de la preuve, par exemple avec les commandes suivantes :

```
  split; auto.
```

Le but a alors la bonne forme pour que notre démonstration réussisse : le fait `H1` fournit une hypothèse de récurrence pour `p` alors que nous devons démontrer la propriété pour “ `S (S p)` ”. Nous pouvons maintenant procéder de la manière suivante :

```
 simpl; auto with arith.
Qed.
```

Il est important de noter une caractéristique particulière de cette démonstration : pour démontrer une propriété particulière par récurrence, on démontre en fait une propriété plus forte (ici la conjonction de la propriété pour un nombre et son successeur). Cette situation se retrouve très fréquemment dans les preuves par récurrence et doit être utilisée comme ligne guide pour organiser une tentative de démonstration : lorsqu'une démonstration par récurrence échoue, il est souvent judicieux de prouver un énoncé plus fort par récurrence et d'en déduire le résultat recherché, une technique qui a été automatisée avec succès dans le système *Nqthm* [18].

Nous retrouvons ici la correspondance de Curry-Howard : l'équivalent en programmation fonctionnelle de cette pratique de démonstration est la technique consistant à programmer une fonction non par une récursion directe, mais à l'aide d'une fonction auxiliaire plus générale (avec plus d'arguments). Cette fonction auxiliaire se programme alors par récursion. L'appel de la fonction principale à la fonction auxiliaire correspond au premier but engendré par l'appel à `cut`.

Exercice 10.6 Définir la fonction `div3` qui calcule le quotient de la division d'un nombre par 3. Démontrer le théorème similaire à `div2_le` qui exprime que le quotient de la division par 3 d'un nombre est toujours inférieur à ce nombre.

Exercice 10.7 Définir la fonction `mod2` qui calcule le reste de la division d'un nombre par 2. Montrer le théorème suivant :

$$\forall n : nat,\ n = 2 \times \texttt{div2}(n) + \texttt{mod2}(n)$$

Exercice 10.8 *

Définir la fonction de Fibonacci avec un pas de récurrence double. Rappelons que la suite de Fibonacci est la suite u_n définie par :

$$u_0 = 1 \qquad u_1 = 1 \qquad u_{n+2} = u_n + u_{n+1}$$

L'équivalent en *OCAML* de la fonction obtenue devrait avoir une complexité exponentielle (il faut approximativement 2^n additions pour calculer u_n). Définir ensuite une fonction qui calcule simultanément u_n et u_{n+1} avec une complexité linéaire (il faut approximativement n additions pour calculer u_n et u_{n+1}). Démontrer que les deux fonctions retournent la même valeur pour tout n. Cet exercice peut être continué avec les exercices 10.10 page 302, 10.15 page 307, 10.17 page 317 et 16.8 page 460.

Récurrence double

La preuve précédente est un cas particulier de récurrence par pas de deux. Il intéressant d'en prouver une version générique utilisable dans d'autres circonstances similaires.

```
Theorem nat_2_ind :
 ∀P:nat→Prop, P 0 → P 1 →(∀n:nat, P n → P (S (S n)))→
   ∀n:nat, P n.
```

La démonstration de ce nouveau principe de récurrence se fait aisément, comme dans la section précédente, en introduisant une conjonction que l'on prouve par récurrence.

```
Proof.
 intros P H0 H1 Hrec n; cut (P n ∧ P (S n)).
 tauto.
 elim n; intuition.
Qed.
```

Ce principe de récurrence s'utilise à l'aide de la variante « `elim ...using` ».

Il est possible d'automatiser la construction de principes de récurrence spécialement adaptés pour les fonctions récursives structurelles. Ce travail a été effectué en particulier par P. Courtieu et décrit dans l'article qu'il a publié avec G. Barthe [9]. Le produit de ce travail est une tactique `functional induction`, disponible dans les versions très récentes de *Coq* [5].

Exercice 10.9 * Démontrer des principes de récurrence triple et quadruple.

Exercice 10.10 ** Le principe `nat_2_ind` n'est pas adapté pour raisonner sur la fonction de Fibonacci (voir exercice 10.8 page 301). Construire et démontrer le principe de récurrence pour cette fonction. Utiliser ce principe pour démontrer la propriété suivante :

$$\forall\, n.\ u_{n+p+2} = u_{n+1}u_{p+1} + u_n u_p.$$

Exercice 10.11 ** Reprendre les exercices de la section précédente et les refaire en utilisant les principes de récurrence adaptés.

10.3.2 Simplification du pas

La structure de la démonstration du principe de récurrence double, qui utilise en fait une récurrence simple, suggère que l'on peut obtenir une fonction de division par deux qui calcule simultanément la valeur pour `n` et pour son successeur.

```
Fixpoint div2'_aux (n:nat) : nat*nat :=
  match n with
  | 0 ⇒ (0, 0)
  | S p ⇒ let (v1,v2) := div2'_aux p in (v2, S v1)
  end.

Definition div2' (n:nat) : nat := fst (div2'_aux n).
```

5. plus récente que Coq7.4

Exercice 10.12 ** Définir par une récurrence à pas simple la fonction :

```
div2_mod2 :∀n:nat, {q:nat & {r:nat | n = (mult2 q)+r ∧ r ≤ 1}}
```

10.3.3 Fonctions récursives à plusieurs arguments

NEW: Merci Gérard :nombreuses occurrences de "fonction" à remplacer par "algos" ; Notamment les deux premières versions de l'addition devraient s'appeler "algorithme séquentiel gauche" (resp. droit)

La valeur retournée par une fonction récursive peut elle-même être une fonction. De cette manière, on construit naturellement des fonctions récursives à plusieurs arguments. Néanmoins, un seul de ces arguments est utilisé pour contrôler la récursion : c'est l'argument *principal*.

Les arguments supplémentaires d'une fonction récursive structurelle peuvent être donnés avant ou après l'argument principal. Pour certaines fonctions, le choix de l'argument principal est arbitraire, ainsi on peut définir plusieurs fonctions du Calcul des Constructions pour représenter la notion mathématique usuelle d'addition de deux entiers naturels, suivant que l'on veut que la récurrence structurelle soit contrôlée par le premier ou le deuxième argument. Voici par exemple la description de l'algorithme d'addition pour les entiers naturels, lorsque la récurrence est contrôlée par le premier argument (cette définition est celle utilisée dans les bibliothèques de *Coq*) :

```
Fixpoint plus (n m:nat){struct n} : nat :=
  match n with O ⇒ m | S p ⇒ S (plus p m) end.
```

En revanche, voici un autre algorithme qui retourne toujours le même résultat que le précédent (au sens classique des mathématiques, ce serait la même fonction), mais dont la récursion est contrôlée par le deuxième argument.

```
Fixpoint plus' (n m:nat){struct m} : nat :=
  match m with O ⇒ n | S p ⇒ S (plus' n p) end.
```

Pour les fonctions récursives structurelles à plusieurs arguments, il est important de se souvenir du principe énoncé en section 7.3.6.

Les raisonnements sur les fonctions récursives structurelles se font naturellement par récurrence sur l'argument principal de ces fonctions puis en suivant la structure des constructions de filtrage contenues dans ces fonctions.

En premier exemple, nous invitons le lecteur à relire la démonstration du théorème `plus_assoc` donné en section 7.3.6. Dans cette démonstration, nous avons utilisé une preuve par récurrence sur la variable quantifiée `n`, qui est la seule à apparaître en position d'argument principal dans chacune des additions où elle apparaît.

Considérons maintenant la démonstration que l'addition est commutative. Pour la fonction `plus`, cette démonstration se décompose naturellement en les démonstrations des deux lemmes suivants :

plus_n_O : $\forall\, n{:}nat,\ n = n{+}0$
plus_n_Sm : $\forall\, n\ m{:}nat,\ S\ (n{+}m) = n{+}S\ m$

Etudions la démonstration de `plus_n_Sm` (les démonstrations de `plus_n_O` et `plus_n_Sm` font partie des bibliothèques de *Coq*).

```
Theorem plus_n_Sm : ∀n m:nat, S (n+m) = n+(S m).
```

Ici, `n` apparaît comme argument principal pour les deux utilisations de la fonction `plus` dans l'égalité. Nous entamons donc cette preuve par la commande suivante :

```
Proof.
intros n; elim n.
...
============================
  ∀ m:nat, S (0+m) = 0+S m
```

Nous pouvons utiliser la tactique `simpl` sur ce premier but. Les deux membres de l'égalité vont se réduire à la même valeur " `S m` ", ce but devient donc trivial. Le second but a la forme suivante :

```
simpl; trivial.
 ...
 n : nat
 ============================
  ∀ n0:nat,
   (∀ m:nat, S (n0+m) = n0+S m)→
   ∀ m:nat, S (S n0+m) = S n0+S m
```

Ce but se démontre aisément en utilisant l'hypothèse de récurrence pour effectuer une réécriture.

```
 intros n0 Hrec m; simpl; rewrite Hrec; trivial.
Qed.
```

NEW: Merci Gérard : On revient à des plreuves assez élémentaires après les développements terribles de 10.2.4 à 10.2.6 ; on devrait mieux graduer les difficultés

Le théorème `plus_n_O` se prouve aussi aisément. Avec ces deux théorèmes, nous pouvons alors aborder la preuve que `plus` est commutative :

```
Theorem plus_comm : ∀n m:nat, n+m = m+n.
```

Nous ne disposons pas pour cette preuve de critère précis pour choisir la variable sur laquelle sera faite la récurrence. En effet, les deux variables quantifiées universellement du théorème apparaissent une fois en position d'argument principal et une fois en position d'argument secondaire. Effectuons par exemple la preuve par récurrence sur `n` :

```
Proof.
 intros n; elim n.
 ...
 n : nat
 ============================
  ∀ m:nat, 0+m = m+0
```

Pour ce premier but, nous utilisons encore la tactique `simpl`. La conclusion devient exactement l'énoncé de `plus_n_O`. Le second but engendré par la commande de preuve par récurrence devient plus lisible après introduction d'hypothèses et de variables et simplification des appels de fonctions récursives.

```
 exact plus_n_O.
 intros p Hrec m; simpl.
 ...
 n : nat
 p : nat
 Hrec : ∀ m:nat, p+m = m+p
 m : nat
 ============================
   S (p+m) = m + S p
```

On voit apparaître dans ce but une instance du membre droit du théorème `plus_n_Sm` :

```
 rewrite <- plus_n_Sm; auto.
Qed.
```

Le cas de `plus'`

Il est intéressant d'étudier la démonstration similaire que `plus'` est commutative, pour bien souligner l'importance de faire les démonstrations par récurrence sur l'argument principal. Nous commençons par prouver les deux lemmes suivants :

```
plus'_O_n : ∀n:nat, n = plus' 0 n
plus'_Sn_m : ∀n m:nat, S (plus' n m) = plus' (S n) m.
```

En suivant notre principe guide, nous écrivons une preuve par récurrence sur l'argument principal, `n` pour le premier lemme. La tactique suivante suffit pour cette preuve.

```
 intros n; elim n; simpl; auto.
```

Le second lemme se prouve aussi facilement en considérant que `m` est l'argument principal de `plus'` :

```
 intros n m; elim m; simpl; auto.
```

La démonstration du théorème de commutativité se décompose alors de la manière suivante :

```
Theorem plus'_comm : ∀n m:nat, plus' n m = plus' m n.
Proof.
```

Ici encore, le choix de la variable sur laquelle doit avoir lieu la récurrence est arbitraire et l'on peut par exemple effectuer la démonstration par récurrence sur `m`. La séquence de tactiques que nous envoyons est la suivante :

```
 intros n m; elim m; simpl.
 apply plus'_O_n.
 intros p Hrec; rewrite <- plus'_Sn_m; auto.
Qed.
```

Avec l'aide des théorèmes de commutativité, il est assez aisé de démontrer que les deux fonctions `plus` et `plus'` ont la même valeur. En voici la démonstration, écrite en prenant la même variable en position d'argument principal pour les deux fonctions :

```
Theorem plus_plus' : ∀n m:nat, n+m = plus' n m.
Proof.
 intros n m; rewrite plus'_comm; elim n; simpl; auto.
Qed.
```

Addition récursive terminale

Il existe une troisième manière de définir l'addition, apparemment très proche, mais qui met à jour certaines des difficultés fréquentes dans les preuves par récurrence concernant des fonctions récursives à plusieurs arguments :

```
Fixpoint plus'' (n m:nat){struct m} : nat :=
  match m with 0 ⇒ n | S p ⇒ plus'' (S n) p end.
```

Quelle est la différence importante de cette définition par rapport aux deux précédentes ? Cet algorithme présente une *récursion terminale*, c'est à dire que dans la branche qui présente une récursion, il n'y a plus de calcul à faire après le retour de l'appel récursif : la valeur de la fonction est la valeur retournée par cet appel récursif. Les fonctions récursives terminales sont très appréciées des développeurs de compilateurs pour les langages fonctionnels, car elles permettent de construire un code final généralement aussi efficace que les codes produits à partir de programmes impératifs, en évitant la gestion de la pile.

Du point de vue des démonstrations, cette fonction est plus difficile à manipuler, et nous pouvons déjà souligner la raison des difficultés : lors de l'appel récursif, l'argument secondaire n'est pas le même que lors de l'appel initial. L'argument secondaire lors de l'appel récursif est “ `S n` ”, tandis que l'argument secondaire lors de l'appel initial est `n`. Lorsque cette situation se produit, il faut appliquer un principe guide supplémentaire :

Lors des preuves par récurrence sur des fonctions récursives à plusieurs arguments, il peut être nécessaire de s'assurer que les arguments secondaires sont quantifiés universellement au moment où l'on engage la preuve par récurrence.

Afin de montrer l'intérêt de ce principe, nous nous proposons de démontrer le théorème suivant, qui peut servir de lemme pour une preuve de la commutativité de `plus''` :

```
Theorem plus''_Sn_m : ∀n m:nat, S (plus'' n m) = plus'' (S n) m.
Proof.
```

Montrons d'abord un essai manqué. Nous nous contentons d'appliquer la même commande que pour les preuves concernant `plus'` :

```
intros n m; elim m; simpl; auto.

intros p Hrec.
...
n : nat
m : nat
p : nat
 Hrec : S (plus'' n p) = plus'' (S n) p
============================
  S (plus'' (S n) p) = plus'' (S (S n)) p
```

Il apparaît évident maintenant que notre démonstration est partie dans une mauvaise direction. L'hypothèse `Hrec` ne peut être utilisée que lorsque `plus''` est appliquée sur `n` ou " `S n` " comme premier argument (l'argument secondaire), mais `plus''` est appliquée à " `S (S n)` " dans la conclusion. La solution est de s'assurer que que l'hypothèse `Hrec` sera quantifiée universellement sur cet argument `n`. Pour cela, il faut que le but contienne une quantification universelle au moment où l'on engage la démonstration par récurrence. Reprenons la preuve depuis le début :

```
Restart.
intros n m; generalize n; elim m; simpl.
auto.

intros p Hrec n0.
...
p : nat
Hrec : ∀ n:nat, S (plus'' n p) = plus'' (S n) p
n0 : nat
============================
S (plus'' (S n0) p) = plus'' (S (S n0)) p
trivial.
Qed.
```

Exercice 10.13 Démontrer que la fonction `plus'` est associative, sans utiliser le fait que (`plus` m n) et (`plus'` m n) sont toujours égaux.

Exercice 10.14 * Démontrer que la fonction `plus''` est associative.

Exercice 10.15 * Définir une fonction qui calcule la suite de Fibonacci avec un algorithme récursif terminal. Démontrer que cette fonction permet d'associer la même valeur à tout nombre entier que la fonction définie dans l'exercice 10.8 page 301.

10.4 ** Division binaire

Dans cette section, nous allons décrire un algorithme de calcul de la division euclidienne pour des nombres représentés sous forme binaire. Nous basons notre développement sur le codage binaire des entiers développé par P. Crégut et fourni dans la bibliothèque `ZArith` de *Coq*. Dans cette représentation, la division et la soustraction ont une complexité polynômiale en la taille de la représentation binaire des nombres, c'est à dire polynômiale en le logarithme des nombres représentés ce qui est rapidement plus efficace que les algorithmes sur la représentation unaire fournie par le type `nat`. Cet algorithme repose sur la représentation des nombres entiers strictement positifs donnée par le type `positive` et présentée en section 7.3.4 page 195.

10.4.1 Division faiblement spécifiée

Nous voulons définir une fonction `div_bin` dont le type soit le suivant :

$$\texttt{positive} \rightarrow \texttt{positive} \rightarrow \texttt{Z*Z}$$

Nous préférons définir une fonction récursive de ce type plutôt qu'une fonction de type “ $\texttt{Z} \rightarrow \texttt{Z} \rightarrow \texttt{Z}$ ” parce que le premier type permet de construire une fonction structurelle récursive en suivant la structure des nombres positifs, ce que le second ne permet pas. La structure de donnée binaire suggère que l'algorithme pour la division de n par m fonctionne de manière différente suivant trois cas :

- si $n = 1$, il y a encore deux cas :
  - si $m = 1$, le quotient est 1 et le reste est $r = 0$,
  - si $m > 1$, le quotient est 0 et le reste est $r = 1$,
- si $n = 2n'$, alors soient q' et r' le quotient et le reste de la division de n' par m, soit
$$n' = q'm + r' \wedge 0 \leq r' < m$$
  Alors on a
$$n = 2q'm + 2r' \wedge 0 \leq 2r' < 2m.$$
  Deux cas se présentent encore :
  - si $2r' < m$, alors le quotient est $2q'$ et le reste est $2r'$,
  - si $2r' \geq m$, alors, puisque $2r' < 2m$, on peut en déduire que
$$0 \leq 2r' - m < m,$$
    le quotient est $2q' + 1$ et le reste est $2r' - m$,
- si $n = 2n' + 1$, alors soient encore q' et r' le quotient et le reste de la division de n' par m, soit
$$n' = q'm + r' \wedge 0 \leq r' < m$$
  Cette fois-ci on a $n = 2q'm + 2r' + 1$, et on peut déduire $r' + 1 \leq m$, puis $2r' + 2 \leq 2m$, puis $2r' + 1 < 2m$. Deux cas se présentent encore :
  - si $2r' + 1 < m$ alors le quotient est $2q'$ et le reste est $2r' + 1$,

– si $2r' + 1 \geq m$, alors on obtient

$$0 \leq 2r' + 1 - m < m,$$

le quotient est $2q' + 1$ et le reste est $2r' + 1 - m$.

Cet algorithme reproduit (en base 2) la méthode de division telle qu'elle est enseignée à l'école primaire. Sa description sous forme de fonction structurelle récursive est la suivante :

```
Open Scope Z_scope.

Fixpoint div_bin (n m:positive){struct n} : Z*Z :=
 match n with
 | 1%positive ⇒ match m with 1%positive ⇒(1,0) | v ⇒(0,1) end
 | xO n' ⇒
   let (q',r'):=div_bin n' m in
   match Z_lt_ge_dec (2*r')(Zpos m) with
   | left Hlt ⇒ (2*q', 2*r')
   | right Hge ⇒ (2*q' + 1, 2*r' - (Zpos m))
   end
 | xI n' ⇒
   let (q',r'):=div_bin n' m in
   match Z_lt_ge_dec (2*r' + 1)(Zpos m) with
   | left Hlt ⇒ (2*q', 2*r' + 1)
   | right Hge ⇒ (2*q' + 1, (2*r' + 1)-(Zpos m))
   end
 end.
```

Notez que la fonction `Z_lt_ge_dec` est une fonction fournie dans les bibliothèques de *Coq* et a le type suivant :

Z_lt_ge_dec : $\forall$ *x y:Z,* $\{x < y\}+\{x \geq y\}$

Nous allons maintenant démontrer que la fonction `div_bin` effectue bien les calculs attendus d'une fonction de division. La propriété à montrer est la suivante :

```
∀(n m:positive)(q r:Z), Zpos m ≠ 0 → div_bin n m = (q, r)→
     Zpos n = q*(Zpos m)+r  ∧ 0 ≤ r < (Zpos m)
```

Commençons par prouver les différents théorèmes qui expriment la propriété d'intervalle sur le reste. Le premier cas correspond à la division de 1 par 1, le reste vaut alors 0.

```
Theorem rem_1_1_interval : 0 ≤ 0 < 1.
Proof.
 omega.
Qed.
```

Le deuxième cas correspond à la division de 1 par un nombre pair non-nul, le reste vaut alors 1.

```
Theorem rem_1_even_interval :
   ∀m:positive,  0 ≤ 1 < Zpos (xO m).
```

Nous tentons d'abord d'effectuer cette démonstration en utilisant `omega`, mais ceci ne fonctionne pas :

```
intros; omega.
```
Error: omega can't solve this system

Il semble que `omega` ne reconnaît pas que `POS(xO n')` est supérieur ou égal à 2. Nous décomposons manuellement pour traiter rapidement la comparaison à zéro :

```
intros n'; split.
auto with zarith.
```

Le but qui reste est de la forme suivante :

```
...
n' : positive
============================
  1 < Zpos (xO n')
```

Pour chercher une solution, nous interrogeons le système *Coq* pour qu'il nous informe de tous les théorèmes qui s'appliquent sur ce motif. La commande à envoyer est la suivante :

```
SearchPattern (1 < Zpos _).
```

La réponse est vide. Il n'y a aucun théorème connu. Nous devons chercher une méthode pour en démontrer un. Pour cela, cherchons d'abord à comprendre comment `<` est défini sur les entiers relatifs :

```
Locate "_ < _".
...
"x < y" := lt x y                : nat_scope
"x < y" := Zlt x y               : Z_scope (default interpretation)
```

```
Print Zlt.
Zlt = fun x y:Z ⇒ (x?=y)=Lt
     : Z→Z→Prop
Argument scopes are [Z_scope Z_scope]
```

Ceci nous indique que le prédicat `<` est en fait un prédicat calculé par une fonction. Pour le simplifier, il est donc raisonnable de faire appel à une commande de conversion du but par calcul.

```
compute.
 ...
 ============================
```

$Lt = Lt$

```
 auto.
Qed.
```

Paradoxalement, aucune des tactiques automatiques fournies par *Coq* ne sait résoudre ce problème si nous n'insérons pas cette tactique de conversion. Ceci montre que la bibliothèque de théorèmes relatifs à l'arithmétique sur `Z` est incomplète. Ce problème sera probablement corrigé dans des versions futures du système, mais il faut retenir le type de démarche que nous pouvons suivre pour trouver une solution lorsque les outils usuels échouent.

Nous pouvons maintenant construire le théorème similaire pour le cas de la division de 1 par un nombre impair supérieur à 1.

```
Theorem rem_1_odd_interval : ∀m:positive, 0 ≤ 1 < Zpos (xI m).
Proof.
 split;[auto with zarith | compute; auto].
Qed.
```

Il nous reste quatre cas généraux à traiter, suivant que le dividende est pair ou impair et que le double (ou le double plus un) du reste obtenu par l'appel récursif est plus grand ou plus petit que le diviseur. Ces cas font appel à des raisonnements simples sur les intervalles. La tactique `omega` est particulièrement bien adaptée à cette tâche.

```
Theorem rem_even_ge_interval :
 ∀m r:Z, 0 ≤ r < m →  2*r ≥ m → 0 ≤ 2*r - m < m.
Proof.
 intros; omega.
Qed.

Theorem rem_even_lt_interval :
 ∀m r:Z, 0 ≤ r < m → 2*r < m → 0 ≤ 2*r < m.
Proof.
 intros; omega.
Qed.

Theorem rem_odd_ge_interval :
 ∀m r:Z, 0 ≤ r < m → 2*r + 1 ≥ m → 2*r + 1 - m <  m.
Proof.
 intros; omega.
Qed.

Theorem rem_odd_lt_interval :
 ∀m r:Z, 0 ≤ r < m → 2*r + 1 < m → 0 ≤ 2*r + 1 < m.
Proof.
 intros; omega.
Qed.
```

Les démonstrations sur la fonction div_bin sont naturellement complexes car elles doivent suivre la structure de l'algorithme, qui est elle-même assez longue. Pour compenser cette longueur nous définissons d'abord une tactique qui effectue tous les traitements par cas. Définir une tactique spécialement adaptée pour une fonction est particulièrement utile si la fonction est complexe et les raisonnements sur cette fonction sont fréquents. Cette tactique spécialisée est assez facile à obtenir en regroupant les étapes élémentaires d'une démonstration assistée par le système *Coq*.

```
Ltac div_bin_tac arg1 arg2 :=
  elim arg1;
   [intros p; lazy beta iota delta [div_bin]; fold div_bin;
      case (div_bin p arg2); unfold snd; intros q' r' Hrec;
      case (Z_lt_ge_dec (2*r' + 1)(Zpos arg2)); intros H
   | intros p; lazy beta iota delta [div_bin]; fold div_bin;
      case (div_bin p arg2); unfold snd; intros q' r' Hrec;
      case (Z_lt_ge_dec (2*r')(Zpos arg2)); intros H
   | case arg2; lazy beta iota delta [div_bin]; intros].
```

Par comparaison avec la description de la fonction div_bin cette tactique peut sembler ne pas bien suivre la structure de la fonction. La raison de cette différence est que la fonction div_bin ne traite pas les différents cas possibles pour l'argument principal dans l'ordre naturel des constructeurs du type positive. Le premier cas traité dans la tactique est donc celui pour le constructeur xI, puis vient le constructeur xO, puis le constructeur xH alors qu'ils sont traités dans l'ordre xH, xO, puis xI dans la fonction. Le lecteur averti aura également noté que nous utilisons les tactiques lazy et fold pour restreindre les simplification à la fonction div_bin qui est la fonction étudiée. Enfin, notons que le comportement de cette tactique que nous avons écrite à la main est également fourni par la tactique functional induction issue des travaux de P. Courtieu et G. Barthe [9].

Nous allons maintenant ajouter les différents théorèmes établis ci-dessus dans la base de données utilisée par la tactique auto.

```
Hint Resolve rem_odd_ge_interval rem_even_ge_interval
 rem_odd_lt_interval rem_even_lt_interval rem_1_odd_interval
 rem_1_even_interval rem_1_1_interval.
```

Avec la tactique spécialisée div_bin_tac et les théorèmes pour chaque cas dans la base de donnée, le script décrivant la démonstration devient très simple.

```
Theorem div_bin_rem_lt :
 ∀n m:positive, 0 ≤ snd (div_bin n m) < Zpos m.
Proof.
 intros n m; div_bin_tac n m; unfold snd; auto.
 omega.
Qed.
```

tion changement de POS vers Zpos le /03

Pour être complet, il faut aussi établir l'égalité entre le diviseur et la somme du reste et du produit du quotient et du diviseur. La démonstration va reposer sur l'utilisation de la tactique `ring` pour vérifier les égalités. Néanmoins pour chaque cas récursif, il faut interpréter les termes `Zpos(xO p)` et `Zpos(xI p)` comme le résultat de multiplications et d'additions à partir de « `Zpos p` ». Pour retrouver les théorèmes qui expriment cette interprétation, nous pouvons utiliser la commande `SearchRewrite` :

```
SearchRewrite (Zpos (xI _)).
```

...

*Zpos_xI: ∀ p:positive, Zpos (xI p) = 2 * Zpos p + 1*

```
SearchRewrite (Zpos (xO _)).
```

*Zpos_xO: ∀ p:positive, Zpos (xO p) = 2 * Zpos p*

Avec ces théorèmes, la démonstration se résume aux quelques lignes suivantes :

```
Theorem div_bin_eq :
 ∀n m:positive,
   Zpos n =  (fst (div_bin n m))*(Zpos m) + snd (div_bin n m).
Proof.
 intros n m; div_bin_tac n m;
  rewrite Zpos_xI || (try rewrite Zpos_xO);
  try rewrite Hrec; unfold fst, snd; ring.
Qed.
```

Cette démonstration est volontairement décrite de façon concise. Nous invitons le lecteur intrigué à rejouer cette démonstration en décomposant chacune des tactiques.

10.4.2 Division bien spécifiée

Nous allons maintenant redéfinir un algorithme de calcul de reste par division binaire, mais cette fois-ci en lui donnant un type expressif, à l'aide de types dépendants. Nous commençons par fournir une propriété inductive qui exprime à quelles conditions un couple de nombres est bien le résultat d'une division de deux autres nombres.

```
Inductive div_data (n m:positive) : Set :=
div_data_def :
 ∀q r:Z, Zpos n = q*(Zpos m)+r → 0≤ r < Zpos m →
  div_data n m.
```

Il s'agit ici d'un usage typique des types inductifs de la sorte `Set` qui réunissent des données (`q` et `r`) et des propriétés de ces données, comme nous l'avons décrit en section 10.1.1. Nous aurions pu utiliser les types `sig` et `sigS` pour représenter la même structure de donnée, mais une définition de type spécifique permet une écriture plus concise.

Nous allons maintenant définir notre fonction par preuve. Nous savons que notre algorithme est récursif sur le premier argument de la fonction et que le second argument sera le même lors de tous les appels récursifs. Ceci se traduit en faisant une preuve par récurrence sur n en plaçant également m dans le contexte.

```
Definition div_bin2 : ∀n m:positive, div_data n m.
  intros n m; elim n.
```

Le premier but nous demande de déterminer la valeur pour le cas où $n = 2 * n' + 1$, sachant que nous disposons de la valeur de la division de n' par m, qui s'exprime à l'aide de deux nombres q et r contraints par deux hypothèses :

```
  intros n' [q r H_eq H_int].
...
H_eq : Zpos n' = q * Zpos m + r
H_int : 0 ≤ r < Zpos m
============================
  div_data (xI n') m
```

Pour déterminer le quotient et le reste nous avons besoin de comparer $2r+1$ et m, ce qui est exprimé par la tactique suivante :

```
case (Z_lt_ge_dec (2*r + 1)(Zpos m)).
...
H_eq : Zpos n' = q * Zpos m + r
H_int : 0 ≤ r < Zpos m
============================
  2*r + 1 < Zpos m → div_data (xI n') m
```

Lorsque $2r + 1$ est assez petit, il peut être utilisé comme reste et le résultat de la division est constitué de $2q$ et $2r + 1$; il ne reste ensuite qu'à démontrer l'égalité et les comparaisons requises par `div_data_def`. Ces preuves se font comme pour les théorèmes de correction de `div_bin`.

```
exists (2*q)(2*r + 1).
rewrite Zpos_xI; rewrite H_eq; ring.
auto.
```

Lorsque $2r + 1$ est trop grand, il faut lui soustraire m et incrémenter le quotient. La suite de la démonstration a donc la forme suivante :

```
exists (2*q+1)(2*r + 1 - (Zpos m)).
rewrite Zpos_xI; rewrite H_eq; ring.
omega.
```

Ainsi, le système *Coq* nous a accompagné dans le processus de description de l'algorithme, en nous indiquant dans chaque but ce qu'il fallait décrire à cette étape de l'algorithme. Le reste de la démonstration, qui doit couvrir les cas où $n = 2n'$ et $n = 1$ est effectué par des tactiques de même complexité, mais nous ne le donnons pas ici.

Cette définition par preuve est facilitée par le système *Coq*, mais on est parfois gêné par le manque de transparence de la définition où la structure algorithmique de la fonction est cachée. Il est possible d'avoir une meilleure lisibilité si nous utilisons la tactique `refine` pour donner le contenu algorithmique, en laissant des trous pour les preuves. Toutefois la lisibilité obtenue est limitée par la nécessité de décrire le typage des constructions de filtrage dépendant qui apparaissent dans cette démonstration. Ici encore, ce terme assez complexe n'a pas été construit directement, mais il est obtenu au terme d'un dialogue où l'assistant de preuve a aidé à connaître l'expression attendue à chaque endroit. Il est en effet possible de commencer avec une expression où tout un sous-terme est remplacé par un joker et de déterminer la valeur attendue pour ce joker, également à l'aide de la tactique `refine`. En fin de travail, il est intéressant de regrouper toutes les étapes dans un seul terme plus synthétique.

```
Definition div_bin3 : ∀n m:positive, div_data n m.
 refine
  ((fix div_bin3 (n:positive) : ∀m:positive, div_data n m :=
      fun m ⇒
        match n return div_data n m with
        | 1%positive ⇒
            match m return div_data 1 m with
            | 1%positive ⇒ div_data_def 1 1 1 0 _ _
            | xO p ⇒ div_data_def 1 (xO p) 0 1 _ _
            | xI p ⇒ div_data_def 1 (xI p) 0 1 _ _
            end
        | xO p ⇒
            match div_bin3 p m with
            | div_data_def q r H_eq H_int ⇒
                match Z_lt_ge_dec (Zmult 2 r)(Zpos m) with
                | left hlt ⇒
                    div_data_def (xO p) m (Zmult 2 q)
                                 (Zmult 2 r) _ _
                | right hge ⇒
                    div_data_def (xO p) m (Zplus (Zmult 2 q) 1)
                      (Zminus (Zmult 2 r)(Zpos m)) _ _
                end
            end
        | xI p ⇒
            match div_bin3 p m with
            | div_data_def q r H_eq H_int ⇒
                match Z_lt_ge_dec (Zplus (Zmult 2 r) 1)(Zpos m)
```

```
                with
                | left hlt ⇒
                    div_data_def (xI p) m (Zmult 2 q)
                      (Zplus (Zmult 2 r) 1) _ _
                | right hge ⇒
                    div_data_def (xI p) m (Zplus (Zmult 2 q) 1)
                      (Zminus (Zplus (Zmult 2 r) 1)(Zpos m)) _ _
                end
            end
        end));
      clear div_bin3; try rewrite Zpos_xI; try rewrite Zpos_xO;
      try rewrite H_eq; auto with zarith; try (ring; fail).
 split;[auto with zarith | compute; auto].
 split;[auto with zarith | compute; auto].
Defined.
```

Il faut noter que le terme fourni à la tactique `refine` contient une construction `fix`, qui introduit donc un terme `div_bin3` dans le contexte. Il est d'une importance capitale que ce terme `div_bin3` soit utilisé avec précaution dans la construction de la solution, car il ne peut être appliqué qu'aux sous-termes structurels de l'argument principal. Pour cette raison, nous effaçons ce terme du contexte grâce à la tactique `clear div_bin3` avant de faire appel à des tactiques de démonstration automatique.

Les utilisations de la tactique `auto` dans les définitions par preuve de `div_bin2` et `div_bin3` reposent sur les mêmes théorèmes que ceux que nous avons utilisés pour démontrer que la fonction `div_bin` était correcte.

Nous pouvons tirer deux leçons de ces exemples. Premièrement, le changement de structure de données peut servir pour décrire et implémenter des algorithmes plus efficaces. Nous aurons l'occasion d'y revenir lorsque nous parlerons d'extraction et de tactiques basées sur la réflexion. Deuxièmement, la construction d'une fonction avec un type expressif est plus difficile que la construction de la fonction correspondante avec un type simple, mais la construction de termes par preuve, en particulier avec la tactique `refine` rend cette tâche plus abordable. Sur le long terme, la fonction bien spécifiée a probablement un coût de développement moindre, car la preuve que la fonction faiblement spécifiée vérifie les bons théorèmes est aussi coûteuse que la construction de la fonction bien spécifiée.

Exercice 10.16 *** Le but de cet exercice est de développer une fonction de racine carrée d'un nombre n. Si n' est le quart de n, s est la racine carrée entière de n' (arrondie par défaut) et si r est le reste tel que $r = n' - s^2$, alors $2s'$ ou $2s'+1$ est la racine carrée de n. En déduire un algorithme pour calculer la racine carrée par défaut d'un nombre n ainsi que le reste associé, et une fonction satisfaisant la spécification suivante :

```
∀p:positive,
  {s:Z &{r:Z | Zpos p = s*s + r ∧ s*s ≤ Zpos p < (s+1)*(s+1)}}
```

Exercice 10.17 *** En utilisant le résultat de l'exercice 10.8 page 301, exprimer u_{2n} et u_{2n+1} en fonction de u_n et u_{n+1}, où u_n désigne le $n$ième terme de la suite de Fibonacci. En déduire une réalisation de la spécification suivante qui passe par un algorithme récursif structurel sur les nombres de type `positive` :

```
∀n:nat, {u:nat & {u':nat | u = fib n ∧ u' = fib (n+1)}}
```

Chapitre 11

* Extraction et programmation impérative

Le système *Coq* permet de modéliser des programmes en les décrivant comme des fonctions dans un langage fonctionnel pur. Il faut cependant bien remarquer que le système *Coq* a pour but de synthétiser des programmes corrects ou de vérifier des programmes, mais pas de les exécuter. Cette tâche est normalement déléguée aux outils habituels (compilateurs, machines abstraites, etc.)

Pour permettre la génération automatique de code exécutable certifié à partir des modèles formels, le système fournit deux approches. La première est basée sur la traduction des programmes fonctionnels du Calcul des Constructions vers des programmes fonctionnels purs d'un langage fonctionnel efficace, principalement le langage *OCAML*. La seconde approche fournit la possibilité de décrire directement des programmes *impératifs* dans le système *Coq*, ainsi que deux outils de traduction : le premier associe au programme représenté un programme fonctionnel pur qui effectue les mêmes opérations et demande à l'utilisateur de vérifier une collection d'obligations de preuves nécessaires pour que ce programme fonctionnel soit reconnu bien formé ; le second outil de traduction permet de produire un programme impératif, encore dans le langage *OCAML*. Les programmes impératifs ainsi obtenus peuvent donc mélanger des traits purement fonctionnels et des instructions impératives.

11.1 Extraction vers les langages fonctionnels

Les fonctions écrites en *Coq* sont souvent les modèles de fonctions écrites dans un langage fonctionnel. En retrouvant automatiquement la fonction de ce langage fonctionnel dont la fonction *Coq* est un modèle, on dispose d'une technique de production de logiciel. On parle alors d'*extraction*. En général, le comportement de la fonction *Coq* permet de prévoir le comportement de la fonction extraite : on dispose d'une chaîne de production de logiciel certifié, puisque les programmes extraits répondent aux spécifications décrites dans les développements formels.

Deux difficultés doivent être contournées dans la production de fonctions extraites. D'une part, les langages de programmation usuels ne disposent pas de types dépendants et les fonctions bien typées dans le Calcul des Constructions ne correspondent pas toujours à des fonctions bien typées dans le langage de programmation fonctionnel visé. D'autre part, les fonctions du Calcul des Constructions contiennent des calculs effectués dans le seul but de fabriquer des démonstrations, mais ces démonstrations n'ont qu'un intérêt limité pour le résultat final. Garder ces calculs mènerait à des implémentations inefficaces des algorithmes. En fait les calculs effectués pour les démonstrations correspondent à des vérifications qui devraient être faites une fois pour toutes, *au moment de la compilation*, tandis que les calculs sur les données effectives doivent être effectué pour chaque donnée présentée à la fonction, *au moment de l'exécution*. Cette séparation entre calculs *au moment de la compilation* ou *au moment de l'exécution* montre les liens qui peuvent exister entre l'extraction de programmes et les techniques de l'évaluation partielle.

Dans le Calcul des Constructions sous sa forme actuelle, la distinction entre les sortes `Prop` et `Set` sert justement à distinguer entre les calculs sur les aspects logiques qui peuvent être effectués au moment de la compilation et les calculs effectifs sur les données qui doivent apparaître au moment de l'exécution.

11.1.1 Le mécanisme d'extraction

Le procédé d'extraction effectue deux opérations distinctes, suivant que l'on traite une définition de type ou une définition de fonction ou de valeur. Pour une définition de type inductif, l'opération consiste à éliminer les arguments de type et de preuve dans les constructeurs. Pour les définitions de fonctions ou de valeurs, les expressions représentant des types sont éliminées, les expressions représentant des preuves le sont généralement, et elles sont parfois remplacées par une valeur unique, ce qui confirme la non-pertinence des preuves. Le nombre d'arguments des fonctions doit alors changer, puisque certains arguments peuvent disparaître. Les contraintes de typage et les restrictions sur les règles d'élimination des types inductifs assurent que les données retournées par ces fonctions ne dépendent jamais des arguments de preuve ou de type, ce qui justifie ce remplacement systématique.

Extraire les types et fonctions non polymorphes

Lorsqu'un type ne présente pas de dépendance, sa mise en correspondance avec un type du langage *OCAML* est immédiate. Par exemple, la déclaration des entiers naturels en *Coq*, donnée par le type inductif suivant :

```
Inductive positive : Set :=
  xI : positive→positive
| xO : positive→positive
| xH : positive.
```

est facilement extraite dans la définition de type suivante :

```
type positive =
  | XI of positive
  | XO of positive
  | XH
```

Les fonctions qui calculent sur ce type sans faire apparaître de types dépendants se traduisent directement dans des fonctions du langage fonctionnel cible. Par exemple la fonction de soustraction sur les entiers naturels est décrite dans les bibliothèques de *Coq* par la définition récursive suivante :

```
Fixpoint Psucc (x:positive) : positive :=
  match x with
    xI x’ ⇒ xO (Psucc x’) | xO x’ ⇒ xI x’ | xH ⇒ xO xH
  end.
```

La traduction vers *OCAML* est directe en remplaçant le mot-clef `Fixpoint` par “`let rec`”, les abstractions “`fun ... ⇒`” par des abstractions “`fun ... ->`” et les filtrages par des filtrages.

```
let rec psucc = function
  | XI x’ -> XO (psucc x’)
  | XO x’ -> XI x’
  | XH -> XO XH
```

Extraire le polymorphisme

Certaines structures de données utilisent les types dépendants simplement pour décrire le polymorphisme. C’est le cas de la structure de donnée des listes, donnée par la définition inductive suivante :

```
Inductive list (A:Set) : Set :=
  nil : list A | cons : A → list A → list A.
```

En pratique, cette déclaration donne aux fonctions `nil` et `cons` les types suivants :

nil : $\forall$ *A:Set, list A*
cons : $\forall$ *A:Set, A* $\rightarrow$ *list A* $\rightarrow$ *list A*

Si l’on cherche à produire un programme `OCaml`, cette déclaration de type correspond à la déclaration de type suivante :

```
type ’a coqlist = Nil | Cons of ’a * ’a coqlist
```

D’un point de vue pratique, la fonction *Coq* `nil` est remplacée par le constructeur `Nil`, qui n’est pas une fonction mais a un type polymorphe, et la fonction *Coq* `cons` est remplacée par un constructeur, qui ne peut pas être directement utilisé comme une fonction en *OCAML*, à cause de restrictions spécifiques à ce langage. Néanmoins, nous avons le typage suivant :

```
# Nil
- : 'a coqlist = Nil
# (fun x y -> Cons(x,y))
- : 'a -> 'a coqlist -> 'a coqlist = <fun>
```

D'un point de vue purement symbolique, l'argument `A` de type `Set` disparaît dans l'extraction. En *Gallina*, le polymorphisme est exprimé par des produits dépendants, tandis qu'en *OCAML* il s'exprime seulement par des paramètres de type polymorphes `'a`, etc. Prenons par exemple la fonction de concaténation de listes `app` :[1] :

```
Fixpoint app (A:Set)(l m:list A){struct l} : list A :=
  match l with
  | nil ⇒ m
  | cons a l1 ⇒ cons A a (app A l1 m)
  end.
```

À l'extraction, le type de `app` perd l'argument correspondant au polymorphisme. Les paramètres réels de `cons` et `app` correspondant aux arguments de types sont enlevés du code :

```
let rec app l =
(fun m -> match l with
            Nil -> m
          | Cons(a,l1) -> Cons(a,app l1 m))
```

Exercice 11.1 Construire l'extraction du type `option` et de la fonction `nth'` :

```
Inductive option (A:Set) : Set :=
  Some : A → option A | None : option A.

Implicit Arguments Some [A].
Implicit Arguments None [A].

Fixpoint nth' (A:Set)(l:list A)(n:nat){struct n} : option A :=
  match l, n with
    nil, _ ⇒ None
  | cons a tl, O ⇒ Some a
  | cons a tl, S p ⇒ nth' A tl p
  end.
```

Oubli des preuves

Lorsque des types inductifs contiennent des propositions, l'extraction fait disparaître ces propositions. Considérons par exemple le type `sumbool` donné dans les bibliothèques de *Coq* par la déclaration suivante :

1. Pour un discours plus clair, nous avons rendus explicites tous les arguments de `app` et `cons` qui sont normalement implicites dans les bibliothèques de *Coq*.

```
Inductive sumbool (A B:Prop) : Set :=
   left : A → sumbool A B | right : B → sumbool A B.
```

L'extraction de ce type fournit le type suivant :

```
type sumbool = Left | Right
```

Ainsi, les deux types `A` et `B` donnés en argument ne réapparaissent même pas comme arguments de polymorphisme, puisque les constructeurs n'auront aucun argument. Cette extraction respecte l'interprétation du type `sumbool` comme une version *enrichie* du type `bool`. En quelque sorte, l'extraction effectue l'*appauvrissement* inverse.

Pour certains types, comme le type `sig`, ceci mène à construire un type inductif à un seul constructeur qui a un seul argument (puisque le deuxième qui serait une proposition disparaît). De tels types sont inutiles et rendent le code extrait moins lisible et moins efficace. Ces types inductifs sont simplement supprimés et l'usage du constructeur dans le code disparaît systématiquement.

Pour les fonctions, le même procédé s'applique naturellement. Ainsi la fonction qui compare deux nombres naturels aurait pu être donnée en *Coq* par la définition suivante :

```
Fixpoint eq_positive_dec (n m:positive){struct m} :
   {n = m}+{n ≠ m} :=
  match n return {n = m}+{n ≠ m} with
  | xI p ⇒
    match m return {xI p = m}+{xI p ≠ m} with
    | xI q ⇒
      match eq_positive_dec p q with
      | left heq ⇒ left _ (eq_xI p q heq)
      | right hneq ⇒ right _ (not_eq_xI p q hneq)
      end
    | xO q ⇒ right _ (xI_xO p q)
    | xH ⇒ right _ (sym_not_equal (xH_xI p))
    end
  | xO p ⇒
    match m return {xO p = m}+{xO p ≠ m} with
    | xI q ⇒ right _ (sym_not_equal (xI_xO q p))
    | xO q ⇒
      match eq_positive_dec p q with
      | left heq ⇒ left _ (eq_xO p q heq)
      | right hneq ⇒ right _ (not_eq_xO p q hneq)
      end
    | xH ⇒ right _ (sym_not_equal (xH_xO p))
    end
  | xH ⇒ match m return {xH = m}+{xH ≠ m} with
                 | xI q ⇒ right _ (xH_xI q)
                 | xO q ⇒ right _ (xH_xO q)
```

```
                  | xH ⇒ left _ (refl_equal xH)
                  end
  end.
```

En appliquant systématiquement la méthode exposée jusqu'à maintenant on pourrait obtenir l'expression suivante :

```
let rec eq_positive_dec n m =
 match n with
 | XI p ->
    (match m with
     | XI q ->
        (match eq_positive_dec p q with
          Left -> Left | Right -> Right)
     | XO q -> Right
     | XH -> Right)
 | XO p ->
    (match m with
     | XI q -> Right
     | XO q ->
       (match eq_positive_dec p q with
          Left -> Left | Right -> Right)
     | XH -> Right)
 | XH -> (match m with
           XI q -> Right | XO q -> Right | XH -> Left)
```

Mais une optimisation simple consiste à reconnaître que l'expression

```
(match eq_positive_dec p q with Left -> Left | Right -> Right)
```

est équivalente à l'expression `eq_nat_dec p q`, de sorte que la fonction extraite ci-dessus devrait être présentée de la manière suivante :

```
let rec eq_positive_dec n m =
 match n with
 | XI p ->
  (match m with
  | XI q -> eq_positive_dec p q | XO q -> Right | XH -> Right)
 | XO p ->
  (match m with
  | XI q -> Right | XO q -> eq_positive_dec p q | XH -> Right)
 | XH ->
  (match m with | XI q -> Right | XO q -> Right | XH -> Left)
```

Ce type d'optimisation est également effectué par l'outil d'extraction de *Coq* (lorsque les optimisations sont autorisées).

Le système d'extraction est cohérent : les fonctions « productrices » de preuves changent de type en même temps que les fonctions « consommatrices ».

Les premières ne produisent plus de preuves et les secondes n'en consomment plus. Considérons par exemple la fonction calculant le prédécesseur d'un nombre naturel et non définie en zéro. Les deux arguments de cette fonction sont un nombre naturel n et une preuve que n est non nul.

```
Definition pred' (n:nat) : n ≠ 0 → nat :=
  match n return n ≠ 0 → nat with
  | O ⇒ fun h:0 ≠ 0 ⇒ False_rec nat (h (refl_equal 0))
  | S p ⇒ fun h:S p ≠ 0 ⇒ p
  end.
```

La fonction extraite pour `pred'` a la forme suivante :

```
let pred' n = match n with O -> assert false | (S p) -> p
```

La fonction `pred'` dans le code extrait prend un argument de moins que la fonction `pred'` du Calcul des Constructions, mais le second argument n'était pas utilisé pour faire le calcul. Dans les usages manuels, c'est la responsabilité de l'utilisateur de n'utiliser cette fonction que lorsque l'argument n'est pas nul. Pour le code extrait, le procédé d'extraction assure que les fonctions ne sont appelées que lorsque les vérifications de bon usage ont toutes été effectuées dans les preuves.

Pour illustrer ceci considérons une fonction `pred2` qui utilise la fonction `pred'` :

```
Definition pred2 (n:nat) : nat :=
  match eq_nat_dec n 0 with
  | left h ⇒ 0
  | right h' ⇒ pred' n h'
  end.
```

Dans cette fonction, l'hypothèse `h'` est la justification qui assure que la spécification est satisfaite pour les données en entrée de la fonction `pred'`. Dans le code extrait cette hypothèse est oubliée mais on sait au moment de l'extraction que la fonction appelée se comporte correctement.

```
let pred2 n =
   match eq_nat_dec n O with Left -> O | Right -> pred' n
```

Ici nous sommes assurés que `pred'` n'est appelée que lorsque `n` est non nul, car la valeur `Right` retournée par `eq_nat_dec` contient « moralement » l'assurance nécessaire.

Il existe néanmoins des valeurs pour lesquelles l'argument de preuve ne peut pas être enlevé directement. Ce sont des valeurs pour lesquelles il n'existe pas de valeur licite dans le monde des programmes. Par exemple, nous pouvons considérer la fonction suivante :

```
Definition pred'_on_O := pred' O.
```

Si l'on appliquait brutalement la technique de faire disparaitre les arguments de preuve, on devrait obtenir la valeur suivante :

```
let pred'_on_O = pred' O
```

Mais cette définition demande d'exécuter `assert false`, ce qui provoque une erreur à l'exécution. Pour contourner ce problème, l'argument représentant une proposition n'est pas enlevé des arguments de la fonction extraite, même si cet argument n'est effectivement pas utilisé pour le calcul. La valeur vraiment extraite est la suivante :

```
let pred'_on_O _ = pred' O
```

Nous pouvons être surs que cette fonction ne reçoit jamais d'arguments au cours de l'exécution de code extrait : ceci signifierait que l'on aurait trouvé une preuve de $0 \neq 0$.

****Récursion bien fondée et extraction**

La récursion bien fondée sera présentée en détail dans la section 16.2, mais nous l'abordons ici sous l'angle unique de l'extraction et le lecteur novice pourra laisser cette section en première lecture. Cette technique de programmation certifiée est particulière car elle est traitée en *Coq* par une récurrence sur une propriété inductive qui, puisqu'elle est une propriété, est amenée à disparaître dans l'extraction.

La récursion bien fondée est basée sur la propriété d'accessibilité décrite par une définition similaire à la suivante.

```
Inductive Acc (A:Set)(R:A→A→Prop) : A→Prop :=
    Acc_intro : ∀x:A, (∀y:A, R y x → Acc A R y)→ Acc A R x.
```

Chaque preuve d'accessibilité pour un `x` donné contient une fonction qui montre que les prédécesseurs de `x` par `R` sont accessibles. On peut également définir une fonction `Acc_inv` qui a la valeur suivante :

```
Definition Acc_inv (A:Set)(R:A→A→Prop)(x:A)(Hacc:Acc A R x) :
  ∀y:A, R y x → Acc A R y :=
  match Hacc as H in (Acc _ _ x)
        return (∀y:A, R y x → Acc A R y) with
  | Acc_intro x f ⇒ f
  end.
```

Si x et y sont des éléments d'un type A, si R est une relation sur A, H_a est une preuve de (`Acc` A R x), et H_r est une preuve de (R y x), alors le terme (`Acc_inv` A R x H_a y H_r) est une preuve de (`Acc` A R y). De plus, cette preuve est structurellement une sous-preuve de H_a (de façon similaire à ce que nous avons vu en section 7.3.5.1), ce qui sera utilisé pour définir des fonctions par la commande `Fixpoint`. Ainsi, un récurseur pour ce type inductif est décrit par la définition suivante (ce récurseur est plus simple que celui qui est effectivement utilisé dans *Coq*) :

```
Fixpoint Acc_iter (A:Set)(R:A→A→Prop)(P:A→Set)
 (f:∀x:A, (∀y:A, R y x → P y)→ P x)(x:A)
 (hacc:Acc A R x){struct hacc} : P x :=
  f x (fun (y:A)(hy:R y x) ⇒
       Acc_iter A R P f y (Acc_inv A R x hacc y hy)).
```

Une fonction `well_founded_induction` est ensuite définie de la façon suivante :

```
Definition well_founded_induction (A:Set)(R:A→A→Prop)
  (Rwf:∀x:A, Acc A R x)(P:A→Set)
  (F:∀x:A, (∀y:A, R y x → P y)→ P x)(x:A) : P x :=
  Acc_iter A R P F x (Rwf x).
```

A l'extraction, on obtient les fonctions suivantes :

```
let rec acc_iter f x = f x (fun y _ -> acc_iter f y)

let well_founded_induction f x = acc_iter f x
```

En effet, les trois premiers arguments de `Acc_iter` sont soit des types, soit des fonctions retournant des types. Le quatrième argument est une fonction qui retourne un élément dans un ensemble de type `Set`, il est donc conservé, le cinquième argument a pour type `A` et `A` a pour type `Set` et est donc conservé et le sixième argument est de type "`Acc A R x`". La fonction `Acc_iter` a donc seulement deux arguments significatifs. Observons maintenant son premier argument, qui est extrait du quatrième argument de `Acc_iter`. C'est une fonction qui prend deux arguments. Le premier argument est de type `A` et est conservé parce que `A` est de sorte `Set`. Le second argument est une fonction qui retourne un élément dans un type de sorte `Set` et est donc conservé. La fonction `f` est donc représentée dans le code extrait par une fonction à deux arguments également.

La valeur de `Acc_iter` est une application de la fonction `f`. Le deuxième argument est une application de `Acc_iter` dont les arguments de rangs un à trois et le sixième disparaissent. Ceci explique la simplicité de la fonction `Acc_iter` obtenue.

Si l'on définit une fonction récursive bien fondée on est amené à construire la fonctionnelle qui sera donnée en 5e argument de `well_founded_induction`, il s'agira encore d'une fonction qui fait intervenir données et preuves. Pour montrer rapidement le fonctionnement de l'extraction sur de telles fonctions, nous allons supposer que les preuves nécessaires sont déjà faites pour une description d'une fonction de calcul de logarithme discret.

```
Fixpoint div2 (n:nat) : nat :=
  match n with S (S p) ⇒ S (div2 p) | _ ⇒ 0 end.
```

```
Hypotheses
  (div2_lt : ∀x:nat, div2 (S x) < S x)
  (lt_wf : ∀x:nat, Acc lt x).

Definition log2_aux_F (x:nat) : (∀y:nat, y < x → nat)→nat :=
  match x return (∀y:nat, y < x → nat)→ nat with
  | O ⇒ fun _ ⇒ 0
  | S p ⇒ fun f ⇒ S (f (div2 (S p))(div2_lt p))
  end.

Definition log2_aux :=
  well_founded_induction lt_wf (fun _:nat ⇒ nat) log2_aux_F.

Definition log2 (x:nat)(h:x ≠ 0) : nat := log2_aux (pred x).
```

Le code extrait pour la fonction `log2_aux` a la forme suivante :

```
let log2_aux x =
  well_founded_induction
   (fun x f ->
     match x with
       O -> O
     | (S p) -> (S (f (div2 (S p)) __)))
```

Si l'on disposait d'outils de transformations de programmes pour le langage *OCAML*, on s'apercevrait après quelques $\beta\eta$-conversions, que cette définition est équivalente à la suivante :

```
let rec log2_aux x =
    match x with O -> O | S p -> S (log2_aux (div2 (S p)))
```

Nous avons fini d'exposer une méthode complète d'extraction de code fonctionnel sans types dépendants à partir d'expressions du calcul des constructions inductives. Cette méthode est approximativement celle qui était utilisée dans *Coq*, dans la version `7.2`. En fait, l'outil d'extraction contient quelques optimisations spécifiques pour la récursion bien fondée, de telle sorte que si l'on utilise `well_founded_induction` au lieu de la fonction que nous avons décrite dans cette section, le code extrait est directement le code équivalent que nous venons de présenter [60]. Notre introduction du type `Acc` n'avait qu'un but didactique permettant de mieux comprendre le mécanisme d'extraction.

11.1.2 La dualité `Prop`/`Set` et l'extraction

La façon dont l'extraction exploite la sorte `Prop` est un exemple supplémentaire du principe de non pertinence des preuves. Les valeurs des preuves n'ont pas d'importance, seule leur existence compte. La première méthode d'extraction allait même plus loin dans l'indifférence, puisque même l'existence des preuves était laissée dans l'oubli par cette méthode, avec l'argument informel suivant : si le calcul d'une certaine fonction est requis, c'est que le terme du Calcul des Constructions correspondant a reçu les preuves nécessaires, il n'est donc pas nécessaire de vérifier même l'existence de la preuve.

Nous avons volontairement évité d'expliquer les justifications qui assurent que chaque fonction extraite se comporte de manière cohérente avec la fonction du Calcul des Constructions associée. Ce sujet est assez complexe et au delà des prétentions de cet ouvrage et nous invitons le lecteur à se reporter à l'article [74] pour une référence plus exacte.

Néanmoins, la cohérence de l'extraction permet de justifier les choix qui ont été effectués dans le typage des constructions de filtrage.

Filtrage sur les propositions

Il n'est pas permis de filtrer une proposition inductive pour produire une donnée de type `Set` ou une donnée de type `Type`. Attardons-nous sur la première partie de cette restriction. S'il était permis de filtrer une propriété inductive pour construire une valeur dont le type est de sorte `Set`, on pourrait ainsi obtenir des données différentes en fonction de preuves différentes.

Supposons que l'on ait permis cette forme de filtrage, on pourrait alors écrire une valeur de la forme suivante :

```
Definition or_to_nat (A,B:Prop)(A∨B) : nat :=
  match H with or_introl h ⇒ S O | or_intror h ⇒ O end.
```

L'application de la fonction "`or_to_nat A B H`" retournerait forcément 1 lorsque `B` est faux et forcément 0 lorsque `A` est faux ce qui contredit le principe de non pertinence des preuves. De plus, cette fonction serait extraite en une valeur constante `or_to_nat`. La correspondance entre le code extrait et la fonction *Coq* initiale serait rompue.

Choix de types inductifs

Il existe trois variantes de quantifications existentielles dans *Coq*, suivant la sorte donnée au type et aux composantes. Nous avons décrit ces variantes dans la section 10.1.1. La différence de traitement entre les types de sorte `Prop` et les types de sorte `Set` dans l'extraction apporte une justification pratique importante à ces variantes. D'un point de vue logique, les trois expressions suivantes sont analogues :

$$\texttt{\{m : A \& \{ n : B | P\}\}} \tag{11.1}$$

$$\texttt{\{m : A | } \exists \texttt{ n : B | P\}} \tag{11.2}$$

$$\exists \texttt{ m : A | } \exists \texttt{ n : B | P} \tag{11.3}$$

Mais une fonction *Coq* calculant une valeur de la forme 11.1 sera extraite en une fonction calculant deux valeurs `m` et `n` satisfaisant la propriété `P`, tandis qu'une fonction calculant une valeur de la forme 11.2 ne calcule qu'une valeur `m`, telle qu'il existe un `n` tel que la propriété `P` soit satisfaite. Enfin, une fonction de *Coq* calculant une valeur de la forme 11.3 est la preuve d'un théorème et sera tout simplement pas extraite vers *ML*.

D'un point de vue logique, les fonctions retournant une valeur dans un type `or` ou dans un type `sumbool` sont également analogues, mais les premières devront être préférées lorsque l'on ne veut pas voir les calculs qu'elles contiennent apparaître dans le type extrait. Une méthode simple pour obtenir une fonction de la première forme lorsque l'on dispose déjà d'une fonction de la deuxième forme est d'utiliser la fonction de « dégradation » suivante :

```
Definition sumbool_to_or (A B:Prop)(v:{A}+{B}) : A∨B :=
  match v with
  | left Ha ⇒ or_introl B Ha
  | right Hb ⇒ or_intror A Hb
  end.
```

Nous avons appelé cette fonction une fonction de dégradation parce que le trajet inverse, de `or` vers `sumbool` est impossible, pour la raison que nous avons exposée dans la section précédente.

11.1.3 Production effective de code *OCAML*

La commande la plus simple pour produire un fichier *OCAML* contenant les fonctions extraites a la forme suivante :

`Extraction "`*file.ml*`"` f_1 `...`f_n`.`

où f_1, ..., f_n sont les fonctions que l'on veut extraire.

Par exemple notre fonction `log` peut être extraite par la commande suivante :

```
Extraction "log.ml" log2.
```

Si nous voulons mettre en œuvre cette fonction, il est utile de se munir de fonctions de conversion entre les nombres de type `int` en *OCAML* et la représentation de type `nat`. On pourra donc tester la fonction, en la combinant avec ces fonctions de conversion, par exemple de la façon suivante :

```
let rec int_to_nat = function 0 -> O | n -> S(int_to_nat (n-1))

let rec nat_to_int = function O -> 0 | S n -> (nat_to_int n)+1

let e_log n = nat_to_int (log (int_to_nat n))
```

L'extraction de programmes fonctionnels est particulièrement intéressante pour des programmes qui effectuent des calculs symboliques. Par exemple, les travaux de L. Théry sur l'algorithme de Buchberger [84] ont permis d'extraire un programme certifié dont l'efficacité était comparable à la version (non certifiée) utilisée dans le système de calcul formel Maple.

En revanche, cette fonctionnalité est moins adaptée pour des programmes qui effectuent du calcul numérique, parce que les nombres sont représentés de manière symbolique dans les programmes extraits, au lieu de bénéficier des capacités arithmétiques des processeurs.

11.2 ** Description de programmes impératifs

La vérification de programmes impératifs peut être réalisée à l'aide d'un outil développé indépendamment de *Coq*, appelé *Why* [42]. Nous commençons par décrire l'utilisation de cet outil, puis nous montrons comment le travail de cet outil aurait pu être simulé manuellement.

11.2.1 L'outil Why

Cette section présente l'outil *Why* de manière très rapide ; pour plus de détails, on consultera la documentation de l'outil et ses nombreux exemples, tous disponibles sur le site `http://why.lri.fr/`.

L'outil *Why* se présente comme un compilateur, prenant en entrée un programme impératif annoté, écrit en syntaxe ML ou C, et produisant en sortie un ensemble de propriétés exprimant la correction et la terminaison de ce programme. Ces propriétés peuvent être produites dans la syntaxe de plusieurs systèmes de preuve, dont *Coq*, et leur démonstration est à la charge de l'utilisateur. Dans cette présentation, nous nous limitons en entrée à la preuve d'un programme ML et en sortie à l'utilisation de *Coq*.

Outre les fonctions définissant le programme proprement dit, le fichier source passé à l'outil *Why* peut contenir un certain nombre de déclarations. On peut ainsi déclarer l'existence d'une constante entière `l` avec la syntaxe

```
external l:int
```

La commande `external` signifie que la valeur en question existe dans le domaine logique servant de modèle aux programmes et à leurs valeurs (ici *Coq* mais ce pourrait être un autre système). On note l'utilisation du type `int`, propre à *Why*, même si celui-ci se trouve être interprété par le type `Z` dans *Coq*.

Une autre déclaration, `parameter`, permet à l'utilisateur de spécifier des paramètres de sa preuve formelle. Ainsi les déclarations

```
parameter a:int array
parameter x,y:int ref
```

introduisent un tableau d'entiers `a` et deux références entières `x` et `y`. À la différence de la déclaration `external`, ces valeurs n'existent pas dans *Coq* ; la preuve formelle va être menée quels que soient `a`, `x` et `y`.

Ensuite le programmeur peut déclarer des fonctions et procédures, sans avoir à en donner le code. La spécification d'une fonction est donnée par un triplet de Hoare (pré- et post-condition) et ses effets de bord doivent être mentionnés. Ainsi une fonction `swap` échangeant deux éléments du tableau `a` peut être ainsi spécifiée :

```
parameter swap:
  i:int -> j:int ->
    { array_length(a) = l }
    unit
```

```
  writes a
  { array_length(a) = l and
    a[i] = a@[j] and a[j] = a@[i] and
    forall k:int.
       0 <= k < l -> k <> i -> k <> j -> a[k]=a@[k] }
```

On remarque que la syntaxe utilisée à l'intérieur des annotations n'est pas celle de *Coq* : c'est une syntaxe pour des prédicats du premier ordre propre à l'outil *Why*, qui sera traduite plus tard vers *Coq* lorsque les obligations de preuve seront produites. L'accès à l'élément `i` du tableau `a` se note `a[i]`. Dans la post-condition, `a@` dénote la valeur initiale de `a`, c'est-à-dire ici au moment de l'appel à `swap`.

Ensuite l'utilisateur peut écrire le programme annoté en utilisant une syntaxe essentiellement empruntée à *OCAML*. En particulier, les accès en lecture dans les références sont écrits avec un point d'exclamation. En revanche, on utilise une syntaxe à la Pascal pour représenter les opérations sur les tableaux : l'accès dans le tableau `a` à l'index `i` sera écrit `a[i]`.

Toutes les boucles doivent être annotées avec un variant et un invariant. Le variant est une expression d'un type quelconque, accompagnée de la relation bien-fondée qui sera utilisée pour assurer la terminaison (voir section 16.2). Lorsque cette dernière n'est pas spécifiée, on suppose la relation d'ordre usuelle sur les entiers naturels. L'invariant est une formule qui doit être vraie à la première itération et qui doit être maintenue à chaque exécution du corps de la boucle, tant que le test de la boucle est positif.

Par exemple, on pourra écrire la procédure suivante pour la boucle qui trouve l'élément maximal d'un tableau et effectue la permutation de cet élément avec le dernier élément. Ces informations sont données par l'annotation qui débute par le mot-clef `variant`. La boucle est également annotée avec un invariant, qui est une formule qui doit être vraie à la première itération et qui doit être maintenue à chaque exécution du corps de la boucle, tant que le test de la boucle est positif.

```
let pgm_max_end =
  { array_length(a) = l }
  begin
    x := 0;
    y := 1;
    while !y < l do
      { invariant 0 <= y <= l and 0 <= x < l and
                  (forall k:int. 0 <= k < y -> a[k] <= a[x])
        variant l-y }
      if a[!y] > a[!x] then x := !y;
      y := !y + 1
    done;
    (swap !x (l-1))
  end
  { (forall k:int. 0 <= k < l-1 -> k <> x -> a[k] = a@[k]) and
```

```
    a[x] = a@[l-1] and a[l-1] = a@[x] and
    (forall k:int. 0 <= k < l-1 -> a[k] <= a[l-1]) }
```

Dans cette description du programme `pgm_max_end`, la ligne

```
  { array_length(a) = l }
```

décrit la précondition de ce programme. C'est une formule qui doit être satisfaite pour que le programme fonctionne normalement. Les lignes

```
  { (forall k:int. 0 <= k < l-1 -> k <> x -> a[k] = a@[k]) and
    a[x] = a@[l-1] and a[l-1] = a@[x] and
    (forall k:int. 0 <= k < l-1 -> a[k] <= a[l-1]) }
```

décrivent la postcondition. C'est une formule qui décrit les propriétés satisfaites par les données à la fin de l'exécution du programme.

En supposant que les déclarations ci-dessus sont contenues dans un fichier `max.mlw`, les obligations de preuve peuvent être obtenues par la commande

```
why --coq max.mlw
```

et ceci a pour effet de produire, ou de mettre à jour, un fichier *Coq* `max_why.v`. Celui-ci contient six buts. Deux correspondent à la préservation de l'invariant et à la décroissance du variant, pour les deux chemins d'exécution possibles dans le corps de la boucle. Un troisième exprime la validité initiale de l'invariant, et un autre la validité finale de la post-condition. Les deux derniers expriment enfin la légalité des accès `a[!x]` et `a[!y]`. En supposant présente dans *Coq* une hypothèse affirmant que `l` est strictement positif, quelques lignes de preuve suffisent à établir la validité de ces six buts, et donc la correction du programme ci-dessus.

Lorsque l'on met au point un tel programme, il est difficile de prévoir à l'avance le contenu que doit prendre l'invariant. Une méthode de travail productive est de commencer avec un invariant trivial (la proposition `true`) et d'ajouter des éléments dans cet invariant jusqu'à ce que tous les buts engendrés soient prouvables.

L'outil *Why* permet également d'écrire des fonctions récursives effectuant des effets de bord. Comme pour les boucles, il est obligatoire de fournir l'assurance que le programme terminera, sous la forme d'un variant et d'une relation bien fondée. Par exemple, on pourra définir la fonction qui additionne les `x` premiers entiers de la façon suivante :

```
parameter v:int ref

let rec sum (x:int):unit {variant x} =
  { 0 <= x }
  if x = 0 then
    v := 0
  else begin
    (sum (x-1)); v := x + !v
```

```
end
{ 2*v = x*(x+1) }
```

L'outil *Why* engendre quatre buts pour ce programme. Le premier demande de vérifier la post-condition dans le cas où le test effectué dans l'expression conditionnelle est positif. Les trois autres buts correspondent à des vérification à effectuer lorsque le test est négatif. L'un demande de vérifier que dans le cas de l'appel récursif l'argument décroît bien pour la relation `(Zwf ‘0‘)`, un autre de vérifier que la précondition de la fonction `sum` est bien vérifiée dans le cas de l'appel récursif. Le dernier demande de vérifier que la post-condition est bien satisfaite. Par exemple, le premier but a la forme suivante :

$$
\begin{array}{l}
\ldots \\
\textit{Pre1} : `0 \leq x0` \\
\textit{resultb} : \textit{bool} \\
\textit{Test1} : `x0 = 0` \\
\textit{v1} : Z \\
\textit{Post5} : v1 = 0 \\
\texttt{============================} \\
\quad `2*v1 = x0*(x0+1)`
\end{array}
$$

Tous ces buts sont assez aisés à démontrer avec l'aide des tactiques `Subst`, `omega` et `ring`.

11.2.2 *** Les dessous de l'outil Why

L'objet de cette section est de montrer comment le travail effectué par l'outil *Why* peut être réalisé manuellement.

En programmation impérative usuelle, les effets de bord sont concentrés sur l'opération d'affectation. Lorsqu'une affectation a lieu, on comprend habituellement que l'état de la machine change, sans que cet état ait été décrit précisément. Si nous voulons rendre compte de ce genre d'opération, il faudra considérer que toutes les fonctions accédant à des variables ré-affectables sont représentables dans le calcul fonctionnel pur par des fonctions prenant un argument supplémentaire, l'état, et toutes les fonctions faisant des effets de bords devront retourner le nouvel état.

11.2.2.1 Représentation de l'état

On pourra tirer avantage de la possibilité de construire des structures enregistrements « `Record` » pour représenter l'état. Ainsi pour un programme manipulant une variable mutable booléenne `b` et une variable entière `x`, on construira l'état suivant :

```
Record tuple : Set := mk_tuple {b:bool; x:Z}.
```

Nous rappelons que, suite à cette définition, `tuple` est un type inductif et `b` et `x` sont des fonctions de type respectif `tuple`→`bool` et `tuple`→`Z`, Typiquement les fonctions `b` et `x` seront utilisées pour représenter les accès aux variables de même nom dans la représentation impérative. Par exemple si l'on veut considérer l'expression $x+3$ dans le contexte impératif, on construira naturellement l'expression " `x t +3` " dans la traduction fonctionnelle, si t est la variable de type `tuple` représentant l'état courant de la machine.

11.2.2.2 Affectation

L'affectation exprime explicitement le changement de l'état sur l'une des variables. D'un point de vue fonctionnel, il s'agit de prendre un état connu et de retourner un nouvel état dont la valeur est changée seulement pour une variable.

Par exemple, si e est une expression sans effet de bord et que l'on veut représenter l'affectation `x` :=e, alors on va construire une fonction prenant en argument l'état en entrée. Si e' est l'expression qui représente la valeur de e dans cet état, alors on représentera l'affectation par la formule fonctionnelle suivante :

NEW: attention passage V8 : regarder aussi les formules pas en altt : Pierre, 07/10/2003

```
fun t:tuple ⇒ mk_tuple t.(b)(e' t).
```

NEW: cette inclusion à vérifier automatiquement (voir VA)

11.2.2.3 Calcul en séquence

Lorsque l'on effectue des calculs en séquence, par exemple $I_1; I_2$, il faut se souvenir que l'instruction I_2 travaille dans l'état retourné par l'instruction I_1. Si les deux instructions I_1 et I_2 sont représentées dans le cadre fonctionnel par des fonctions f_1 et f_2, la fonction représentant la séquence des deux instructions aura la forme suivante :

```
fun t:tuple ⇒ f2 (f1 t).
```

Observons quelques exemples. La séquence d'instructions

```
b := false;
x := 1;
```

Sera représentée naïvement par la fonction suivante :

```
fun t:tuple ⇒
  (fun t':tuple ⇒ mk_tuple (b t') 1)
    ((fun t'':tuple ⇒ mk_tuple false (x t'')) t).
```

Si l'on réduit cette expression selon les règles de réduction du Calcul des Constructions inductives on peut obtenir la valeur suivante :

```
fun t:tuple ⇒ mk_tuple false 1
```

On pourra utiliser la commande `Eval Compute in ...` pour vérifier cette réduction.

11.2.2.4 Instructions conditionnelles

Pour les instructions conditionnelles, nous allons pour le moment considérer des expressions de test à valeur booléenne sans effet de bord. Lorsqu'un programme contient l'instruction

```
if e then I1 else I2
```

l'expression `e` doit être évaluée dans l'état initial de la commande, puis l'une des branches doit être évaluée dans le même état. Si e' est l'expression fonctionnelle représentant le calcul de l'expression `e` et f_1 et f_2 sont les fonctions représentant les instructions `I1` et `I2` on pourra représenter l'instruction conditionnelle complète par la fonction suivante :

`fun t:tuple` $\Rightarrow$ `if` e' `then` f_1 `t else` f_2 `t.`

En pratique, il sera souvent nécessaire de faire des démonstrations sur les expressions obtenues et il sera préférable d'utiliser la technique de renforcement minimal de spécification décrite en section 10.2.6 page 292, en faisant intervenir une fonction e'' de type " $\forall$`t:tuple,{`e' `t = true}+{`e' `t = false}` ", et en construisant l'expression suivante :

```
fun t:tuple ⇒
  match e'' t with left h ⇒ f1 t | right h' ⇒ f2 t end.
```

Les hypothèses `h` et `h'` pourront être utilisées dans les preuves ou pour l'encodage des conditions de terminaison pour les boucles.

11.2.2.5 Boucles

Dans le contexte générale de la programmation interactive, les boucles permettent d'avoir des calculs qui ne terminent pas. Mais seulement les programmes qui terminent peuvent être modélisés en *Coq* et nous ne représenterons donc que des boucles qui terminent. Ceci se fait en exhibant une propriété qui assure la terminaison en reposant sur la notions de relation bien-fondée (étudiée plus en détail en section 16.2). Par exemple, la boucle suivante termine pour toute valeur initiale de la variable entière `x`, parce qu'elle termine si la valeur initiale est négative, et décroit strictement à chaque itération sinon.

```
while x > 0 do
  x := !x - 1
done
```

Nous utiliserons la fonction `Zgt_bool` pour représenter la fonction de test avec un théorème compagnon pour exprimer ses propriétés :

```
Check Zgt_bool.
```
Zgt_bool : Z→Z→bool

```
Check Zgt_cases.
```
Zgt_cases : $\forall$ *n m:Z, if Zgt_bool n m then* $n > m$ *else* $n \leq m$

En reprenant l'approche utilisée dans la section précédente pour les instructions conditionnelles, nous utilisons la fonction obtenue par un renforcement minimal de spécification, définie comme dans la section 10.2.6 :

```
Definition Zgt_bool' :
  ∀x y:Z, {Zgt_bool x y = true}+{Zgt_bool x y = false}.
intros x0 y0; case (Zgt_bool x0 y0); auto.
Defined.
```

Pour la relation bien fondée, nous utilisons la relation `Zwf` et le théorème `Zwf_well_founded` que nous avons déjà rencontrés en section 9.4.1. De nouvelles relations bien fondées peuvent être obtenues par composition, en utilisant des théorèmes du module `Wellfounded` :

```
Print Zwf.
Zwf = fun c x y:Z ⇒ c ≤ y ∧ x < y : Z→Z→Z→Prop
Argument scopes are [Z_scope Z_scope Z_scope]

Check Zwf_well_founded.
Zwf_well_founded : ∀ c:Z, well_founded (Zwf c)

Check wf_inverse_image.
wf_inverse_image
    : ∀ (A B:Set)(R:B→B→Prop)(f:A→B),
        well_founded R → well_founded (fun x y:A ⇒ R (f x)(f y))
```

Pour notre exemple, nous utilisons le théorème `wf_inverse_image` en instantiant `f` avec la fonction de projection qui retourne la valeur de la variable `x`.

Nous représentons la boucle par une fonction récursive `loop1` de type `tuple → tuple`, mais la récursion bien-fondée requiert que la boucle soit construite avec une fonction auxilliaire de type plus complexe :

```
Definition loop1' :
  ∀t:tuple, (∀t1:tuple, Zwf 0 (x t1)(x t) → tuple)→tuple.
refine
 (fun (t:tuple)
    (loop_again:∀t':tuple, Zwf 0 (x t')(x t) → tuple) ⇒
    match Zgt_bool' (x t) 0 with
    | left h ⇒ loop_again (mk_tuple (b t)((x t)-1)) _
    | right h ⇒ t
    end).
  ...
  t : tuple
  loop_again : ∀ t1:tuple, Zwf 0 (x t1) (x t)→ tuple
  h : Zgt_bool (x t) 0 = true
  ============================
   Zwf 0 (x (mk_tuple (b t)(x t - 1))) (x t)
```

```
generalize (Zgt_cases (x t) 0); rewrite h; intros; simpl.
unfold Zwf; omega.
Defined.

Definition loop1 : tuple→tuple :=
  well_founded_induction
    (wf_inverse_image tuple Z (Zwf 0) x (Zwf_well_founded 0))
    (fun _:tuple ⇒ tuple) loop1'.
```

11.2.2.6 Tableaux

On peut représenter les tableaux par des listes. Une approche alternative, proposée dans le module `Arrays` des bibliothèques de *Coq*, est de représenter les tableaux comme des fonctions de `nat` vers le type des éléments du tableau, mais avec une borne en dehors de laquelle la persistance des données placées dans le tableau n'est pas garantie.

```
Parameter array : Z→Set→Set.

Parameter new : ∀(n:Z)(T:Set), T→array n T.

Parameter access : ∀(n:Z)(T:Set), array n T → Z→T.

Parameter
  store : ∀(n:Z)(T:Set), array n T → Z → T → array n T.

Axiom new_def :
    ∀(n:Z)(T:Set)(v0:T)(i:Z),
      0 ≤ i < n → access (new n v0) i = v0.

Axiom store_def_1 :
    ∀(n:Z)(T:Set)(t:array n T)(v:T)(i:Z),
      0 ≤ i < n → access (store t i v) i = v.

Axiom store_def_2 :
    ∀(n:Z)(T:Set)(t:array n T)(v:T)(i j:Z),
    0 ≤ i < n → 0 ≤ j < n → i ≠ j →
    access (store t i v) j = access t j.
```

Ces axiomes restreignent la façon dont les accès successifs dans un tableau peuvent être réduits à des valeurs connues. Pour une modélisation précise des programmes impératifs, il faut aussi s'imposer d'interdire les accès en lecture comme en écriture en dehors du tableau.

Si `a` est un tableau de longueur `l`, et que nous voulons raisonner sur un programme manipulant `a`, nous allons travailler avec un nouvel état dans lequel existe un champ supplémentaire pour ce tableau. Le type pour cet état pourra être déclaré avec la définition suivante (si l'on suppose qu'il y a aussi deux variables ré-affectables `y` et `z` de type `Z` dans le programme) :

```
Parameter l : Z.
Record tuple':Set := mk_tuple' {a:array l Z; y:Z; z:Z}.
```

et si e' et i' représentent le calcul des expressions `e` et `i` dans l'état t on pourra représenter l'affectation suivante :

```
a[i] := e
```

par l'expression suivante :

```
(fun (t:tuple')(h:0≤i' t<l)⇒
    mk_tuple' (store (a t)(i' t)(e' t))(y t)(z t)) p.
```

Bien sûr, dans cette expression, p doit représenter une preuve de $0 \leq (i'\texttt{t}) < \texttt{l}$. Cette représentation de l'accès dans le tableau contient donc bien une obligation de preuve pour décrire les contraintes de bornes.

11.2.2.7 Exemple d'insertion

Par exemple, le programme suivant réalise l'insertion d'une valeur dans un tableau, en insérant entre l'indice `y` et l'indice `l-1`.

```
while z<l do
  if y > a[z] then
   begin a[z-1] := a[z]; z:= z+1 end
  else
   begin a[z-1] := y; y = l end
done
```

Ce fragment de programme pourra être représenté par l'expression suivante, qui peut sembler complexe, mais a été composée avec l'aide interactive du système :

```
Definition insert_loop : tuple'→tuple'.
refine
 (well_founded_induction
    (wf_inverse_image _ _ _ (fun t:tuple' ⇒ l-(z t))
       (Zwf_well_founded 0))
    (fun _:tuple' ⇒ tuple')
    (fun (t:tuple')
       (loop_again:∀t':tuple',
                    Zwf 0 (l-(z t'))(l-(z t))→tuple') ⇒
       match Z_gt_le_dec l (z t) with
       | left h0 ⇒
         match Z_gt_le_dec (y t)(z t) with
         | left _ ⇒
             (fun (h1:0 ≤ (z t)-1 < l)
```

```
              (h2:0 ≤ (z t) < l) ⇒
              loop_again
                (mk_tuple'
                   (store (a t)((z t)-1)
                          (access (a t)(z t)))
                   (y t)((z t)+1)) _) _ _
        | right _ ⇒
            (fun h1:0 ≤ (z t) < l ⇒
              loop_again
                (mk_tuple' (store (a t)((z t)-1)(y t))
                    (y t) l)
                _) _
        end
      | right h3 ⇒ t
      end)).
```

Cette commande engendre 5 buts. Trois des buts correspondent aux conditions de bornes pour les accès dans le tableau. Ces buts ne sont pas aisément résolus si nous ne savons pas à l'avance que `y` est positif. Ceci montre que l'on peut avoir besoin d'exprimer des invariants de boucles.

11.2.2.8 Invariants de boucle

Nous pouvons exprimer un invariant de boucle en indiquant que la fonction récursive utilisée pour représenter la boucle ne prend pas n'importe quel état, mais un état qui respecte un invariant. Ici, nous pouvons donner comme invariant la propriété `‘0≤y‘`. La fonction définie par récurrence bien fondée n'est plus de type `tuple'→tuple'` mais de type

`∀t:tuple', 0 < (z t) → tuple'.`

La définition prend alors la forme suivante :

```
Definition insert_loop' : ∀t:tuple', 0 < (z t) → tuple'.
refine
 (well_founded_induction
    (wf_inverse_image _ _ _
       (fun t:tuple' ⇒ l-(z t))(Zwf_well_founded 0))
    (fun t:tuple' ⇒ 0 < (z t) → tuple')
    (fun (t:tuple')
       (loop_again:∀t':tuple',
                     Zwf 0 (l-(z t'))(l-(z t)) →
                     0 < (z t') → tuple')(h4:0 < (z t)) ⇒
       match Z_gt_le_dec l (z t) with
       | left h0 ⇒
         match Z_gt_le_dec (y t)(z t) with
         | left _ ⇒
             (fun (h1:0 ≤ (z t)-1 < l)
```

```
                (h2:0 ≤ (z t) < l) ⇒
                loop_again
                  (mk_tuple'
                     (store (a t)((z t)-1)(access (a t)(z t)))
                     (y t)((z t)+1)) _ _) _ _
         | right _ ⇒
             (fun h1:0 ≤ (z t) < l ⇒
                loop_again
                  (mk_tuple' (store (a t)((z t)-1)(y t))(y t) l)
                  _ _) _
         end
       | right _ ⇒ t
       end)).
```

Avec la nouvelle définition, il y a maintenant 7 buts à démontrer. Trois d'entre eux sont toujours les buts limitant les accès dans le tableau, deux autres sont les buts exprimant que la fonction représentant la boucle terminera. Les deux nouveaux buts expriment que l'invariant est bien satisfait. Pour être vraiment utile, il faudrait utiliser un invariant plus expressif que '0<(z t)' pour exprimer que la boucle d'insertion effectue bien le travail attendu. Pour cela, on peut avoir besoin de mentionner la valeur initiale de `(z t)` que l'on appellerait `z0` et la valeur initiale de `(a t)` que l'on appellerait `a0`. Un invariant raisonnable pourrait être le suivant :

```
0 < z t ∧
(z t < l →
 ∀u:Z, z0 ≤ u < (z t) →
   y t > access (a t) u ∧ access (a t) u = access a0 (u+1)) ∧
(z t = l →
 ∃p:Z |
   (∀u:Z, z0 ≤ u < p →
     y t > access (a t) u ∧ access (a t) u = access a0 (u+1)) ∧
   access (a t) p = (y t) ∧
   (∀u:Z, p < u < l → access (a t) u = access a0 u)).
```

Pour ne pas lasser le lecteur, nous n'allons pas écrire ici la nouvelle modélisation de la boucle d'insertion pour cet invariant. Ce type de travail est long, fastidieux et, comme nous l'avons montré en section 11.2.1, automatisable.

Chapitre 12

* Étude de cas

Les principes de l'extraction de programmes ont été présentés dans le chapitre 11. Nous proposons ici une étude de cas simple pour illustrer les subtilités des rapports entre `Prop` et `Set`. Nous verrons comment la connaissance du mécanisme d'extraction permet de faciliter la construction de programmes certifiés sans négliger les considérations d'efficacité.

La notion d'*arbre binaire de recherche* sert de support à notre étude. Nous nous proposons de construire des programmes certifiés pour la recherche, l'insertion et la destruction d'information dans de tels arbres. Le développement complet se trouve dans les contributions d'utilisateurs du système *Coq* [1] ; nous ne donnons ici que les détails se rapportant à l'extraction de programmes.

12.1 Les arbres binaires de recherche

Classiquement, un arbre binaire de recherche est un arbre binaire dont les feuilles ne portent aucune information et dont les sommets internes sont étiquetés — dans notre cas par des entiers (de type `Z`)— ; il est de plus requis que pour tout sommet interne étiqueté par n, le sous-arbre gauche (respectivement : droit) issu de ce sommet ne contienne que des étiquettes strictement inférieures à n (respectivement : strictement supérieures). La figure 12.1 présente un tel arbre.

12.1.1 Les arbres de recherche en *Coq*

Dans ce développement, nous n'associons pas directement un type *Coq* aux arbres binaires de recherche. Nous considérons en premier lieu un type de donnée — celui des arbres binaires étiquetés par des entiers — puis définissons le prédicat « être de recherche » sur ce type. La définition de ce prédicat requiert quelques définitions auxiliaires.

1. Sur le site `http://coq.inria.fr/contribs-eng.html`, cliquer sur `search-trees`

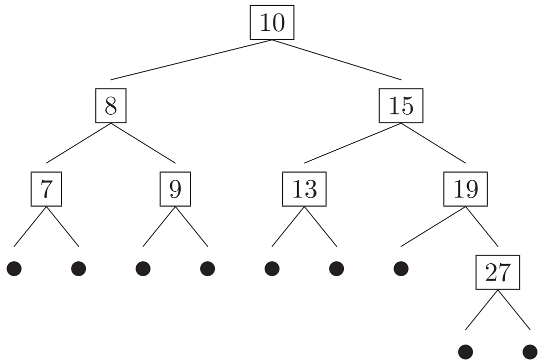


FIGURE 12.1 – Un arbre binaire de recherche

La structure de données

La section 7.3.4 contient une définition inductive du type des arbres binaires dont les sommets internes sont étiquetés par des entiers ; rappelons cette définition :

```
Open Scope Z_scope.

Inductive Z_btree : Set :=
  Z_leaf : Z_btree
| Z_bnode : Z→Z_btree→Z_btree→Z_btree.
```

À titre d'exemple, donnons le terme *Gallina* qui définit l'arbre de la figure 12.1 page 344 :

```
Z_bnode 10
  (Z_bnode 8
    (Z_bnode 7 Z_leaf Z_leaf)
    (Z_bnode 9 Z_leaf Z_leaf))
  (Z_bnode 15
     (Z_bnode 13 Z_leaf Z_leaf)
     (Z_bnode 19 Z_leaf (Z_bnode 27 Z_leaf Z_leaf)))
```

Notion d'occurrence

Nous donnons ici une définition inductive de la proposition “ `occ n t` ” : « l'entier `n` a au moins une occurrence dans l'arbre `t` » :

```
Inductive occ (n:Z) : Z_btree→Prop :=
| occ_root : ∀t1 t2:Z_btree, occ n (Z_bnode n t1 t2)
| occ_l :
    ∀(p:Z)(t1 t2:Z_btree), occ n t1 → occ n (Z_bnode p t1 t2)
```

```
| occ_r :
    ∀(p:Z)(t1 t2:Z_btree), occ n t2 → occ n (Z_bnode p t1 t2).
```

Décidabilité du test d'occurrence : une approche naïve

Nous voulons développer un programme certifié permettant de tester si un entier n possède ou non une occurrence dans un arbre t. En bref, nous souhaitons construire un terme de *Gallina* ayant le type suivant :

```
∀(n:Z)(t:Z_btree), {occ n t}+{~occ n t}
```

Une stratégie simple consiste à utiliser une récurrence sur t, ainsi que la décidabilité de l'égalité sur `Z` (lemme `Z_eq_dec`). Voici cette construction [2] :

```
Definition naive_occ_dec :
  ∀(n:Z)(t:Z_btree), {occ n t}+{~occ n t}.
 induction t.
 right; auto with searchtrees.
 case (Z_eq_dec n z).
 induction 1; left; auto with searchtrees.
 case IHt1; case IHt2; intros; auto with searchtrees.
 right; intro H; elim (occ_inv H); auto with searchtrees.
 tauto.
Defined.
```

À l'aide de la commande " `Extraction naive_occ_dec` ", nous pouvons observer le code extrait correspondant à cette fonction :

```
let rec naive_occ_dec n = function
  Z_leaf ->Right
| Z_bnode (z1, z0, z)→
    (match Z_eq_dec n z1 with
       Left ->Left
     | Right →
         (match naive_occ_dec n z0 with
            Left ->Left
          | Right ->naive_occ_dec n z))
```

Nous remarquons immédiatement le manque d'efficacité de ce programme : dans le cas où l'entier n n'a aucune occurrence dans l'arbre t, le diagnostic (renvoi de la valeur `Right`, assimilée à `false`) est obtenu après un parcours total de l'arbre.

Nous pouvons également remarquer que la spécification de `naive_occ_dec` interdit toute amélioration notable : le second argument de cette fonction est spécifié comme arbre binaire *quelconque*, et rien ne permet d'éviter un parcours

2. La base `searchtrees`, propre à ce développement, contient quelques lemmes techniques destinés à faciliter l'automatisation de preuves ; nous n'en détaillons pas ici le contenu.

total de cet arbre en cas d'absence de l'information cherchée. Ce manque d'efficacité peut être évité si l'on restreint le test d'occurrence à une classe d'arbres binaires possédant des propriétés qui rendent inutile un tel parcours complet.

Caractérisation des arbres de recherche

Nous pouvons définir de façon inductive le prédicat « être un arbre de recherche » :

- Toute feuille est un arbre de recherche,
- Si t_1 et t_2 sont des arbres de recherche, et si n est strictement supérieur à toute étiquette de t_1 et inférieur à toute étiquette de t_2, alors l'arbre de racine n, de fils gauche t_1 et de fils droit t_2 est un arbre de recherche.

La formalisation en *Coq* se fait en trois étapes :

1. Définition d'un prédicat à deux places " `min z t` " : « `z` est inférieur à toute étiquette de `t` »,
2. *idem* pour " `maj z t` " : « `z` est supérieur à toute étiquette de `t` »,
3. Définition inductive de `search_tree: Z_btree → Prop`, utilisant `maj` et `min` comme prédicats auxiliaires.

```
Inductive min (n:Z)(t:Z_btree) : Prop :=
    min_intro : (∀p:Z, occ p t → n < p)→ min n t.

Inductive maj (n:Z)(t:Z_btree) : Prop :=
    maj_intro : (∀p:Z, occ p t → p < n)→ maj n t.

Inductive search_tree : Z_btree→Prop :=
  | leaf_search_tree : search_tree Z_leaf
  | bnode_search_tree :
    ∀(n:Z)(t1 t2:Z_btree),
    search_tree t1 → search_tree t2 → maj n t1 → min n t2 →
    search_tree (Z_bnode n t1 t2).
```

Remarque 12.1 Il peut paraître surprenant que `min` et `maj` soient définies de façon inductive ; le lecteur peut trouver plus naturelle une simple définition de la forme suivante :

```
Definition min (n:Z)(t:Z_btree) : Prop :=
  ∀p:Z, occ p t → n < p.
```

L'auteur de ce développement a préféré le confort d'un type inductif à un seul constructeur, qui lui permet d'utiliser les tactiques `split` en introduction et `case` en élimination, alors que la définition ci-dessus obligerait à contrôler la δ-expansion de `min` par la tactique `unfold` ; d'autre part, une utilisation mal maîtrisée d'`unfold` pourrait provoquer des expansions de `min` non voulues, et perturber la lisibilité des buts. Avec la solution retenue, les constructions et analyses par cas de propositions de la forme " `min` n t " ne sont faites qu'en cas de besoin. Ceci est une méthode générale que nous conseillons d'utiliser fréquemment.

Ce choix entre une définition simple et un type inductif à un seul constructeur se rencontre également en programmation ; considérons par exemple en *OCAML* la définition d'un type pour définir des variables indicées par des entiers (par exemple dans un compilateur). Nous préférons la définition d'un nouveau type :

```
type variable = Mkvar of int
```

à une simple synonymie :

```
type variable = int
```

Exercice 12.1 ** Les prédicats `min` et `maj` auraient pu être directement décrits par une définition inductive récursive sur le type `Z_btree` (sans utiliser explicitement le prédicat `occ`). Reprendre le développement sur les arbres binaires de recherche avec cette nouvelle définition . Vous devrez comparer le confort d'utilisation respectif de ces deux approches. Il faudra veiller à minimiser le travail de modification dû à ce changement. En ce sens, la maintenance de preuves présente les mêmes problèmes et solutions que le génie logiciel.

12.2 Spécification des programmes

Les spécifications des programmes de recherche, d'ajout et de suppression dans les arbres binaires de recherche sont données sous forme de types de sorte `Set` ; ces spécifications utilisent les prédicats `occ` et `search_tree`, ainsi que les constructions `sig` et `sumbool`, présentées en sections 10.1.1 et 10.1.3.

12.2.1 Test d'occurrence

Une fonction de recherche d'un entier p dans un arbre t doit calculer une valeur précisant si p a ou non une occurrence dans t. Le type `sumbool` nous permet d'exprimer la relation entre la valeur calculée (`left` ou `right`) et la certitude que n apparaît ou non dans t. De plus, la construction d'un programme efficace de test d'occurrence peut (doit ?) présupposer que t est un arbre de recherche.

Nous proposons alors la spécification suivante pour la valeur retournée, sous la forme d'un type paramétré par p et t :

```
Definition occ_dec_spec (p:Z)(t:Z_btree) : Set :=
  search_tree t → {occ p t}+{~occ p t}.
```

La spécification du programme à construire est alors la suivante :

```
∀(p:Z)(t:Z_btree), occ_dec_spec p t
```

12.2.2 Programme d'insertion

La spécification d'une fonction pour insérer une occurrence de p dans un arbre t doit préciser une relation entre les occurrences d'entiers dans t et dans l'arbre résultant de cette insertion. D'autre part, cette spécification doit permettre de rejeter toute réalisation qui ne produirait pas un arbre de recherche. Dans le cas contraire, une solution triviale mais sans intérêt, consistant en la construction de l'arbre " `Z_bnode` n t `Z_leaf` " serait conforme à la spécification.

Prédicat associé à l'insertion

Pour les mêmes raisons que `min` et `maj`, nous utilisons un prédicat inductif à un constructeur pour formaliser la relation « t' est obtenu par insertion de n dans t », que nous noterons en *Coq* " `INSERT` n t t' ". Ce prédicat exprime les propriétés suivantes :

- Tout entier apparaissant dans t doit apparaître dans t',
- t' contient au moins une occurrence de n,
- Tout entier apparaissant dans t' apparait également dans t', ou est égal à n, apparaissant dans t,
- t' est un arbre de recherche.

```
Inductive INSERT (n:Z)(t t':Z_btree) : Prop :=
  insert_intro :
  (∀p:Z, occ p t → occ p t')→ occ n t' →
  (∀p:Z, occ p t' → occ p t ∨ n = p)→ search_tree t' →
  INSERT n t t'.
```

Spécification de la fonction d'insertion

Le programme certifié d'insertion doit associer à tout entier n et tout arbre de recherche t un arbre t', accompagné d'une preuve de la proposition " `INSERT` n t t' ". Nous pouvons alors construire la spécification (dépendante) suivante :

```
Definition insert_spec (n:Z)(t:Z_btree) : Set :=
 search_tree t → {t':Z_btree | INSERT n t t'}.
```

La spécification du programme à construire est alors la suivante :

```
∀(n:Z)(t:Z_btree), insert_spec n t
```

12.2.3 Programme de destruction

Pour la destruction d'une occurrence dans un arbre, la démarche est tout à fait similaire à celle de l'insertion : nous donnons sans plus de commentaires la définition d'un prédicat `RM` et du type dépendant `rm_spec` :

```
Inductive RM (n:Z)(t t':Z_btree) : Prop :=
  rm_intro :
  ~occ n t' →
  (∀p:Z, occ p t' → occ p t)→
  (∀p:Z, occ p t → occ p t' ∨ n = p)→
  search_tree t' →
  RM n t t'.

Definition rm_spec (n:Z)(t:Z_btree) : Set :=
  search_tree t → {t' : Z_btree | RM n t t'}.
```

Le type du programme certifié à construire sera alors le suivant :

```
∀(n:Z)(t:Z_btree), rm_spec n t
```

Exercice 12.2 ** Que deviennent les spécifications précédentes si l'on choisit de définir un type « arbre binaire de recherche » de sorte `Set` ?

12.3 Lemmes préliminaires

Une fois spécifiés les programmes à construire, il serait illusoire de commencer leur développement sans travail préalable. L'utilisateur qui se lancerait dans une telle démarche serait vite bloqué par l'accumulation de buts à résoudre.

À titre d'exemple, voici un des nombreux sous-buts pouvant apparaître au cours du développement des programmes certifiés opérant sur les arbres de recherche (cet exemple précis est tiré du développement du programme de destruction).

...
n : Z
p : Z
t1 : Z_btree
t2 : Z_btree
t' : Z_btree
H : n < p
H0 : search_tree (Z_bnode n t1 t2)
H1 : RM p t2 t'
H2 : occ p (Z_bnode n t1 t')
D : occ p t1 ∨ occ p t'
============================
~occ p t1

Comme cette situation se présente à plusieurs reprises dans notre développement, il apparaît utile de construire une mini-bibliothèque de lemmes techniques sur les arbres de recherche, qui en expriment les propriétés simples : la situation décrite ci-dessus se résout (parfois automatiquement) si l'on a préalablement prouvé le lemme suivant :

```
Lemma not_left :
 ∀(n:Z)(t1 t2:Z_btree),
   search_tree (Z_bnode n t1 t2)→∀p:Z, p ≥ n → ~occ p t1.
```

Afin de montrer quels types de résultats ont servi à faciliter le reste du développement, nous donnons uniquement les énoncés de tous ces lemmes techniques dans la figure 12.2 page 351.

12.4 Vers la réalisation

Une fois le terrain préparé, il ne reste plus qu'à développer les programmes certifiés ; à la différence du premier essai présenté en 12.1.1, nous ne voulons plus risquer d'obtenir des programmes peu performants en nous laissant trop guider par l'interaction avec *Coq* et les outils d'automatisation. Parmi les outils actuellement disponibles pour guider la construction d'un programme certifié à partir d'une intuition calculatoire, nous utilisons dans ce développement la tactique `refine`, déjà présentée en section 10.2.7.

12.4.1 Réalisation du test d'occurrence

Rappelons que le but est de fournir un terme pour la spécification suivante :

```
Definition occ_dec : ∀(p:Z)(t:Z_btree), occ_dec_spec p t.
```

Nous pouvons évacuer le cas le plus simple, où l'arbre est réduit à une feuille ; nous savons que l'entier p ne peut apparaître dans `Z_leaf`, et le constructeur `right` du type `sumbool` est alors approprié.

Dans le cas général d'un arbre de la forme “ `Z_bnode` n t_1 t_2 ”, il nous faut comparer n et p afin de chercher l'occurrence de p soit dans la racine n, soit dans t_1, soit dans t_2. La décidabilité de l'ordre total sur `Z` s'exprime en *Coq* à l'aide des deux théorèmes suivants, pris dans la bibliothèque `ZArith` :

```
Z_le_gt_dec : ∀x y:Z, {x ≤ y}+{x > y}

Z_le_lt_eq_dec : ∀x y:Z, x ≤ y → {x < y}+{x = y}
```

Ces deux théorèmes, appliqués à n et p, permettent de distinguer les trois cas suivants :

- si $p < n$, le résultat d'un appel récursif “ `occ_dec` p t_1 ” détermine directement le résultat de l'appel principal :
  - si “ `occ_dec` p t_1 ” se réduit en “ `(left _ _` π ”, où π est une preuve de “ `occ` p t_1 ”, alors on retourne “ `left _ _` π' ”, où π' est un terme de preuve de “ `occ` p `(Z_bnode` n t_1 t_2`)` ”,
  - si “ `occ_dec` p t_1 ” se réduit en “ `right _ _` π ”, où π est une preuve de “ $\sim$ `occ` p t_1 ”, alors on retourne “ `right _ _` π' ”, où π' est un terme de preuve de “ $\sim$ `occ` p `(Z_bnode` n t_1 t_2`)` ”. La construction de π' à partir de π est une application de `go_left`.

```
Lemma min_leaf : ∀z:Z, min z Z_leaf.

Lemma maj_leaf : ∀z:Z, maj z Z_leaf.

Lemma maj_not_occ : ∀(z:Z)(t:Z_btree), maj z t → ~occ z t.

Lemma min_not_occ : ∀(z:Z)(t:Z_btree), min z t → ~occ z t.

Section search_tree_basic_properties.
  Variable n : Z.
  Variables t1 t2 : Z_btree.
  Hypothesis se : search_tree (Z_bnode n t1 t2).

  Lemma search_tree_l : search_tree t1.

  Lemma search_tree_r : search_tree t2.

  Lemma maj_l : maj n t1.

  Lemma min_r : min n t2.

  Lemma not_right : ∀p:Z, p ≤ n → ~occ p t2.

  Lemma not_left : ∀p:Z, p ≥ n → ~occ p t1.

  Lemma go_left :
   ∀p:Z, occ p (Z_bnode n t1 t2)→ p < n → occ p t1.

  Lemma go_right :
   ∀p:Z, occ p (Z_bnode n t1 t2)→ p > n → occ p t2.

End search_tree_basic_properties.

Hint Resolve go_left  go_right  not_left not_right
  search_tree_l search_tree_r maj_l min_r : searchtrees.
```

FIGURE 12.2 – Lemmes techniques sur les arbres de recherche

– si $p = n$, on retourne “ `left _ _` π ”, où π est une preuve de la proposition “ `occ` n `(Z_bnode` n t_1 t_2`)` ”
– si $p > n$, on applique une démarche symétrique au cas $p < n$.

En utilisant `refine`, nous donnons à *Coq* un terme dont les composants logiques (auxquels, rappelons-le, nous sommes profondément indifférents) sont remplacés par le symbole ‘`_`’. Le texte du développement de `occ_dec` montre que ces indéterminations sont pour la plupart levées par `refine` ou les lemmes techniques placées dans la base `searchtrees` :

```
Definition occ_dec : ∀(p:Z)(t:Z_btree), occ_dec_spec p t.
 refine
  (fix occ_dec (p:Z)(t:Z_btree){struct t} : occ_dec_spec p t :=
    match t as x return occ_dec_spec p x with
    | Z_leaf ⇒ fun h ⇒ right _ _
    | Z_bnode n t1 t2 ⇒
      fun h ⇒
        match Z_le_gt_dec p n with
        | left h1 ⇒
          match Z_le_lt_eq_dec p n h1 with
          | left h'1 ⇒
            match occ_dec p t1 _ with
            | left h''1 ⇒ left _ _
            | right h''2 ⇒ right _ _
            end
          | right h'2 ⇒ left _ _
          end
        | right h2 ⇒
          match occ_dec p t2 _ with
          | left h''1 ⇒ left _ _
          | right h''2 ⇒ right _ _
          end
        end
    end); eauto with searchtrees.
 rewrite h'2; auto with searchtrees.
Defined.
```

La commande “ `Extraction occ_dec` ” nous permet de retrouver le contenu algorithmique de notre développement en syntaxe *OCAML*. Notons que les constructeurs `Left` et `Right` doivent être assimilés respectivement à `true` et à `false`.

```
let rec occ_dec p = function
| Z_leaf -> Right
| Z_bnode (n, t1, t2) ->
  (match z_le_gt_dec p n with
   | Left ->
     (match z_le_lt_eq_dec p n with
```

```
        Left -> occ_dec p t1
      | Right -> Left)
   | Right -> occ_dec p t2)
```

Le programme obtenu par extraction effectue donc un parcours d'une seule branche de l'arbre binaire de recherche, dirigé par des comparaisons entre le nombre cherché et les racines successivement rencontrées. Avec des notations plus classiques, nous aurions obtenu le programme *OCAML* ci-dessous :

```
let rec occ_dec p t =
 match t with
  Z_leaf -> false
| Z_bnode(n,t1,t2) ->
    if p <= n
    then  if p < n then occ_dec p t1 else true
    else occ_dec p t2
```

12.4.2 Insertion

La démarche pour l'insertion d'un entier dans un arbre de recherche est similaire à celle pour le test d'occurrence ; aussi ne montrons-nous que les différences notables avec l'étude précédente.

Analyse *à la Prolog*

La programmation de l'insertion dans un arbre de recherche est plus complexe que le test d'occurrence ; il s'agit en effet de construire un nouvel arbre, de s'assurer qu'il est bien un arbre de recherche, alors que la fonction `occ_dec` ne retournait qu'un booléen. Pour pallier cette difficulté supplémentaire, nous avons choisi de commencer le développement du programme certifié d'insertion par la preuve d'une suite de lemmes sur le prédicat `INSERT`, qui s'apparente fortement au paquet de clauses qu'écrirait un programmeur *Prolog* pour définir ce prédicat. Nous donnons en figure 12.3 page 359 ces principaux lemmes (sans leur preuve).

Construction par `refine`

Le placement de ces quatre lemmes dans la base `searchtrees` facilite remarquablement la construction par `refine` du programme certifié `insert`, présentée figure 12.4.

Remarquons que les types des trois arguments de la fonction `insert` appartiennent aux sortes `Set` et `Prop`. Le résultat retourné par `insert` est un couple dont une composante est dans la sorte `Set` et une composante est dans la sorte `Prop`. La distinction entre ces sortes joue un rôle important pour contrôler la quantité de calculs qui sont effectués dans la fonction extraite. Dans la fonction *Coq*, certains calculs sont inclus pour construire la composante preuve du résultat. Au moment de l'extraction, l'argument de la fonction `insert` qui est une preuve disparait, ainsi que la composante preuve du résultat et les calculs effectués pour la construire. Même si les fonction fortement spécifiées semblent contenir plus de calculs que les fonctions faiblement spécifiées, leur correspondant extrait peut être aussi efficace si le concepteur a pris soin de placer seulement les données pertinentes dans la sorte `Set`.

Faut-il un test de décision pour `search_tree` ?

Nous avons déjà remarqué que nombre de nos lemmes et programmes certifiés utilisaient une précondition de la forme “ `search_tree` *t* ” ; or le prédicat `search_tree` a pour type `Z_btree→Prop` et ne doit pas être confondu avec une fonction à valeur booléenne pouvant être utilisée dans des programmes. On pourrait envisager de développer un test de décision pour `search_tree`, qui aurait le type suivant :

```
search_tree_dec : ∀t:Z_btree, {search_tree t}+{∼search_tree t}
```

Nous n'avons pas choisi cette démarche, s'appliquant aux habitants quelconques de `Z_btree`. Il est plus naturel de considérer que les arbres manipulés par nos programmes sont construits à partir d'arbres vides par insertions successives.

Par exemple, nous spécifions la construction d'un arbre de recherche à partir d'une liste d'entiers, de façon que l'arbre obtenu contienne exactement les éléments de la liste passée en argument :

```
Definition list2tree_spec (l:list Z) : Set :=
  {t : Z_btree | search_tree t ∧ (∀p:Z, In p l ↔ occ p t)}.
```

L'interêt d'une telle spécification est de permettre la construction d'arbres de recherche complexes, sans avoir à vérifier à chaque étape que l'insertion d'un entier ne s'opère que dans un arbre de recherche.

Pour le développement de ce programme de conversion de listes en arbres de recherche, nous avons pris une démarche classique en programmation fonctionnelle, consistant en un appel à une fonction récursive terminale auxiliaire.

Nous commençons alors par spécifier cette fonction, qui prend en argument une liste l et un arbre de recherche t pour construire un arbre de recherche t' contenant exactement la réunion des éléments de l et de t :

```
Definition list2tree_aux_spec (l:list Z)(t:Z_btree) :=
  search_tree t →
 {t' : Z_btree | search_tree t' ∧
               (∀p:Z, In p l ∨ occ p t ↔ occ p t')}.
```

Il ne reste plus qu'à proposer — toujours par `refine` —, une réalisation pour `list2tree_aux_spec`, puis `list2tree_spec`. Le terme fourni en argument à `refine` dans le développement de la fonction auxiliaire peut paraître un peu complexe, et nous invitons les lecteurs à l'étudier de près.

```
Definition list2tree_aux :
  ∀(l:list Z)(t:Z_btree), list2tree_aux_spec l t.
refine
 (fix list2tree_aux (l:list Z) :
   ∀t:Z_btree, list2tree_aux_spec l t :=
   fun t ⇒
     match l return list2tree_aux_spec l t with
     | nil ⇒ fun s ⇒ exist _ t _
     | cons p l' ⇒
       fun s ⇒
         match insert p (t:=t) s with
         | exist t' _ ⇒
           match list2tree_aux l' t' _ with
           | exist t'' _ ⇒ exist _ t'' _
           end
         end
     end).
...
Defined.

Definition list2tree : ∀l:list Z, list2tree_spec l.
 refine
  (fun l ⇒ match list2tree_aux l (t:=Z_leaf) _ with
                    | exist t _ ⇒ exist _ t _
                    end).
...
Defined.
```

Programmes extraits

Les programmes extraits pour les fonctions `insert` et `list2tree` sont très simples ; Catherine Parent[71] et Jean-Christophe Filliâtre[41] et Antonia Balaa [7] ont étudié les moyens de rendre ces constructions de programmes certifiés moins détaillées. On peut espérer que des versions futures de *Coq* permettent d'accroître la simplicité des descriptions d'algorithmes.

```
let rec insert n = function
| Z_leaf -> Z_bnode (n, Z_leaf, Z_leaf)
| Z_bnode (p, t1, t2) ->
  (match z_le_gt_dec n p with
   | Left ->
     (match z_le_lt_eq_dec n p with
      | Left -> Z_bnode (p, (insert n t1), t2)
      | Right -> Z_bnode (n, t1, t2))
   | Right -> Z_bnode (p, t1, (insert n t2)))

let rec list2tree_aux l t =
```

```
  match l with
    | Nil -> t
    | Cons (p, l') -> list2tree_aux l' (insert p t)

let list2tree l =
  list2tree_aux l Z_leaf
```

12.4.3 ** Destruction

La destruction d'un item dans un arbre binaire de recherche ne pose pas plus de problème que l'insertion, excepté dans le cas où le sommet à détruire est la racine de l'arbre considéré ; la solution classique pour ôter n d'un arbre de recherche " `Z_bnode` n t_1 t_2 " consiste à enlever de t_1 son plus grand item q et retourner l'arbre " `Z_bnode` q r t_2 ", où r est l'arbre obtenu en enlevant q à t_1. Dans le cas où t_1 est réduit à une feuille, il suffit de renvoyer l'arbre t_2.

Du point de vue de la programmation, nous voyons qu'il faut spécifier et réaliser l'opération auxiliaire « enlever la plus grande étiquette d'un arbre de recherche non vide », puis l'utiliser dans la fonction de destruction principale.

Nous procédons comme pour les sections précédentes, en définissant un prédicat inductif `RMAX` :

```
Inductive RMAX (t t':Z_btree)(n:Z) : Prop :=
  rmax_intro :
  occ n t →
  (∀p:Z, occ p t → p ≤ n)→
  (∀q:Z, occ q t' → occ q t)→
  (∀q:Z, occ q t → occ q t' ∨ n = q)→
  ~occ n t' → search_tree t' → RMAX t t' n.
```

Après avoir prouvé un certain nombre de « lemmes à la *Prolog* » sur `RMAX`, nous construisons une fonction pour enlever le plus grand entier d'un arbre de recherche non vide ; noter l'utilisation du type inductif `sigS` (voir section 10.1.2).

```
Definition rmax_sig (t:Z_btree)(q:Z) :=
  {t':Z_btree | RMAX t t' q}.

Definition rmax_spec (t:Z_btree) :=
  search_tree t → is_bnode t → {q:Z &  rmax_sig t q}.

Definition rmax : ∀t:Z_btree, rmax_spec t.
...
```

Le développement de la fonction de suppression dans un arbre de recherche se poursuit alors comme pour l'insertion. Nous ne donnons pas les détails de cette partie, et le lecteur pourra les trouver dans les contributions de *Coq*. Il est cependant intéressant de terminer par le code obtenu par extraction de `rm` et `rmax`.

```
let rec rmax = function
| Z_leaf -> assert false (* absurd case *)
| Z_bnode (r, t1, t2) ->
  (match t2 with
   | Z_leaf -> ExistS (r, t1)
   | Z_bnode (n', t'1, t'2) ->
     let ExistS (num, r0) = rmax t2 in
       ExistS (num, (Z_bnode (r, t1, r0))))

let rec rm n = function
| Z_leaf -> Z_leaf
| Z_bnode (p, t1, t2) ->
  (match z_le_gt_dec n p with
   | Left ->
     (match z_le_lt_eq_dec n p with
      | Left -> Z_bnode (p, (rm n t1), t2)
      | Right ->
        (match t1 with
         | Z_leaf -> t2
         | Z_bnode (p', t'1, t'2) ->
           let ExistS (q, r) = rmax (Z_bnode (p', t'1, t'2)) in
             Z_bnode (q, r, t2)))
   | Right -> Z_bnode (p, t1, (rm n t2)))
```

On notera l'expression “ `assert false` ” dans le code extrait pour `rmax`, qui correspond à la précondition “ `is_bnode t` ”, en conflit avec la structure primitive récursive de la fonction, qui doit donc examiner le cas $t =$ `Z_leaf`.

12.5 Améliorations possibles

Les arbres de recherche présentés ci-dessus ne permettent de représenter que des ensembles finis d'entiers. Or les seules propriétés de `Z` que nous avons utilisées sont celles de la relation d'ordre $\leq$, et principalement le fait que la comparaison de deux entiers est décidable (théorèmes `Z_le_lt_eq_dec` et `Z_le_gt_dec`). On doit pouvoir généraliser notre approche à tout type possédant ces caractéristiques : non seulement `nat`, mais aussi `nat * nat`, `list bool`, etc.

D'autre part, une utilisation fréquente des arbres de recherche est la représentation de fonctions de domaine fini, ce que ne permet pas notre implémentation jouet.

Le chapitre suivant étudie comment pallier ces insuffisances, dans le cadre du tout nouveau système de modules de *Coq*. Un développement unique permet la représentation de fonctions de domaine fini ; les arbres de recherche sont utilisés dès que le domaine peut être muni d'une relation d'ordre total où la comparaison est décidable.

12.6 Un autre exemple

Le chapitre 12 de l'ouvrage de Jean-François Monin [68] illustre les possibilités de *Coq* sur la spécification et la dérivation d'un programme de recherche en table. Le problème est d'abord spécifié avec concision et en termes très généraux au moyen de « types riches ». À l'aide d'une simple application, il est alors aisé d'en déduire des versions spécialisées à des tableaux ou à des listes, qui se prêtent à l'élaboration d'une solution algorithmique obtenue par extraction.

```
Lemma insert_leaf :
 ∀n:Z, INSERT n Z_leaf (Z_bnode n Z_leaf Z_leaf).

Lemma insert_l :
 ∀(n p:Z)(t1 t'1 t2:Z_btree),
   n < p → search_tree (Z_bnode p t1 t2)→ INSERT n t1 t'1 →
   INSERT n (Z_bnode p t1 t2)(Z_bnode p t'1 t2).

Lemma insert_r :
 ∀(n p:Z)(t1 t2 t'2:Z_btree),
   n > p → search_tree (Z_bnode p t1 t2)→ INSERT n t2 t'2 →
   INSERT n (Z_bnode p t1 t2)(Z_bnode p t1 t'2).

Lemma insert_eq :
 ∀(n:Z)(t1 t2:Z_btree), search_tree (Z_bnode n t1 t2)→
   INSERT n (Z_bnode n t1 t2)(Z_bnode n t1 t2).

Hints Resolve insert_leaf insert_l insert_r insert_eq
  : searchtrees.
```

FIGURE 12.3 – Lemmes à la Prolog pour l'insertion

```
Definition insert : ∀(n:Z)(t:Z_btree), insert_spec n t.
 refine
  (fix insert (n:Z)(t:Z_btree){struct t} : insert_spec n t :=
     match t return insert_spec n t with
     | Z_leaf ⇒ fun s ⇒ exist _ (Z_bnode n Z_leaf Z_leaf) _
     | Z_bnode p t1 t2 ⇒
       fun s ⇒
        match Z_le_gt_dec n p with
        | left h ⇒
          match Z_le_lt_eq_dec n p h with
          | left _ ⇒
            match insert n t1 _ with
            | exist t3 _ ⇒ exist _ (Z_bnode p t3 t2) _
            end
          | right h' ⇒ exist _ (Z_bnode n t1 t2) _
          end
        | right _ ⇒
          match insert n t2 _ with
          | exist t3 _ ⇒ exist _ (Z_bnode p t1 t3) _
          end
        end
     end); eauto with searchtrees.
 rewrite h'; eauto with searchtrees.
Defined.
```

FIGURE 12.4 – Développement du programme d'insertion

Chapitre 13

* Le système de modules

La plupart des langages de programmation actuels permettent de structurer les programmes en unités appelées *modules*. S'ils sont bien conçus, les modules sont réutilisables dans des contextes d'application très variés. Chacun de ces modules possède ses propres structures de données et opérations. Une notion d'*interface* permet de spécifier quelles parties d'implémentation doivent être visibles du reste du programme. Trop de visibilité donnerait accès à des détails d'implémentation, ce qui interdirait toute évolution de cette implémentation ; en effet, si l'auteur d'un module B utilise explicitement un détail d'implémentation du module A, ce détail doit être maintenu dans les évolutions de A, et probablement devenir un frein à des améliorations importantes.

La modularité se traite de diverses façons selon le langage de programmation considéré : utilisations de fichiers `.h` et `.c` en `C`, gestion des droits d'accès, interfaces et paquetages en *Java*, etc. Le livre de L. Paulson sur *ML* [76] présente le système de modules de *Standard ML* ; ce système, par l'utilisation de structures, signatures et modules paramétriques (*foncteurs*), est d'une grande souplesse d'utilisation. Xavier Leroy a proposé une variante de ce système de modules, notamment par l'introduction de *spécifications de types manifestes* permettant la compilation séparée de modules [58, 59]. Ces avancées ont été utilisées dans le langage *Caml Special Light*, puis *OCAML*. Le système de modules de *Coq* en reprend les caractéristiques principales, et est actuellement développé par Jacek Chrząszcz.

Afin de montrer l'utilité d'un système de modules dans le cadre d'un assistant de preuves, nous reprenons un exemple du livre de L. Paulson, développé dans le cadre de la programmation fonctionnelle à base de foncteurs, en montrant l'apport de la partie logique. Nous montrerons ainsi comment construire des structures de données paramétrées et certifiées.

Dans ce chapitre, nous allons travailler avec un objet informatique simple, que nous appellerons *dictionnaire*. *Grosso modo*, un dictionnaire est une structure permettant de retrouver des informations à partir de *clefs*. Spécifier un type abstrait de dictionnaire revient à définir comment les créer, y stocker de l'information, et consulter cette information. Nous appellerons par la suite *entrée* un couple formé par une clef et une donnée associée.

13.1 Signatures

Une signature est une structure syntaxique spécifiant les composantes que toute implémentation d'un module doit posséder. Tout comme les fichiers en-tête de *C* ou les interfaces de *Java*, et bien sûr les signatures de *Standard ML* ou *OCAML*, une signature est composée de déclarations. L'originalité du système de types de *Coq* fait que ces déclarations peuvent porter sur des types ou des fonctions, mais aussi sur des informations logiques, à la différence des langages cités ci-dessus, où ces informations se trouvent réduites à l'état de commentaires.

À titre d'exemple, proposons une signature pour les dictionnaires, contenant en premier lieu les spécifications de champs suivants :

– un type `key` pour représenter les clefs,
– un type `data` pour représenter les valeurs associées aux clefs,
– un type `dict` pour représenter les dictionnaires,
– une constante `empty:dict` pour représenter le dictionnaire sans entrée,
– une fonction `add`, de type “ `key` $\rightarrow$ `data` $\rightarrow$ `dict` $\rightarrow$ `dict` ” : “ `add` k v d ” est le dictionnaire obtenu à partir de d en y ajoutant l'entrée de clé k et de valeur v,
– une fonction `find`, de type “ `key` $\rightarrow$ `dict` $\rightarrow$ `option data` ”, telle que “ `find` k d ” retourne (si possible) la valeur associée à k dans d.

Il est clair que les informations de type sont insuffisantes : rien de nous empêche de proposer le type `nat` pour `dict`, la valeur 0 pour `empty`, une fonction constante retournant toujours 0 pour `add`, et retournant toujours “ `None data` ” pour `find`, alors qu'un dictionnaire doit permettre de retrouver les informations qui y ont été stockées. Cette contrainte peut alors être exprimée par trois propositions (*axiomes*) reliant `empty`, `find` et `add` :

– Le dictionnaire vide ne permet de retrouver aucune valeur :
– Si l'entrée la plus récente dans un dictionnaire est constituée de la clef k et de la valeur d, alors la consultation de ce dictionnaire à la clef k retourne d. Nous remarquons que cette contrainte impose que, dans le cas de plusieurs entrées ayant la même clef, seule la dernière peut être conservée ; une implémentation peut, soit enlever les entrées devenues obsolètes, soit les rendre inaccessibles.
– Si l'on consulte un dictionnaire à une clef différente de celle présente dans l'entrée la plus récente, alors cette consultation renvoie le même résultat qu'avant l'insertion de cette entrée.

La coexistence en *Coq* d'informations de types logique et calculatoire nous autorise à regrouper dans une même structure les déclarations de types, d'opérations, et les propriétés que celles-ci doivent satisfaire.

Une signature s'écrit en *Coq* à l'aide des mots clefs “`Module Type`”[1] suivis du

1. En effet, puisque les signatures sont utilisées comme spécifications de module, elles jouent par rapport aux modules un rôle similaire aux types par rapport aux termes. D'où le nom « type de module » qu'on leur donne en *OCAML* et en *Coq*.

nom de la signature. Les déclarations des champs se font avec la même syntaxe que les déclarations usuelles en « *Coq* de base » : les déclarations de types et d'opérations sont introduites par `Parameter`, celles de propositions par `Axiom` ; tout comme les sections, l'écriture d'une signature se termine par `End`, suivi du nom de la signature.

```
Module Type DICT.

 Parameters key data dict : Set.

 Parameter empty : dict.

 Parameter add : key→data→dict→dict.

 Parameter find : key→dict→ option data.

 Axiom empty_def : ∀k:key, find k empty = None.

 Axiom  success :
     ∀(d:dict)(k:key)(v:data), find k (add k v d) = Some v.

 Axiom diff_key :
     ∀(d:dict)(k k':key)(v:data),
       k ≠ k' → find k (add k' v d) = find k d.
End DICT.
```

Remarque 13.1 Nous n'avons jusqu'ici présenté les signatures que sous la forme d'une suite de déclarations. Nous verrons page 367 comment ajouter *lorsque c'est indispensable* des informations d'implémentation à une signature.

Déclaration de module dans une signature

Une signature peut contenir une déclaration de module. Par exemple, on peut considérer un enrichissement de la théorie des dictionnaires, possédant un constructeur transformant une liste d'entrées en dictionnaire. Nous pouvons construire une signature déclarant une implémentation arbitraire `Dict` de `DICT`, et spécifiant une constante `build` dont le type utilise les champs `key`, `data` et `dict` de `Dict`.

```
Module Type DICT_PLUS.
 Declare Module Dict : DICT.
 Parameter build : list (Dict.key*Dict.data)→ Dict.dict.
End DICT_PLUS.
```

13.2 Modules

Un module est une structure syntaxique regroupant les composantes d'une implémentation. On peut donc le caractériser comme un regroupement de définitions, à prendre au sens large : ces définitions peuvent être des programmes ou des théorèmes, et peuvent être transparentes ou opaques. De plus, un module peut être confronté à une spécification, afin de vérifier la conformité des définitions par rapport aux spécifications. Il est également possible de masquer tout champ l'implémentation qui n'est pas explicité dans la signature.

Les premiers modules que nous étudierons seront construits « à la main », c'est à dire champ par champ ; nous verrons page 368 comment les modules paramétriques permettent un niveau d'abstraction très confortable dans l'écriture de modules.

13.2.1 Étapes de la construction d'un module

Déclaration du module

La construction d'un module démarre par une commande qui admet trois variantes suivant les nécessités de contrôler ou non par une signature, et de masquer ou non l'implémentation. Par la suite, M est le nom du module à construire, et S une signature.

- “ `Module` M `.` ”
  Ouvre la construction d'un module de nom M, sans spécifier de signature associée. Cela sert surtout à regrouper des définitions sous le nom M.
- “ `Module` $M : S$ `.` ”
  Ouvre la construction d'un module de nom M, spécifié par la signature S. Les définitions de M non explicitées (c'est à dire seulement déclarées) dans S sont rendues opaques (voir section 4.4.1, page 86), et les définitions de M présentes dans S sont transparentes. Les champs de M absents de S (ni déclarés ni définis), sont rendus invisibles à l'extérieur du module M. Ceci permet de réaliser les masquages d'implémentation souhaitables.
- “ `Module` $M <: S$ `.` ”
  Ouvre la construction d'un module de nom M, compatible avec la signature S. Contrairement au cas précédent, aucun masquage n'est effectué ; l'opacité ou la transparence des définitions de M sont celles habituelles en *Coq*.

Définition des champs

Après l'ouverture vient l'étape de définition des champs du module en cours de construction. Elle s'apparente à un développement *Coq* habituel, où les noms des théorèmes ou définitions seront les champs du module. Suivant le cas, les commandes `Definition`, `Theorem`, `Lemma`, etc. seront utilisées.

Toute l'interactivité de *Coq* peut bien sûr faciliter le travail au cours de cette étape.

Fermeture d'un module

Un module M se ferme grâce à la commande " `End` M ". Si une signature a été spécifiée au début, *Coq* contrôle que toutes les spécifications de cette signature ont été réalisées. Une erreur peut donc survenir, soit parce qu'un champ de la signature n'est pas défini dans le module, soit parce que la définition de ce champ n'est pas conforme à la réalisation proposée. Si la contrainte par la signature demande un masquage d'implémentation (syntaxe " $M : S$ "), alors celui-ci est réalisé : les définitions de M dont la veleur est donnée dans S sont conservées, celles dont le type est spécifié dans S deviennent opaques au sens décrit dans les autres deviennent invisibles.

13.2.2 Exemple : la notion de clef

Nous illustrons la construction interactive de modules sur un exemple simple, qui sera utilisé dans notre exemple de dictionnaires. Les algorithmes de recherche dans les dictionnaires peuvent faire appel à des comparaisons de clefs. Nous commençons par étudier le cas où cette comparaison se fait par rapport à l'égalité de Leibniz (prédicat `eq` de *Coq*). La signature `KEY` ci-dessous permet d'axiomatiser la notion de « type sur lequel `eq` est décidable » :

```
Module Type KEY.
 Parameter A : Set.
 Parameter eqdec : ∀a b:A, {a = b}+{a ≠ b}.
End KEY.
```

13.2.2.1 Un module masqué

Le module ci-dessous, conforme à la signature `KEY`, est construit interactivement :

```
Open Scope Z_scope.

Module ZKey : KEY.
 Definition A:=Z.

 SearchPattern ({_ = _ :>Z}+{~_}).
```
Z_eq_dec: ∀ x y:Z, {x = y}+{x ≠ y}
...
```
 Definition eqdec := Z_eq_dec.
End ZKey.
```
Module ZKey is defined

Ce module est bien conforme à la signature `KEY` ; le dialogue ci-dessous montre comment accéder aux champs de ce module par la commande `Check`, en utilisant les noms qualifiés <*nom du module*>.<*nom du champ*>.

```
Check ZKey.A.
```

ZKey.A : Set

```
Check ZKey.eqdec.
```
ZKey.eqdec
: $\forall$ a b:ZKey.A, {a = b}+{a $\neq$ b}

En revanche, la commande `Print` ne permet pas de retrouver les définitions `(A := Z)` et `(eqdec := Z_eq_dec)` ; en effet ces définitions sont « abstraites », car absentes de la signature `KEY` : seul leur type importe et peut être utilisé (voir les notions de transparence et d'opacité, page 86.)

```
Print ZKey.A.
```

**** [ZKey.A : Set]*
```
Print ZKey.eqdec.
```

**** [ZKey.eqdec : $\forall$ a b:ZKey.A, { a = b}+{a $\neq$ b}]*

Le message d'erreur suivant montre que l'opacité de `A` pose problème : l'extérieur du module `ZKey` ne peut pas utiliser de clefs représentées sous la forme de nombres entiers.

```
Check (ZKey.eqdec (9*7)(8*8)).
```
...
*Error: The term 9*7 has type Z while it is expected to have type ZKey.A*

En effet, la contrainte "`Zkey: KEY`" a pour effet de masquer la définition de `A` par `Z`, et il devient impossible de rendre compatibles le type `Z` de " `9 * 7` " et " `8 * 8` " avec le type abstrait `ZKey.A`.

13.2.2.2 Contrôle sans masquage

NEW: J'utilise des espaces négatifs à cause de texttt ; probleme avec la version de Latex de Yves

Nous pouvons choisir de seulement vérifier la conformité d'une implémentation de `KEY` à base de nombres entiers, sans réaliser de masquage : il suffit alors d'utiliser l'opérateur '`<:`'.

```
Module ZKey <: KEY.
  Definition A:=Z.
  Definition eqdec := Z_eq_dec.
End ZKey.

Check (ZKey.eqdec (9*7)(8*8)).
```
*: {9*7 = 8*8}+{9*7 $\neq$ 8*8}*

Remarquons que, du coup, la définition de `eqdec` est également rendue transparente.

```
Print ZKey.eqdec.
```
ZKey.eqdec $=$ *Z_ eq_ dec :* $\forall$ *x y:Z,* $\{x = y\}+\{x \neq y\}$

Dans le cas de modules plus complexes, le fait d'exporter une définition de fonction peut se révéler un frein pour des évolutions possibles de ce module. Il suffit qu'un module client utilise cette définition et ses propriétés pour que tout changement rende ce module erroné. La solution suivante permet d'exporter une partie des définitions en considérant une signature avec spécification de type manifeste.

13.2.2.3 Signature enrichie de définitions

Reprenons notre exemple de clefs numériques. Si nous souhaitons utiliser le fait que des clefs sont représentées par des entiers, il faut considérer cette information comme faisant partie d'une spécification de « clef entière » et non comme une implémentation de « clef générique ».

La solution proposée par *Coq* est la même qu'en *OCAML* : on peut, à partir d'une signature, obtenir une autre signature en définissant un de ses champs ; il suffit d'ajouter à la signature une clause de la forme " `with Definition` A`:=` t ". Toute implémentation de cette nouvelle signature doit respecter cette contrainte ; de façon symétrique, l'extérieur d'un module contraint par cette signature enrichie peut exploiter la définition de A par t. Nous retrouvons ainsi les spécifications de types manifestes de X. Leroy. Par exemple, nous pouvons définir une nouvelle signature de clefs où le type est contraint d'être le type des entiers : Nous pouvons maintenant construire notre module avec cette nouvelle signature.

```
Module ZKey : ZKEY.
  Definition A:=Z.
  Definition eqdec := Z_eq_dec.
End ZKey.

Check (ZKey.eqdec (9*7)(8*8)).
```
   : $\{9*7 = 8*8\}+\{9*7 \neq 8*8\}$

```
Print ZKey.eqdec.
```
**** [ZKey.eqdec :* $\forall$ *a b:ZKey.A,* $\{a = b\}+\{a \neq b\}$ *]*

Ajouter une clause de déclaration manifeste peut aussi se faire directement au moment ou l'on construit le module. Dans l'exemple, suivant, nous créons un module de clefs à l'aide d'entiers de Peano :

```
Module NatKey : KEY with Definition A:=nat.
 Definition A:= nat.
 Definition eqdec := eq_nat_dec.
End NatKey.
```

Aussi étrange que cela puisse paraître, il est nécessaire de répéter la définition deux fois. La première fois constitue la définition manifeste d'un champ et la seconde fois la définition du champ de module.

13.2.3 Modules paramétriques (foncteurs)

Nous avons considéré dans nos exemples précédents des implémentations simples de `KEY` ; supposons que nous voulions considérer des clefs plus complexes : listes, couples de clefs, etc. Il serait dommage de devoir construire à la main un module pour les listes de clefs numériques, un autre pour les listes de couples de clefs numériques, ...

La notion de module paramétrique (également appelé *foncteur*) nous permet de considérer des abstractions comme « liste de clefs », « couple de clefs », etc. Par exemple, un tel foncteur exprime comment, à partir d'un module `K` de type `KEY`, construire un module de type “ `KEY with Definition A:= list K.A` ”. Cette construction comprendra un développement d'un programme certifié :

```
eqdec : ∀a b:A, {a = b}+{a ≠ b}
```

Ce développement utilise la décidabilité de l'égalité sur `K.A`, c'est à dire `K.eqdec`, ainsi que les tactiques `discriminate` et `injection` sur les constructeurs `nil` et `cons` (voir pages 179 et 181).

Un module paramétrique M s'écrit aussi simplement qu'un module « de base ». La différence tient en une liste de paramètres `(`M_1`:`T_1`) ...(`M_k`:`T_k`)` où les T_i sont des types de module et les M_i des identificateurs, chacun représentant une implémentation *arbitraire* de T_i. La contrainte éventuelle par une signature se fait de la même façon que pour les modules simples : opérateurs `:`, `<:`, utilisation de signatures enrichies de définitions, ou même absence de contrainte par une signature.

L'instanciation de chaque M_i par un module M'_i compatible avec T_i se fait par l'intermédiaire d'une notation applicative “ $M\ M'_1\ \ldots M'_k$ ”. C'est cette *ressemblance* avec l'application fonctionnelle qui justifie le nom populaire de foncteur, très chic car emprunté au vocabulaire mathématique des catégories. Rappelons cependant que foncteurs et fonctions ne jouent pas sur le même terrain : une fonction s'applique à des termes et son application calcule un terme, tandis qu'un foncteur s'applique à des modules et construit un module.

Un foncteur pour les listes de clefs

Nous montrons comment construire des implémentations de `KEY` où les clefs sont elles-mêmes des listes de clefs plus simples.

Le foncteur suivant prend comme paramètre un module `K:KEY` et construit un module de type “ `KEY with Definition A:=(list K.A)` ” :

```
Require List.

Module LKey (K:KEY) : KEY with Definition A := list K.A.
```

```
 Definition A := list K.A.

 Definition eqdec : ∀a b:A, {a = b}+{a ≠ b}.
  intro a; elim a.
  induction b; [left; auto | right; red; discriminate 1].
  intros a0 k Ha; induction b.
  right; red; discriminate 1.
  case (K.eqdec a0 a1); intro H0.
  case (Ha b); intro H1.
  left; rewrite H1; rewrite H0; auto.
  right; red; injection 1.
  intro H3; case (H1 H3).
  right; red; injection 1.
  intros H3 H4; case (H0 H4).
 Defined.
End LKey.
```

Si l'on veut obtenir une implémentation de clefs à partir de listes d'entiers, il suffit d'appliquer le foncteur `LKey` au module `ZKey` ; en itérant cette construction, nous pouvons également obtenir une implémentation à base de listes de listes d'entiers.

```
Module LZKey := LKey ZKey.
```
Module LZKey is defined

```
Check (LZKey.eqdec (cons 7 nil)(cons (3+4) nil)).
```
   : {7::nil = 3+4::nil}+{7::nil ≠ 3+4::nil}

```
Module LLZKey := LKey LZKey.

Check  (LLZKey.eqdec (cons (cons 7 nil) nil)
                     (cons (cons (3+4) nil) nil)).
```
...

Couples de clefs

D'une façon similaire à `LKey`, nous pouvons construire un foncteur permettant de définir un type de clefs à partir de deux modules `K1` et `K2` de type `KEY`.

```
Module PairKey (K1:KEY)(K2:KEY) : KEY with Definition
  A := prod K1.A  K2.A.

 Open Scope type_scope.
 Definition A := K1.A*K2.A.
```

```
 Definition eqdec : ∀a b:A, {a = b}+{a ≠ b}.
     destruct a; destruct b.
    case (K1.eqdec a a1); intro H;
     case (K2.eqdec a0 a2); intro H0;
     [left | right | right | right];
     try (rewrite H; rewrite H0; trivial); red;
     intro H1; injection H1; tauto.
 Defined.
End PairKey.


Module ZZKey := PairKey ZKey ZKey.
```
Module ZZKey is defined

```
Check (ZZKey.eqdec (5, (-8))((2+3), ((-2)*4))).
```
...

Remarque 13.2 Lors de l'application d'un foncteur de paramètre $M : S$ à un module M', il suffit que M' soit conforme à la signature S. Il n'est pas nécessaire que les champs de M soient masqués par S. Par exemple, l'implémentation suivante de clefs comme listes de booléens se construit à partir du module `BoolKey` défini sans contrainte.

```
Module BoolKey.
 Definition A:= bool.
 Definition eqdec : ∀a b:A, {a = b}+{a ≠ b}.
  destruct a; destruct b; auto; right; discriminate.
 Defined.
End BoolKey.

Module BoolKeys : KEY with Definition A := list bool
                := LKey BoolKey.

Check (BoolKeys.eqdec (cons true nil)
                      (cons true (cons false nil))).
```
: *{true::nil = true::false::nil}+*
{true::nil ≠ true::false::nil}

13.3 Une théorie : les relations d'ordre décidables

Nous présentons un exemple un peu plus élaboré que le précédent, dans lequel le système de modules permet de représenter une petite *théorie*. Cette théorie sera utilisée dans le cadre d'algorithmes utilisant une fonction de comparaison associée à un ordre total (*cf* l'interface `Comparable` de *Java*.) Nous verrons comment les implémentations de dictionnaires par listes ordonnées ou arbres de recherches exploitent cette notion.

Nous allons associer une signature `DEC_ORDER` aux domaines munis d'une relation d'ordre total décidable. Les composantes de cette signature seront de quatre natures :

– Un type A (de sorte `Set`) sur lequel est définie la relation d'ordre,
– deux relations binaires *le* ($\leq$) et *lt* ($<$) sur A,
– des axiomes exprimant que *le* est une relation d'ordre et que *lt* est l'ordre strict associé,
– une spécification de programme certifié de comparaison de deux habitants de A.

```
Module Type DEC_ORDER.
 Parameter A : Set.
 Parameter le : A→A→Prop.
 Parameter lt : A→A→Prop.
 Axiom ordered : order A le.
 Axiom lt_le_weak : ∀a b:A, lt a b → le a b.
 Axiom lt_diff : ∀a b:A, lt a b → a ≠ b.
 Axiom le_lt_or_eq : ∀a b:A, le a b → lt a b ∨ a = b.
 Parameter lt_eq_lt_dec :
    ∀a b:A, {lt a b}+{a = b}+{lt b a}.
End DEC_ORDER.
```

Une telle signature, déclarant à la fois un domaine, des relations et leurs propriétés constitue bien ce qu'on appelle une *théorie*. En ce sens, un module compatible avec la représentation d'une théorie par un type de module constitue bien une preuve de cohérence de cette théorie.

Notons que les théories que l'on peut représenter avec le système de modules de *Coq* comportent des informations aussi bien logiques (pour lesquelles on admet la propriété de non-pertinence des preuves) que calculatoires. Il est donc naturel de considérer diverses implémentations d'une même théorie : par exemple, une théorie des tris pourra se voir réalisée à l'aide du tri par sélection, du tri rapide, etc.

13.3.1 Enrichissement d'une théorie par foncteur

La conception d'une signature (type de module) doit prendre en considération le travail à effectuer pour construire un module associé. En effet, chaque axiome dans un type de module pourra engendrer une obligation de preuve, de même pour les spécifications de programme.

Nous avons par exemple exprimé de façon minimale les relations entre `le` et `lt`, sans spécifier que `lt` est un ordre strict (transitif et irréflexif).

Or, la transitivité et l'irréflexivité de l'ordre strict sont des conséquences logiques des axiomes de `DEC_ORDER` ; de même, le programme certifié décidant si $a < b$ ou $a = b$ sous condition que $a \leq b$ peut se construire à partir des champs de `DEC_ORDER`.

Il est alors naturel de considérer une théorie « enrichie » des relations d'ordre décidables. Le passage de la théorie « pauvre » à sa version enrichie se fait alors par un foncteur, qui exprime en quoi les constructions de la théorie enrichie sont dérivées de celle de sa parente pauvre. On n'écrira donc qu'une seule preuve de transitivité de l'ordre strict, à l'intérieur d'un module paramétrique. C'est par application du foncteur à un module approprié de type `DEC_ORDER` que l'on obtiendra un théorème spécifique à l'ordre considéré. Cette considération s'étend bien sûr aux programmes certifiés.

Nous donnons ci-dessous la signature `MORE_DEC_ORDERS`, venant compléter `DEC_ORDER`, ainsi que le foncteur `More_Dec_Orders` permettant de transformer toute implémentation de `DEC_ORDER` en une implémentation du module `MORE_DEC_ORDERS` concernant les mêmes relations d'ordre. *Les parties de preuves supprimées sont laissées en exercice.*

```
Module Type MORE_DEC_ORDERS.
  Parameter A : Set.
  Parameter le : A→A→Prop.
  Parameter lt : A→A→Prop.

  Axiom le_trans : transitive A le.
  Axiom le_refl : reflexive A le.
  Axiom le_antisym : antisymmetric A le.
  Axiom lt_irreflexive : ∀a:A, ~lt a a.
  Axiom lt_trans : transitive A lt.
  Axiom lt_not_le : ∀a b:A, lt a b → ~le b a.
  Axiom le_not_lt : ∀a b:A, le a b → ~lt b a.
  Axiom lt_intro : ∀a b:A, le a b → a ≠ b → lt a b.

  Parameter le_lt_dec : ∀a b:A, {le a b}+{lt b a}.
  Parameter le_lt_eq_dec :
      ∀a b:A, le a b → {lt a b}+{a = b}.
End MORE_DEC_ORDERS.

Module More_Dec_Orders (P:DEC_ORDER) :
                       MORE_DEC_ORDERS
                       with Definition  A := P.A
                       with Definition le := P.le
                       with Definition lt := P.lt.
 Definition A := P.A.
 Definition le := P.le.
 Definition lt := P.lt.

 Theorem le_trans : transitive A le.
 Proof.
  case P.ordered; auto.
 Qed.

 Theorem le_refl : reflexive A le.
```

```
 (* Proof erased *)

 Theorem le_antisym : antisymmetric A le.
  (* Proof erased *)

 Theorem lt_intro : ∀a b:A, le a b → a ≠ b → lt a b.
 Proof.
  intros a b H diff; case (P.le_lt_or_eq a b H); tauto.
 Qed.

 Theorem lt_irreflexive : ∀a:A, ~lt a a.
 Proof.
  intros a H; case (P.lt_diff _ _ H); trivial.
 Qed.

 Theorem lt_not_le : ∀a b:A, lt a b → ~le b a.
  (* Proof erased *)

 Theorem le_not_lt : ∀a b:A, le a b → ~lt b a.
 (* Proof erased *)

 Theorem lt_trans : transitive A lt.
 (* Proof erased *)

 Definition le_lt_dec : ∀a b:A, {le a b}+{lt b a}.
   intros a b; case (P.lt_eq_lt_dec a b).
   intro d; case d; auto.
   left; apply P.lt_le_weak; trivial.
   induction 1; left; apply le_refl.
   right; trivial.
 Defined.

 Definition le_lt_eq_dec : ∀a b:A, le a b → {lt a b}+{a = b}.
  (* Definition erased *)
End More_Dec_Orders.
```

13.3.2 Le produit lexicographique vu comme foncteur

D'une façon similaire au produit cartésien d'espaces de clefs, nous pouvons construire un ordre total décidable par produit lexicographique. Ceci se fait *via* un module paramétré par `D1` et `D2` de type `DEC_ORDER`.

Plusieurs remarques doivent se faire au sujet de cette construction. En premier lieu, nous exportons les définitions des ordres `le` et `lt`, pouvant être utiles à l'extérieur du module. Ceci se fait en renonçant au masquage d'implémentation (par l'opérateur " `<:` "). D'autre part, pour faciliter le développement de ce module, nous utilisons le foncteur `More_Dec_Orders`, qui nous permet d'utiliser par exemple l'irréflexivité de l'ordre strict `D1.lt`.

Voici un extrait de la définition de `Lexico` :

```
Module Lexico (D1:DEC_ORDER)(D2:DEC_ORDER) <:
              DEC_ORDER with Definition A := (D1.A*D2.A)%type.

 Open Scope type_scope.

 Module M1 := More_Dec_Orders D1.
 Module M2 := More_Dec_Orders D2.

 Definition A := D1.A*D2.A.

 Definition le (a b:A) : Prop :=
   let (a1, a2) := a in
   let (b1, b2) := b in D1.lt a1 b1 ∨ a1 = b1 ∧ D2.le a2 b2.
```

Définition de lt, développements et preuves utilisant M1.lt_trans, D1.lt_eq_lt_dec, M2.lt_irreflexive, etc.

```
End Lexico.
```

De même que pour la signature `KEY`, on obtient des relations d'ordre décidables en appliquant des foncteurs à des modules de base. Voici par exemple comment obtenir une relation d'ordre sur $\mathbb{N} \times \mathbb{N}$.

On associe en premier lieu un module à l'ordre naturel sur $\mathbb{N}$; remarquons que nous exportons les définitions de la relation d'ordre et de l'ordre strict associé :

```
Require Import Arith.
Module Nat_Order : DEC_ORDER
                   with Definition A := nat
                   with Definition le := Peano.le
                   with Definition lt := Peano.lt.

  Definition A := nat.
  Definition le := Peano.le.
  Definition lt := Peano.lt.

  Theorem ordered : order A le.
  Proof.
   split; unfold A, le, reflexive, transitive, antisymmetric;
    eauto with arith.
  Qed.

  Theorem lt_le_weak : ∀a b:A, lt a b → le a b.
```

```
  Proof.
   unfold A; exact lt_le_weak.
  Qed.

  Theorem lt_diff : ∀a b:A, lt a b → a ≠ b.
  Proof.
   unfold A, lt, le; intros a b H e.
   rewrite e in H.
   case (lt_irrefl b H).
  Qed.

  Theorem le_lt_or_eq : ∀a b:A, le a b → lt a b ∨ a = b.
  Proof.
   unfold A, le, lt; exact le_lt_or_eq.
  Qed.

  Definition lt_eq_lt_dec : ∀a b:A, {lt a b}+{a = b}+{lt b a} :=
    Compare_dec.lt_eq_lt_dec.

End Nat_Order.
```

L'ordre sur $\mathbb{N} \times \mathbb{N}$ s'obtient par application de `Lexico` :

```
Module NatNat := Lexico Nat_Order Nat_Order.

Module MoreNatNat := More_Dec_Orders NatNat.

Check (fun x y :nat ⇒ MoreNatNat.lt_irreflexive (x,y)).
```

: ∀ x y:nat, ∼MoreNatNat.lt (x, y)(x, y)

Exercice 13.1 ** En suivant l'exemple de `Lexico`, compléter le développement suivant :

```
Module List_Order (D:DEC_ORDER) :
       DEC_ORDER with Definition A:= list D.A.
...
End List_Order.
```

Expérimenter votre foncteur avec des listes d'entiers naturels.

13.4 Développement d'un module : les dictionnaires

Nous montrons dans le reste de ce chapitre comment construire des modules implémentant `DICT`. Tous les outils de base ayant déjà été abordés, seules quelques subtilités seront soulignées.

13.4.1 Implémentations enrichies

De même que pour les relations d'ordre, nous considérons de façon séparée les opérations dérivées sur les dictionnaires, ceci afin d'automatiser leur construction à partir d'une implémentation simple. Dans notre cas, il s'agira de pouvoir ajouter par un simple appel une liste d'entrées, et par conséquent de construire un dictionnaire en donnant une liste d'entrées initiale.

```
Module Dict_Plus (D:DICT) : DICT_PLUS with Module Dict := D.
 Module Dict := D.
 Definition key := D.key.
 Definition data := D.data.
 Definition dict := D.dict.
 Definition add := D.add.
 Definition empty := D.empty.

 Fixpoint
  addlist (l:list (key*data))(d:dict){struct l} : dict :=
   match l with
   | nil ⇒ d
   | cons p l' ⇒ match p with
                 | pair k v ⇒ addlist l' (add k v d)
                 end
   end.

 Definition build (l:list (key*data)) := addlist l empty.

End Dict_Plus.
```

13.4.2 Construction de dictionnaires par application de foncteurs

Les implémentations de DICT seront vues comme des foncteurs, dont les paramètres seront d'une part, le type de clefs utilisées (ordonnées ou non), d'autre part le type des informations associées aux clefs.

Ce dernier type est spécifié par la signature suivante :

```
Module Type DATA.
 Parameter data:Set.
End DATA.
```

13.4.3 Une implémentation triviale

Il ne faut pas négliger le rôle du maquettage dans le développement d'un logiciel. En effet, la construction d'un logiciel par raffinements successifs impose de retarder le plus possible les décisions d'optimisation, en général prises après un profilage. D'autre part, un développement de taille raisonnable se fait en général à plusieurs, de façon asynchrone. Après accord sur la spécification d'un composant, la réalisation immédiate d'une maquette, même dépourvue de toute efficacité, permet aux modules clients de disposer d'une implémentation sans attendre le développement d'une réalisation effective. Dans le cas des programmes certifiés, ceci est d'autant plus vrai que les preuves de correction peuvent demander du temps. Il faut en outre remarquer que, plus une théorie contient d'axiomes, plus il est difficile de se convaincre de sa cohérence. Une réalisation, même inefficace du point de vue calculatoire, est une preuve rassurante de cohérence.

Nous présentons ci-dessous une maquette de dictionnaire, très facile à valider, puisqu'elle est pratiquement une paraphrase des axiomes de `DICT`. Puisque un dictionnaire sert à représenter une fonction partielle des clefs vers les valeurs associées, définissons simplement les dictionnaires comme des fonctions.

```
Module TrivialDict (Key:KEY)(Val:DATA) :
  DICT with Definition  key := Key.A
       with Definition data := Val.data.

 Definition key := Key.A.

 Definition data := Val.data.

 Definition dict := key → option data.

 Definition empty (k:key) := None (A:=data).

 Definition find (k:key)(d:dict) := d k.

 Definition add (k:key)(v:data)(d:dict) : dict :=
   fun k':key ⇒
     match Key.eqdec k' k with
     | left _ ⇒ Some v
     | right _ ⇒ d k'
     end.

 Theorem empty_def : ∀k:key, find k empty = None.
 Proof.
  unfold find, empty; auto.
 Qed.

 Theorem success :
  ∀(d:dict)(k:key)(v:data), find k (add k v d) = Some v.
 Proof.
  unfold find, add; intros d k v.
  case (Key.eqdec k k); simpl; tauto.
```

```
 Qed.

 Theorem diff_key :
  ∀(d:dict)(k k':key)(v:data),
    k ≠ k' → find k (add k' v d) = find k d.
 Proof.
  unfold find, add; intros d k k' v.
  case (Key.eqdec k k'); simpl; tauto.
 Qed.

End TrivialDict.
```

Il est facile de créer par appel de foncteur un dictionnaire simple dont les clefs sont par exemple des listes d'entiers et les informations associées des listes d'entiers naturels :

```
Module Nats <: DATA .
 Definition data := list nat.
End Nats.

Module Dict1 := TrivialDict LZKey Nats.
Module Dict1Plus := Dict_Plus Dict1.

Check (Dict1Plus.build
        (cons ((cons 3%Z (cons (-4)%Z nil)),
               (cons 6%nat nil))
           (cons ((cons 33%Z (cons (-14)%Z nil)),
                  (cons 7%nat nil))
             nil))).
```

...
: Dict1Plus.Dict.dict

Exercice 13.2 *** Proposer une implémentation moins naïve de `DICT` en vous basant sur des *listes d'associations*, c'est à dire des listes de couples (*clef,valeur*).

13.4.4 Une implémentation efficace

Le module paramétrique que nous allons décrire est une adaptation du développement présenté dans le chapitre 12. En effet les arbres binaires de recherche constituent une implémentation raisonnable des dictionnaires. Il est intéressant de comparer le précédent développement avec la version modulaire que nous présentons.

En premier lieu, les arbres de recherche considérés étaient étiquetés par `Z` ; la version que nous proposons utilise n'importe quel module réalisant `DEC_ORDER`. De plus, nous ne considérons pas les arbres de recherche pour eux-mêmes, mais comme une réalisation de `DICT`. Le travail d'adaptation de la version sur `Z` à la version modulaire s'est révélé assez mécanique. Seuls les types de base et les spécifications ont du être modifiés, les preuves de lemmes sur les arbres de recherche n'ont subi que quelques modifications locales.

13.4.4.1 Structure du foncteur

Le module paramétrique associé à la réalisation de dictionnaires par arbres de recherche ressemble beaucoup à ceux que nous avons déjà vus. Nous commençons par importer quelques champs des paramètres `Key`, de signature `DEC_ORDER` et `Val` de signature `DATA` :

```
Module TDict (Key:DEC_ORDER)(Val:DATA) :
  DICT with Definition  key := Key.A
       with Definition data := Val.data.

 Definition key := Key.A.
 Definition data := Val.data.

 Module M := More_Dec_Orders Key.
```

Travail d'adaptation

Les arbres manipulés dans notre implémentation se distinguent de ceux du chapitre 12 par le double étiquetage (*clef,valeur*) des sommets internes. Toutes les définitions de types : `occ`, `min`, `maj`, `search_tree`, ..., prennent en compte cette modification : par exemple le prédicat `occ` a pour type " `data` $\rightarrow$ `key`$\rightarrow$ `btree`$\rightarrow$ `Prop` ", la proposition " `occ` d k t " signifiant : « il existe un sommet de t étiqueté par la clef k et la donnée d ». Nous donnons ci-dessous le type inductif associé aux arbres binaires, en laissant au lecteur le soin de définir les prédicats `occ` et `search_tree`.

```
Inductive btree : Set :=
  | leaf : btree
  | bnode : key→data→btree→btree→btree.
```

Le travail d'adaptation se poursuit en modifiant les notions de recherche et d'insertion. Le type du programme de recherche pour une clef k dans un arbre t utilise `sumor` ; en effet, soit on retourne une valeur d associée à k dans t, accompagnée d'une preuve que l'entrée (k, d) est bien dans t, soit une preuve qu'aucune valeur ne se trouve associée à k.

```
Definition occ_dec_spec (k:key)(t:btree) :=
  search_tree t → {d:data | occ d k t}+{(∀d:data, ~occ d k t)}.
```

La spécification de l'insertion tient compte d'une conséquence de la spécification `DICT` : si on insère à une clef déjà présente dans l'arbre, la nouvelle entrée masque la précédente :

```
Inductive INSERT (k:key)(d:data)(t t':btree) : Prop :=
  insert_intro :
   (∀ (k':key)(d':data), occ d' k' t → k' = k ∨ occ d' k' t')→
   occ d k t' →
   (∀ (k':key)(d':data),
          occ d' k' t' → occ d' k' t ∨ k = k' ∧ d = d')→
   search_tree t' →
   INSERT k d t t'.

Definition insert_spec (k:key)(d:data)(t:btree) : Set :=
  search_tree t → {t':btree | INSERT k d t t'}.
```

Réalisation des champs de DICT

Avant de définir les champs `dict`, `empty`, `find` et `add` de `DICT`, on construit les programmes certifiés suivants en reprenant quasiment les développements faits dans le cas des arbres étiquetés par `Z`.

```
Definition occ_dec : ∀ (k:key)(t:btree), occ_dec_spec k t.
(* Definition erased *)

Definition insert :
  ∀ (k:key)(d:data)(t:btree), insert_spec k d t.
(* Definition erased *)
```

Le champ `dict:Set` sert de type de donnée pour représenter les dictionnaires. La richesse des types de *Coq* nous autorise à prendre pour `dict` le type des arbres de recherche *certifiés*. Nous pouvons alors définir les trois champs `empty`, `find` et `insert` en utilisant les programmes certifiés `occ_dec` et `insert`.

```
Definition dict : Set := sig (A := btree) search_tree.

Definition empty : dict.
  unfold dict; exists leaf.
  left.
Defined.

Definition find (k:key)(d:dict) : option data :=
   let (t, Ht) := d in
   match occ_dec k t Ht with
   | inleft s ⇒ let (v, _) := s in Some v
   | inright _ ⇒ None
   end.

 Definition add : key→data→dict→dict.
  refine
   (fun (k:key)(v:data)(d:dict) ⇒
```

```
      let (t, Ht) := d in
      let (x, Hx) := insert k v t Ht in exist search_tree x _).
  case Hx; auto.
 Defined.
```

Il ne reste plus qu'à remplir les derniers champs de `DICT` (les preuves ont été supprimées dans ce listing) :

```
Theorem empty_def : ∀k:key, find k empty = None.

Theorem success :
 ∀(d:dict)(k:key)(v:data), find k (add k v d) = Some v.


Theorem diff_key :
 ∀(d:dict)(k k':key)(v:data),
   k ≠ k' → find k (add k' v d) = find k d.

End TDict.
```

Le module paramétrique `TDict` s'utilise aussi simplement que `TrivialDict` ; néammoins, ce foncteur ne s'applique qu'à des implémentations de `DEC_ORDER`, construites soit directement, soit par application de foncteurs comme `Lexico` ou `List_Order` (exercice 13.1.)

```
Open Scope nat_scope.

Module Dict2 <: DICT := TDict NatNat Nats.
Module Dict2plus := Dict_Plus Dict2.

Check (Dict2plus.Dict.find (3, 7)).
```
Dict2plus.Dict.find (3, 7)
: Dict2plus.Dict.dict → option Dict2plus.Dict.data

Exercice 13.3 *** En suivant l'exemple de `TDict`, proposer une implémentation des dictionnaires utilisant une liste d'association ordonnée (par rapport aux clefs).

Exercice 13.4 *** Considérer également la notion de destruction d'une entrée dans la formalisation des dictionnaires (types de module et foncteurs.) *Attention au problème des adjonctions multiples pour une même clef !*

Exercice 13.5 *** Sur le modèle des dictionnaires, écrire une signature pour le tri sur une liste dont les éléments appartiennent à un domaine ordonné décidable. On écrira deux programmes certifiés de tri sous forme de deux foncteurs, l'un associé au tri rapide, l'autre au tri par insertion.

13.5 Conclusion

Le système de modules de *Coq* emprunte la plupart de ses traits à la programmation (il s'inspire fortement des systèmes de modules de *Standard ML* et *OCAML*). Son application à la logique permet d'aborder simplement la notion de théorie. Dans le cas de programmes certifiés, l'utilisation de modules paramétriques permet de partager l'effort de preuve. Pierre Letouzey a récemment adapté les techniques d'extraction de programmes au nouveau système de modules.

Au moment où ces lignes sont écrites, ce système en est à ses débuts. Il est certain que nombre de développements réalisés dans des cadres restreints pourront se généraliser à l'aide des modules paramétriques, et devenir de véritables composants certifiés réutilisables et composables.

Chapitre 14

** Objets et preuves infinis

L'une des utilisations les plus fascinantes d'un assistant de preuve est la possibilité de manipuler des objets infinis, ceci dans le cadre essentiellement finitaire d'un ordinateur.

Les techniques de preuve par récurrence nous permettaient déjà de prouver un énoncé pour un nombre infini d'objets : entiers, arbres binaires, etc. Bien sûr, chacun de ces objets, pris individuellement, se construisait en un nombre fini d'étapes, ce qui justifiait l'utilisation de la récurrence. Nous proposons un pas de plus, en présentant des techniques permettant de construire et manipuler des objets infinis. L'intégration de ces techniques dans *Coq* est due à Eduardo Gimenez [45, 46]. L'exemple que nous développons ici est une formalisation de listes finies ou infinies, lesquelles sont particulièrement adaptées à la modélisation de comportements de systèmes réactifs ; en effet dans des domaines tels que les communications, l'énergie ou les transports, l'infinitude des exécutions est la norme.

14.1 Types co-inductifs

Les types que nous allons étudier sont souvent des extensions de types de données classiques : flots (listes finies ou infinies), arbres finis ou infinis, etc. La plupart de nos exemples s'appuieront sur les flots, et des exercices seront consacrés à un type d'arbre binaire pouvant contenir une infinité de sommets. Nous citons pour mémoire le type des listes infinies, défini dans la bibliothèque standard de *Coq*, en invitant le lecteur à en étudier les développements et utilisations.

14.1.1 La commande `CoInductive`

Pour comprendre la notion de type co-inductif, nous pouvons la comparer avec le concept de type inductif. Les termes d'un type inductif sont obtenus par l'usage répété des constructeurs fournis dans la définition. De plus, les termes doivent être construits de telle façon qu'il n'y ait pas de branches infinies. Cette contrainte est exprimée par le principe de récurrence associé au type. Les types co-inductifs sont similaires aux types inductifs, dans le sens où les termes doivent encore être obtenus par l'usage répété des constructeurs. Néanmoins, il n'y a pas de principe de récurrence et les branches des types inductifs peuvent être infinies.

La commande pour définir un nouveau type co-inductif est donc très proche de la commande permettant de définir un nouveau type inductif : nous devons fournir le nom du type, son propre type, ainsi que le nom et le type des constructeurs. La seule différence entre ces deux formes de définition est une différence de mot-clef : les définitions co-inductives sont introduites par le mot-clef `CoInductive`.

14.1.2 Spécificité des types co-inductifs

Le fait que les termes d'un type co-inductif doivent être obtenus à l'aide des constructeurs est assuré par la possibilité de définir des fonctions par filtrage sur les types co-inductifs, comme pour les types inductifs. En revanche, les fonctions récursives ne peuvent pas être définies de la même manière. Il est important de préserver la propriété que toute fonction définissable dans le calcul des constructions représente des calculs qui terminent toujours. Nous ne pouvons donc pas décrire des fonctions qui effectuent le parcours complet des termes dans un type co-inductif. Neanmoins, nous pouvons considérer une classe de fonctions récursives *paresseuses* qui construisent des termes infinis dans les types co-inductifs. Les termes que construisent ces fonctions peuvent être infinis, mais tant que l'on ne cherche à observer qu'une partie finie de ces termes, ces fonctions n'ont besoin de n'effectuer que des calculs finis. Ces fonctions sont qualifiées *co-récursives*.

Une caractéristique importante des fonctions co-récursives est qu'elles permettent de *construire* des valeurs dans un type co-inductif, alors que les fonctions récursives que nous avons étudiées jusqu'à maintenant *consomment* des valeurs dans des types inductifs. Le terme *co-inductif* provient de cette dualité : les types co-inductifs sont les co-domaines de fonctions co-récursives, tandis que les types inductifs sont les domaines de fonctions récursives.

Enfin, nous verrons que l'égalité de Leibniz — le prédicat `eq` — est trop forte pour être utilisée pour la comparaison de valeurs co-inductives. Dans les cas où nous ne pourrons prouver que deux valeurs obtenues par des moyens différents sont identiques, nous devrons nous contenter d'une notion plus faible d'équivalence : deux objets sont « égaux » si leur exploration — à l'infini ! — trouve des composants identiques aux mêmes emplacements. On dit que deux tels objets sont *bisimilaires*.

Les trois prochaines sections décrivent des exemples de types co-inductifs. Le filtrage est décrit dans la section 14.2.2 page 387 et les fonctions co-récursives sont décrites dans la section 14.3.2 page 388.

14.1.3 Listes infinies (*Flots*)

La bibliothèque `Streams` de *Coq* définit la constante `Stream:Set→Set` associée aux suites infinies. Soit A un type de sorte `Set`, le type `Stream` A des listes infinies d'éléments de type A est défini au moyen d'un seul constructeur `Cons` :

```
CoInductive Stream (A:Set): Set :=
 Cons : A → Stream A → Stream A.
```

Nous notons une différence importante avec la définition des listes donnée en 7.4.1. Il n'y a pas de constructeur associé à la liste vide ; si un tel constructeur existait, il rendrait possible la construction de listes finies. Il est aisé de démontrer qu'un type inductif de cette forme devrait être vide, mais nous définissons ici un type co-inductif dont tous les éléments doivent être de la forme “ `Cons` a l ”.

14.1.4 Listes paresseuses

Le type “ `LList` A ” des listes *paresseuses* diffère de “ `Stream` A ” par la présence d'un constructeur de liste vide `LNil`. On peut considérer une liste paresseuse comme le flot de sortie d'un processus dont le comportement peut être fini ou infini. Pour cette raison, nous appelerons *flot* tout habitant de “ `LList` A ”, dans la mesure où ce type est plus général que le type `Stream` fourni par la bibliothèque de *Coq*.

```
Set Implicit Arguments.

CoInductive LList (A:Set) : Set :=
  LNil : LList A
| LCons : A → LList A → LList A.

Implicit Arguments LNil [A].
```

Si l'on adoptait une vision ensembliste des définitions de type, — c'est à dire considérer l'ensemble des habitants d'un type donné —, on dirait que “ `LList` A ” est le plus grand ensemble de termes bâtis à l'aide des constructeurs `LNil` et `LCons`, ce qui inclut les termes finis et infinis. Comme pour les types inductifs, on considère que les constructeurs sont injectifs et que deux constructeurs différents retournent toujours deux résultats différents. On pourra donc utiliser les tactiques `injection` et `discriminate` (voir sections 7.2.4 et 7.2.2).

La plupart des constantes que nous allons définir dans nos exemples commenceront par la lettre '`L`'[1] Ce préfixe permettra de les distinguer de leur équivalent dans les modules `List` et `Stream` de la distribution *Coq*.

1. Ce 'L' est un mnémonique pour l'anglais “lazy” (paresseux) ; en effet de telles structures de données infinies sont souvent qualifiées de « paresseuses ».

14.1.5 Arbres finis ou infinis

Nous définissons ci-dessous le type des arbres binaires finis ou infinis, dont les sommets internes sont étiquetés par des valeurs de type `A`. Nous qualifierons ces arbres de *paresseux*

```
CoInductive LTree (A:Set) : Set :=
  LLeaf : LTree A
| LBin : A → LTree A → LTree A → LTree A.

Implicit Arguments LLeaf [A].
```

Ces arbres permettront de se poser des problèmes plus complexes que dans le cadre des listes. En effet, un arbre de type “ `LTree` A ” peut être soit fini, soit infini, et dans ce dernier cas avoir des branches finies ou bien ne posséder que des branches infinies.

14.2 Technologie des types co-inductifs

Il semble paradoxal de prétendre construire des termes infinis, surtout dans le cadre borné d'une mémoire d'ordinateur. Le premier problème à résoudre est donc un problème de réprésentation. Cette situation est à rapprocher de la simulation de flots dans des langages applicatifs avec appel par valeur, cette simulation se faisant grâce à l'emploi de fonctions anonymes (voir par exemple [20].) En général, une représentation finie d'objets infinis est acceptable si elle permet de fournir un moyen d'accéder à toute composante de cet objet, ceci par un calcul fini. Par exemple, une représentation finie d'un flot l doit nous permettre de déterminer, pour n'importe quel entier naturel n, si l possède bien un n-ième élément, et si oui de déterminer sa valeur. De même, une représentation d'un arbre fini ou infini doit nous permettre de suivre un chemin d'accès quelconque à partir de la racine, et de déterminer s'il conduit à une feuille, un sommet interne, ou bien sort du domaine de l'arbre.

Nous présenterons une extension du Calcul des Constructions Inductives par une nouvelle construction de termes, ainsi que les mécanismes de calcul permettant l'accès aux composantes des structures de données. De même que pour les définitions inductives et les fonctions récursives, quelques contraintes devront être respectées pour assurer la cohérence de l'extension des constructions inductives aux types co-inductifs. De nombreux exemples et exercices permettront au lecteur de se familiariser avec la manipulation d'objets et de preuves infinis.

14.2.1 Construction d'objets finis

Les types co-inductifs sont définis par une énumération de constructeurs. Il est alors possible d'appliquer ceux-ci un nombre fini de fois, à condition toutefois qu'il existe au moins un constructeur non récursif. C'est le cas de `LNil`, ainsi que de `LLeaf`. En revanche, cette possibilité n'existe pas pour le type `Stream`.

L'exemple suivant construit un flot contenant (dans l'ordre) les entiers 1, 2 et 3. On remarquera que, grâce au mode d'arguments implicites, l'argument de type du constructeur `LCons` doit être omis dès que son type peut être inféré à partir des autres arguments.

```
Check (LCons 1 (LCons 2 (LCons 3 LNil))).
```
LCons 1 (LCons 2 (LCons 3 LNil)) : LList nat

Exercice 14.1 * Définir une injection de `(list` A`)` vers `(LList` A`)`. Vous devrez montrer que la fonction que vous proposez est bien injective.

14.2.2 Décomposition selon les constructeurs

Puisque les objets infinis sont représentés de façon finie par un certain codage plus ou moins complexe, il importe de faciliter l'accès à leurs composantes.

Comme pour les types inductifs, nous pouvons exploiter le fait qu'un terme de type co-inductif C est forcément de la forme " $c\ t_1\ \ldots t_n$ ", où c est un constructeur de C. La construction `match` (voir la section 7.1.4 dans le cadre des types inductifs) est alors le moyen standard pour décomposer un terme quelconque de type C.

La figure 14.1 page 415 montre dans le cas des flots comment écrire la proposition « être vide », et définir les fonctions « tête », « queue », et « accès au n-ième élément ». Ci-dessous figure un calcul utilisant cette dernière fonction :

```
Eval compute in (LNth 2 (LCons 4 (LCons 3 (LCons 90 LNil)))).
```
= Some 90 : option nat

Exercice 14.2 * Définir les prédicats et fonctions d'accès suivants pour le type des arbres paresseux :
- `is_LLeaf` : prédicat « être une feuille »,
- `L_root` : retourne l'étiquette de la racine d'un arbre,
- `L_left_son` : retourne le sous-arbre gauche (s'il existe),
- `L_right_son` : retourne le sous-arbre droit (s'il existe),
- `L_subtree` : retourne le sous-arbre déterminé par un chemin à partir de la racine (s'il existe),
- `Ltree_label` : retourne l'étiquette du sommet déterminé par un chemin à partir de la racine (s'il existe),

Dans les deux dernières fonctions, on assimilera la notion de « chemin » à celle de « liste de directions », où le type `direction` est défini comme suit :

```
Inductive direction : Set := d0 (* left *) | d1 (* right *).
```

14.3 Construction d'objets infinis

Nous devons étudier comment représenter de façon finie des structures infinies. Remarquons que nous ne pouvons prétendre représenter *tous* les arbres ou tous les flots. Un simple argument de cardinalité nous enléverait toute illusion en ce sens : avec les listes infinies de booléens, nous pouvons représenter tout l'intervalle de réels $[0, 1]$; par ailleurs, un système d'expressions finies ne peut représenter qu'un ensemble dénombrable de valeurs différentes. Quelques exemples sur les flots nous permettront d'aborder la construction d'objets infinis.

14.3.1 Tentatives infructueuses

Considérons un problème simple : construire le flot de tous les entiers naturels. Il est bien entendu exclu de construire à la main un terme infini de la forme ci-dessous :

```
LCons 0 (LCons 1 (LCons 2 (LCons 3 ...))).
```

Une façon de procéder est de considérer le flot “ `from n` ” de tous les entiers supérieurs ou égaux à `n`. Ce flot devrait vérifier l'égalité suivante pour tout n :

```
from n = LCons n (from (S n))
```

On pourrait être alors tenté d'utiliser une définition récursive par `Fixpoint` :

```
Fixpoint from (n:nat) {struct n} : LList nat :=
   LCons n (from (S n)).
```

Cela n'a aucune chance d'être correct. En effet, la définition ci-dessus n'est pas bien formée, car non basée sur une décroissance de son argument `n`, bien au contraire. À ce titre, elle est refusée par le système *Coq* :

Error: Recursive definition of from is ill-formed.
In environment n : nat,
Recursive call to from has principal argument equal to
S n instead of a subterm of n

14.3.2 La commande `CoFixpoint`

La syntaxe de la commande `CoFixpoint` est proche de celle de `Definition`. Elle autorise cependant des appels récursifs de la fonction définie. Voici la définition par `CoFixpoint` du flot des entiers à partir de n :

```
CoFixpoint from (n:nat) : LList nat := LCons n (from (S n)).

Definition Nats : LList nat := from 0.
```

On remarquera que, bien que l'argument de l'appel récursif de `from` soit “ `S n` ”, aucun bouclage ne se produit lors de l'appel à `from` provoqué par la définition de `Nats`.

Construction anonyme

Il existe une forme anonyme de `CoFixpoint`, appelée `cofix`, d'usage similaire à `fix` ; voici une définition du flot des carrés des entiers à partir de n :

```
Definition Squares_from :=
  let sqr := fun n:nat ⇒ n*n in
  cofix F : nat → LList nat :=
     fun n:nat ⇒ LCons (sqr n)(F (S n)).
```

Cette forme anonyme `cofix` est très importante : c'est l'ajout de cette nouvelle construction aux termes du Calcul des Constructions Inductives qui nous permet la construction d'objets et de preuves infinis.

La commande `CoFixpoint` est plus proche de la commande `Definition` parce que la co-récursion repose sur le fait que le co-domaine de la fonction est un type co-inductif et il n'y a pas de contrainte sur le domaine de la fonction, tandis que la commande `Fixpoint` requiert que l'on indique quel est l'argument principal. Dans la suite nous appellerons *fonctions co-récursives* les fonctions obtenues par `CoFixpoint` et `cofix`. L'utilisation de fonctions co-récursives s'accompagne de restrictions et de règles de calcul, que nous présentons dans les sections suivantes.

Conditions de garde

La commande `CoFixpoint`, ainsi que la construction `cofix`, ne permettent pas d'écrire n'importe quelle définition récursive. En premier lieu, une telle définition doit servir à construire un objet de type co-inductif ; elle doit respecter d'autre part une condition syntaxique sur les appels récursifs, que l'on pourra comparer à celle régissant les définitions par `Fixpoint` ou `fix`.

D'un point de vue général, une définition par `CoFixpoint` est acceptée si tous les appels récursifs (comme " `from (S n)` ") se trouvent encapsulés dans des constructeurs du type co-inductif considéré, c'est à dire que ces appels ne peuvent apparaître que comme arguments d'un ou plusieurs constructeurs de ce type. Cette condition (dite de « garde par les constructeurs ») s'inspire des langages fonctionnels paresseux, dans lesquels les constructeurs n'évaluent pas leurs arguments. Ainsi l'évaluation d'un appel de la forme `(from (O))` ne peut « boucler ».

Dans notre définition de `from`, cette condition de garde est bien respectée ; en effet, elle ne contient qu'un appel récursif : le terme " `from (S n)` ", qui apparaît comme second argument du constructeur `LCons`.

Pour bien comprendre cette condition, il faut considérer comment sont construits et utilisés les objets infinis. Un tel objet est représenté par un terme contenant des appels à `cofix`, et le seul moyen d'accéder aux informations qu'il contient est le filtrage. Par exemple, l'accés à un item d'un flot se fait par la fonction `LNth`, ce qui revient à appliquer un certain nombre de filtrages élémentaires (par `match`). La construction de termes par co-points-fixes doit être telle que ces fonctions de filtrage terminent.

La condition de garde ci-dessus nous assure que, si une expression de co-point-fixe est destructurée, alors le déroulement de cette expression produira au moins un constructeur du type co-inductif, qui permettra l'analyse par cas.

```
Eval simpl in (isEmpty Nats).
   = False : Prop
```

Comment *Coq* a-t-il obtenu ce résultat ? Rappelons que le prédicat `isEmpty` est défini par filtrage (voir page 415). Après une étape de $\delta\beta$-réductions, le terme à simplifier est le suivant :

```
match (cofix from : nat → LList nat :=
        fun n:nat ⇒ LCons n (from (S n))) 0 with
| LNil ⇒ True
| LCons _ _ ⇒ False
end
   : Prop
```

La construction `match` provoque alors une expansion de l'expression de point-fixe, qui produit le constructeur `LCons`, appliqué à `0` et à “ `from 1` ”. Par filtrage, l'expression entière s'évalue en `False`.

Si en revanche nous tentons d'évaluer une expression sans l'encapsuler dans une construction de filtrage, les expressions de co-point-fixes ne sont pas déroulées :

```
Eval simpl in (from 3).
 = from 3 : LList nat

Eval compute in (from 3).
= (cofix from : nat → LList nat :=
       fun n:nat ⇒ LCons n (from (S n))) 3
   : LList nat
```

Les deux exemples ci-dessous montrent comment composer les accès par filtrage à une telle structure. Les interactions entre le filtrage et le déroulement de co-point-fixes s'enchaînent autant de fois que nécessaire.

```
Eval compute in (LHead (LTail (from 3))).
  = Some 4 : option nat

Eval compute in (LNth 19 (from 17)).
  =  Some 36 : option nat
```

14.3.3 Quelques constructions par co-point-fixe

Nous présentons quelques nouveaux exemples de construction à l'aide de `CoFixpoint` ; pour chaque exemple, le lecteur vérifiera que la condition de garde par constructeurs est bien respectée. Ces exemples portent sur les flots, et quelques exercices sont consacrés aux arbres.

Répetition d'un même élément

La fonction `repeat` prend en argument un élément que l'on veut répéter indéfiniment :

```
CoFixpoint repeat (A:Set)(a:A) : LList A := LCons a (repeat a).
```

Concaténation

Nous voulons définir la concaténation de deux flots sur le même type A. La définition suivante utilise une décomposition par filtrage du premier argument de la concaténation.

```
CoFixpoint LAppend (A:Set)(u v:LList A) : LList A :=
  match u with
  | LNil ⇒ v
  | LCons a u' ⇒ LCons a (LAppend u' v)
  end.
```

Notons que nous avons ici un premier exemple de fonction qui *consomme* un flot et en *produit* un autre. Nous pouvons voir intuitivement que, si u et v sont deux flots, alors le constructeur principal de leur concaténation est parfaitement déterminé. En effet, si u est vide, ce constructeur sera le constructeur principal de v, sinon `LCons`.

Les deux exemples ci-dessous montrent comment les fonctions de construction `repeat` et `LAppend` collaborent avec la fonction d'accès `LNth`, qui itère les filtrages. Dans le premier exemple, les 123 appels récursifs à `LNth` provoquent autant de déroulements de " `repeat 33` ". En revanche, dans le second exemple, le premier argument de `LAppend` est un flot (fini) trop court, et nous devons explorer le second argument.

```
Eval compute in (LNth 123 (LAppend (repeat 33) Nats)).
```
= Some 33 : option nat

```
Eval compute in
  (LNth 123 (LAppend (LCons 0 (LCons 1 (LCons 2 LNil))) Nats)).
```
= Some 120 : option nat

Répétition à l'infini d'un flot

Nous pouvons aborder des constructions par co-point-fixe un peu plus complexe que les précédentes Nous souhaitons généraliser `repeat` en considérant la considérant la répétition à l'infini d'un motif donné sous la forme d'un flot. Par exemple, la répétition infinie du flot “ `LCons 0 (LCons 1 LNil)` ” est un flot infini alternant les occurrences de `0` et `1`. Nous conviendrons que la répétition à l'infini d'un flot infini sera ce flot lui-même [2].

Il est clair qu'une définition directe par `CoFixpoint` ne peut être envisagée : en effet, une approche (co-)récursive directe consisterait à exprimer l'itération infinie de “ `LCons` a v ” en fonction de celle de v, ce qui est impossible. Il est très fréquent que ce genre de problème se résolve par une simple généralisation de la spécification à réaliser. C'est encore le cas ici : considérons le calcul de la concaténation d'un flot u avec une répétition infinie du flot v, que nous notons provisoirement uv^ω et montrons comment déterminer la structure (constructeur principal) du résultat :

- si v est vide, alors le résultat est égal à u,
- sinon, posons $v =$ “ `LCons` b v' ” ;
  - si u est vide, le résultat est un flot de tête b, dont la queue est $v'v^\omega$
  - si $u =$ `LCons` a u', nous construisons un flot de tête a et de queue $u'v^\omega$

Il est alors possible de définir par `CoFixpoint` une fonction calculant uv^ω, puis de l'appliquer avec $u = v$ pour résoudre le problème initialement posé.

```
CoFixpoint general_omega (A:Set)(u v:LList A) : LList A :=
  match v with
  | LNil ⇒ u
  | LCons b v' ⇒
      match u with
      | LNil ⇒ LCons b (general_omega v' v)
      | LCons a u' ⇒ LCons a (general_omega u' v)
      end
  end.

Definition omega (A:Set)(u:LList A) : LList A  :=
   general_omega u u.
```

La complexité apparente de ces deux définitions ne doit pas effrayer le lecteur ; nous verrons très rapidement comment en extraire quelques lemmes simples qui nous permettront de travailler facilement.

Exercice 14.3 ** En reprenant l'exemple des arbres binaires paresseux, de l'exercice 14.1.5, construire un arbre contenant tous les entiers strictement positifs.

Exercice 14.4 * Définir une fonction `graft` de type

2. Nous verrons en section 14.7 qu'en toute précision, nous obtiendrons une *copie* du flot d'origine.

$\forall$ A :Set, LTree A $\rightarrow$ LTree A $\rightarrow$ LTree A

telle que l'arbre « `graft` t t' » soit le résultat du remplacement de toute feuille de t par t'.

14.3.4 Exemples de définitions mal formées

Dès qu'une définition ne respecte pas les contraintes de garde, elle est rejetée par le système. Considérons quelques exemples classiques, pris dans le travail de thèse d'Eduardo Gimenez.

Appels récursifs non encapsulés

La définition suivante du « filtre », c'est à dire une fonctionnelle ne gardant d'un flot que les éléments satisfaisant un prédicat booléen p, n'est pas acceptée, car un des deux appels récursifs à `filter` n'est pas encapsulé dans des constructeurs.

```
CoFixpoint filter (A:Set)(p:A→bool)(l:LList A) : LList A :=
 match  l with
    LNil ⇒ LNil
 |  LCons a l' ⇒ if p a then LCons a (filter p l')
                         else (filter p l')
 end.
```

Si *Coq* acceptait une telle définition, l'évaluation du terme suivant bouclerait, ce qui est inacceptable en *Coq* :

```
LHead (filter (fun p:nat ⇒
                 match p with 0 ⇒ true | S n ⇒ false end)
              (from 1))
```

Un autre cas est la définition suivante, où le premier appel à `buggy_repeat` est l'argument de `match`, donc non encapsulé : une évaluation de l'expression (`buggy_repeat` a) bouclerait directement.

```
CoFixpoint buggy_repeat (A:Set)(a:A) : (LList A) :=
   match buggy_repeat a with
     LNil ⇒ LNil
   | LCons b l' ⇒ LCons a (buggy_repeat a)
   end.
```

Appel récursif encapsulé dans un non-constructeur

Dans la définition ci-dessous, l'appel le plus interne à `F` se trouve encapsulé dans un appel à la fonction `F` elle-même, qui n'est pas un constructeur.

```
CoFixpoint F (u:LList nat) : LList nat :=
  match u with
```

```
   LNil ⇒ LNil
 | LCons a v ⇒ match  a with
                 O  ⇒ LNil
               | S b ⇒ LCons  b (F (F v))
               end
 end.
```

Déterminer si une telle définition est une méthode correcte de construction demanderait une analyse trop fine pour être automatisée. Le système *Coq* se restreint à une condition syntaxique aisément vérifiable : la garde par constructeurs, et rejette donc ce type de définition.

Exercice 14.5 * Définir la fonctionnelle :

```
LMap : ∀A B :Set, (A → B) → LList A → LList B
```

telle que (LMap f l) soit le flot des images par f des éléments de l.

Pouvez-vous faire de même pour la fonctionnelle `LMapcan` de type :

```
∀A B :Set, (A → LList B) → LList A → LList B
```

telle que “ `LMapcan f k` ” soit la concaténation (par `LAppend`) des images par `f` des éléments de `l` ? Pourquoi ?

14.4 Techniques de dépliage

Nous nous intéressons aux techniques de preuves sur les fonctions définies par co-point-fixe. Les calculs portant sur des termes potentiellement infinis sont de nature plus complexes que ceux sur les termes de types inductifs, ceci étant dû à l'exigence de terminaison uniforme des réductions. Les deux exemples qui suivent montrent que des techniques usuelles telles la simplification peuvent échouer, même dans des cas d'aspect simple.

Commençons par une simple tentative de calcul de la concaténation de deux flots finis :

```
Eval simpl in
  (LAppend (LCons 1 (LCons 2 LNil))(LCons 3 (LCons 4 LNil))).
 = LAppend (LCons 1 (LCons 2 LNil))(LCons 3 (LCons 4 LNil))
   : LList nat
```

Aucune simplification n'est vraiment effectuée, faute de filtrage permettant le déroulement de la définition de `LAppend` ; le second exemple montre une tentative de preuve par simplification conduisant à l'échec :

```
Theorem LAppend_LCons :
 ∀(A:Set)(a:A)(u v:LList A),
   LAppend (LCons a u) v = LCons a (LAppend u v).
Proof.
```

```
 intros; simpl.
 ...
 ============================
   LAppend (LCons a u) v = LCons a (LAppend u v)
```

Le but n'a pas vraiment changé. C'est un échec au sens où la démonstration ne progresse pas.

14.4.1 Décomposition systématique

Nous avons vu page 391 que la définition d'une fonction comme `LAppend` pouvait se dérouler en présence d'une opération d'accès (c'est à dire de filtrage suivant les constructeurs de `LList`). De façon plus générale, si un terme t a pour type un type co-inductif C, alors il existe un constructeur du type C tel que t est obtenu en appliquant ce constructeur. Cette propriété peut s'exprimer sous la forme d'une égalité entre t et une construction `match` regroupant tous les cas possibles. Il est possible de construire et de prouver un tel *lemme de décomposition* pour tout type inductif ou co-inductif, mais il n'est vraiment utile que pour les types co-inductifs. Nous verrons page 396 comment ces lemmes permettent de raisonner sur les fonctions définies sur les types co-inductifs (cette approche nous a été suggérée par Christine Paulin).

Le cas des flots

Nous pouvons définir sur `LList` A une fonction de décomposition systématique, prenant en argument un flot l quelconque, et renvoyant sa décomposition selon le constructeur `LNil` ou `LCons`.

```
Definition LList_decompose (A:Set)(l:LList A) : LList A :=
  match l with
  | LNil ⇒ LNil
  | LCons a l' ⇒ LCons a l'
  end.
```

Le lemme suivant montre que `LList_decompose` est une identité sur `(LList` A`)` ; en d'autres termes, il exprime que l'analyse par cas selon les deux constructeurs de `LList` est exhaustive ; sa démonstration se fait à l'aide de la tactique `case` :

```
Theorem LList_decomposition_lemma :
   ∀(A:Set)(l:LList A), l = LList_decompose l.
Proof.
 intros A l; case l; trivial.
Qed.
```

Exercice 14.6 * Effectuer la même démarche dans le cadre des arbres binaires paresseux : écrire une fonction de décomposition systématique, et prouver le lemme de décomposition associé.

14.4.2 Simplification et décomposition

D'un point de vue fonctionnel, une fonction comme `LList_decompose` calcule un flot qui sera indistinguable de son argument. Pourquoi alors la définir ? L'intérêt opérationnel d'une telle fonction est que, si son argument est l'application d'une fonction f définie par co-point-fixe, le filtrage par `match` pourra provoquer une expansion de la définition de f :

```
Eval simpl in (repeat 33).
    = repeat 33  : LList nat

Eval simpl in (LList_decompose (repeat 33)).
  = LCons 33 (repeat 33) : LList nat
```

Exercice 14.7 * Définir une fonction `LList_decomp_n` de type

```
∀A:Set, nat → LList A → LList A
```

itérant l'application de `LList_decompose`. Par exemple, on devra obtenir :

```
Eval simpl in (LList_decomp_n 4
                (LAppend (LCons 1 (LCons 2 LNil))
                         (LCons 3 (LCons 4 LNil)))).
 = LCons 1 (LCons 2 (LCons 3 (LCons 4 LNil)))
    : LList nat

Eval simpl in (LList_decomp_n 6 Nats).
  = LCons 0
       (LCons 1
         (LCons 2
           (LCons 3
             (LCons 4 (LCons 5 (from 6))))))
    : LList nat

Eval simpl in
    (LList_decomp_n 5 (omega (LCons 1 (LCons 2 LNil)))).
    = LCons 1
       (LCons 2
         (LCons 1
           (LCons 2
             (LCons 1
               (general_omega (LCons 2 LNil)(LCons 1 (LCons 2 LNil)))))))
    : LList nat
```

Généraliser le lemme de décomposition en utilisant `LList_decomp_n`.

14.4.3 Utilisation des lemmes de décomposition

Afin de prouver des propriétés de la concaténation de flots, nous souhaitons démontrer des égalités pouvant justifier des tactiques de simplification. Prenons par exemple l'égalité “ `LAppend LNil v = v` ” pour tout type `A` et tout flot `v` dans “ `LList A` ”. Nous avons vu sur des exemples similaires que cette égalité ne peut se prouver directement par `simpl`. En revanche, le lemme `LList_decomposition_lemma` nous permet de transformer ce but en l'égalité suivante.

“ `LList_decompose (LAppend LNil v) = v` ”.

Une analyse par cas sur `v` engendre alors deux sous-buts :

– `LList_decompose (LAppend LNil LNil) = LNil`
– `LList_decompose (LAppend LNil LCons a v) = LCons a v`

Dans chacun des cas, des simplifications du type de celles vues plus haut mènent à une égalité triviale.

Voici le script de preuve complet, précédé par une définition de tactique permettant d'en simplifier l'écriture, et qui sera abondamment utilisée dans les exemples et exercices à venir.

```
Ltac LList_unfold term :=
  apply trans_equal with (1 := LList_decomposition_lemma term).

Theorem LAppend_LNil : ∀(A:Set)(v:LList A), LAppend LNil v = v.
Proof.
  intros A v.
  LList_unfold (LAppend LNil v).
  case v; simpl; auto.
 Qed.
```

De la même manière, nous réussissons à prouver le lemme `LAppend_LCons` (se rappeler l'échec page 394.)

```
Theorem LAppend_LCons :
  ∀(A:Set)(a:A)(u v:LList A),
    LAppend (LCons a u) v = LCons a (LAppend u v).
Proof.
  intros A a u v.
  LList_unfold (LAppend (LCons a u) v).
  case v; simpl; auto.
Qed.
```

Ces lemmes fort utiles sont alors mis à disposition de la tactique `autorewrite`, dans une base de tactiques `llists` que nous créons à cette occasion.

```
Hint Rewrite [LAppend_LNil LAppend_LCons] in llists.
```

Exercice 14.8 ** Prouver les lemmes de dépliage suivants :

```
Lemma from_unfold : ∀n:nat, from n = LCons n (from (S n)).

Lemma repeat_unfold :
 ∀(A:Set)(a:A), repeat a = LCons a (repeat a).

Lemma general_omega_LNil : ∀A:Set, omega LNil = LNil (A := A).

Lemma general_omega_LCons :
 ∀(A:Set)(a:A)(u v:LList A),
   general_omega (LCons a u) v = LCons a (general_omega u v).

Lemma general_omega_LNil_LCons :
 ∀(A:Set)(a:A)(u:LList A),
   general_omega LNil (LCons a u) =
   LCons a (general_omega u (LCons a u)).
```

En déduire le lemme ci-dessous :

```
Lemma general_omega_shoots_again : ∀(A:Set)(v:LList A),
   general_omega LNil v = general_omega v v.
```

Remarque 14.1 Nous aurions bien aimé démontrer le lemme suivant :

```
Lemma omega_unfold :
 ∀(A:Set)(u:LList A), omega u = LAppend u (omega u).
```

Il est malheureusement impossible d'appliquer la règle d'introduction de l'égalité `refl_equal`, car cette application exigerait que les termes `u` et “ `omega u` ” soient unifiables. Aucun argument ne nous permet de l'affirmer dans le cas ou `u` est infini. En fait, nous n'arriverons à prouver cette égalité qu'en introduisant une hypothèse de finitude de u, mais au prix d'un effort de raisonnement plus important que dans les exemples ci-dessus. Une autre possibilité, — étudiée en section 14.7 —, consistera en la définition et l'utilisation d'une relation d'équivalence plus faible que l'égalité de *Coq* : deux flots sont équivalents s'ils ont les mêmes éléments à la même place.

Exercice 14.9 ** Prouver des lemmes de dépliage pour la fonction `graft` définie dans l'exercice 14.4.

14.5 Prédicats inductifs sur un type co-inductif

Beaucoup d'outils étudiés dans les chapitres précédents s'adaptent au traitement de termes d'un type co-inductif construits sans co-points-fixes. En particulier nous avons la possibilité de définir des prédicats inductifs.

14.5.1 Le prédicat « être un flot fini »

Comme le type “ `LList` A ” est habité par des flots finis ou infinis, il est intéressant de disposer du prédicat `Finite` permettant de caractériser la finitude. Une flot fini se construisant par une suite finie d'applications des constructeurs `LNil` et `LCons`, il est naturel de proposer une définition inductive :

```
Inductive Finite (A:Set) : LList A → Prop :=
  | Finite_LNil : Finite LNil
  | Finite_LCons :
      ∀(a:A)(l:LList A), Finite l → Finite (LCons a l).

Hint Resolve Finite_LNil Finite_LCons : llists.
```

Les techniques d'application de constructeurs, d'inversion et d'induction s'utilisent sans aucun problème. On remarquera comment les tactiques d'automatisation `auto` et `autorewrite` sont employées dans les preuves qui suivent.

```
Lemma one_two_three :
 Finite (LCons 1 (LCons 2 (LCons 3 LNil))).
Proof.
 auto with llists.
Qed.

Theorem Finite_of_LCons :
 ∀(A:Set)(a:A)(l:LList A), Finite (LCons a l) → Finite l.
Proof.
 intros A a l H; inversion H; assumption.
Qed.


Theorem LAppend_of_Finite :
 ∀(A:Set)(l l':LList A),
   Finite l → Finite l' → Finite (LAppend l l').
Proof.
  induction 1; autorewrite [llists] using auto with llists.
Qed.
```

Exercice 14.10 *** Prouver le théorème suivant, exprimant le caractère itératif de la fonction `omega` ; on notera que la restriction aux flots finis nous permet de résoudre l'impossibilité décrite dans la remarque 14.1, page 398.

```
Theorem omega_of_Finite :
 ∀(A:Set)(u:LList A), Finite u → omega u = LAppend u (omega u).
```

On pourra utiliser les lemmes démontrés dans l'exercice 14.8.

Exercice 14.11 Définir le prédicat sur “ `LTree` A ” « être un arbre fini ».
Montrer l'égalité “ `graft t LLeaf = t` ” pour tout arbre fini `t`.

14.6 Propriétés co-inductives

Depuis le chapitre 4, nous savons que les propositions sont des types et leurs preuves des habitants de ces types. Par conséquent, la définition de types co-inductifs de sorte `Prop` nous servira à définir des prédicats co-inductifs ; les preuves de théorèmes portant sur de tels prédicats seront alors des termes de preuve infinis, que nous pourrons construire à l'aide de `cofix`. Rien, hormis les subtilités de la non-pertinence des preuves, ne différenciera alors la construction d'une structure de donnée infinie de la preuve d'une de ses propriétés. À titre d'exemple introductif, nous allons définir le prédicat caractérisant les flots infinis.

14.6.1 Le prédicat « être infini »

Nous avons déjà caractérisé la finitude d'un flot de (`LList` A) par le prédicat inductif `Finite` : une preuve de finitude de u est un terme composé d'un nombre fini d'applications du constructeur `Finite_LCons` et d'une application de `Finite_LNil`. D'une façon symétrique, nous proposons la définition du type co-inductif `Infinite` à un seul constructeur :

```
CoInductive Infinite (A:Set) : LList A → Prop :=
    Infinite_LCons :
      ∀(a:A)(l:LList A), Infinite l → Infinite (LCons a l).
Hint Resolve Infinite_LCons : llists.
```

Il nous reste à présenter des techniques de preuve sur les prédicats co-inductifs, en commençant par le prédicat `Infinite`. Nous commençons par les techniques d'introduction, qui consistent en la construction d'un « terme de preuve infini », ou plutôt d'un procédé de construction définissant un tel terme. Les techniques d'élimination sont une partie de celles déjà vues à propos des types inductifs.

14.6.2 Construction de preuves infinies

Approche intuitive du problème

Commençons par un exemple simple : nous avons défini la fonction `from` qui à tout n associe le flot des entiers naturels à partir de n. Nous voulons prouver que tout flot construit avec `from` est infini, c'est à dire construire un terme du type suivant :

```
∀n:nat, Infinite (from n)
```

Nous proposons en premier lieu une preuve manuelle afin de mieux comprendre les constructions impliquées dans une telle preuve. La tactique `Cofix`, présentée quelques lignes plus loin, rendra ce type de preuve très simple à construire.

Un moyen de construire un terme du type requis est de prendre le co-point-fixe d'une fonction de " `∀n:nat, Infinite (from n)` " dans lui même, à condition que cette fonction soit compatible avec les conditions de garde par constructeurs.

Nous définissons cette fonction ci-dessous ; bien que son type soit une proposition, nous la définissons avec `Defined` pour la rendre transparente ; ainsi les propriétés de sa définition (et pas seulement celles de son type) pourront être utilisées au moment de la vérification des conditions de garde de `cofix`. Le respect de ces conditions est garanti par l'appel au constructeur `Infinite_LCons`.

```
Definition F_from :
  (∀n:nat, Infinite (from n))→∀n:nat, Infinite (from n).

 intros H n; rewrite (from_unfold n).
 ...
 H : ∀ n:nat, Infinite (from n)
 n : nat
 ============================
  Infinite (LCons n (from (S n)))

 split.

 ...
 H : ∀ n:nat, Infinite (from n)
 n : nat
 ============================
 Infinite (from (S n))

 trivial.
Defined.
```

Il reste alors à utiliser `cofix` pour obtenir une preuve du théorème voulu :

```
Theorem from_Infinite_V0 : ∀n:nat, Infinite (from n).
Proof cofix H : ∀n:nat, Infinite (from n) := F_from H.
```

La tactique `cofix`

Une preuve de propriété co-inductive se fait usuellement en faisant appel à la tactique `cofix`, qui nous dispense de construire une preuve auxiliaire comme dans l'exemple précédent. Mais le principe reste le même : pour prouver une proposition de la forme P, où P est un prédicat co-inductif, on bâtit un terme de la forme " `cofix` H `:` P `:=` t " où t a pour type P dans le contexte étendu par la déclaration `(`H `:` P`)` et le terme ainsi construit respecte la condition de garde par constructeurs.

Du point de vue de l'usager, la tactique `cofix` H se charge de l'introduction de l'hypothèse H et de l'activation d'un but d'énoncé P ; une fois ce but résolu, le terme complet est construit et la condition de garde est testée. Cette tactique est illustrée par une démonstration interactive que tout flot construit par `from` est infini.

```
Theorem from_Infinite : ∀n:nat, Infinite (from n).
Proof.
 cofix H.
 ...
 H : ∀ n:nat, Infinite (from n)
 ============================
  ∀ n:nat, Infinite (from n)

 intro n; rewrite (from_unfold n).
 split; auto.
Qed.
```

Le lecteur pourra — en imprimant le terme de preuve de `from_Infinite` — observer la construction de co-point fixe et le respect de la garde.

Notons également que `Cofix` peut s'utiliser sans donner d'arguments. Le système choisit en général d'introduire une hypothèse portant le nom du théorème à prouver, ce qui souligne le caractère un peu déstabilisant des preuves par co-induction.

14.6.3 Manque de respect de la garde

Dans la preuve ci-dessus, c'est l'appel à `split` qui impose le respect de la garde. Une tentative de pousser l'automatisation trop loin risquerait de faire perdre ce respect et de mener directement à l'échec.

Dans le script suivant, l'appel à “ `auto with llists` ” privilégie l'application directe de l'hypothèse `H`, et le terme de preuve construit ne respecte plus la condition de garde. Ce genre de situation est l'un des rares où l'utilisateur, après avoir sauté de joie au message indiquant la fin de la preuve interactive, est d'autant plus déçu quand *Coq* refuse à juste titre de sauvegarder la preuve.

```
Lemma from_Infinite_buggy : ∀n:nat, Infinite (from n).
 Proof.
 cofix H.
 auto with llists.
 Proof completed.
 Qed.
 Error: Recursive definition of H is ill-formed.
In environment
H : ∀ n:nat, Infinite (from n)
```

unguarded recursive call in H

Dans le cas de preuves beaucoup plus complexes que la précédente, on peut s'interroger sur la perversité du système qui nous laisse construire un terme de preuve pour nous annoncer sa non-conformité au moment de la sauvegarde. Heureusement, la commande `Guarded` permet de tester pendant cette construction si la condition de garde est respectée jusqu'à présent. Nous conseillons d'y faire appel à chaque moment de doute, et surtout à chaque utilisation — même implicite par `assumption`, `auto` ou autre — de l'hypothèse introduite par `cofix`.

Sur notre petit exemple, l'utilisation de cette commande nous aurait mis en garde à temps et permis de prendre une autre approche.

```
Lemma from_Infinite_saved : ∀n:nat, Infinite (from n).
Proof.
 cofix H.
 auto with llists.
 Guarded.
 Error: Recursive definition of H is ill-formed.
In environment
H : ∀ n:nat, Infinite (from n)
unguarded recursive call in H
 Undo.
 intro n; rewrite (from_unfold n).
 split; auto.
 Guarded.
The condition holds up to here
Qed.
```

Exercice 14.12 * Prouver — à l'aide de la tactique `cofix` — les lemmes suivants :

```
Lemma repeat_infinite : ∀(A:Set)(a:A), Infinite (repeat a).

Lemma general_omega_infinite :
 ∀(A:Set)(a:A)(u v:LList A),
   Infinite (general_omega v (LCons a u)).
```

Déduire du dernier lemme le théorème :

```
Theorem omega_infinite :
 ∀(A:Set)(a:A)(l:LList A), Infinite (omega (LCons a l)).
```

Exercice 14.13 Un apprenti distrait se trompe de mot-clef et donne une définition *inductive* de l'infinitude :

```
Inductive BugInfinite (A:Set) : LList A → Prop :=
    BugInfinite_intro :
      ∀(a:A)(l:LList A),
        BugInfinite l → BugInfinite (LCons a l).
```

Montrer que le type ainsi défini est toujours vide.

Exercice 14.14 ** Sur le type des arbres binaires paresseux, définir les prédicats « avoir au moins une branche infinie » et « avoir toutes ses branches infinies », de même pour les branches finies. Pour chacun de ces prédicats, construire un arbre le satisfaisant, et valider votre exemple par une preuve.

On étudiera également les relations entre ces prédicats, en remarquant que l'implication suivante :

> « Si un arbre n'a aucune branche finie, alors il possède une branche infinie »

ne peut être prouvée qu'en logique classique, c'est à dire en admettant l'hypothèse suivante :

```
∀P:Prop, ~~P→P.
```

14.6.4 Techniques d'élimination

Soit C un prédicat co-inductif défini sur un type co-inductif A. Comment prouver un théorème de la forme “ `∀a:A, C a → P a` ” ? Il est clair que nous ne disposons plus de techniques de preuves par récurrence, étant donné le caractère potentiellement infini des objets considérés. Restent l'analyse par cas sur une hypothèse `(H:(C a))` et les techniques d'inversion. Nous illustrons ces techniques par un exemple simple et laissons quelques preuves intéressantes en exercice.

LNil n'est pas infini

La preuve suivante utilise une inversion sur une hypothèse d'énoncé “ `Infinite (LNil (A := A))` ”. Étant donnée l'absence de constructeur du type `Infinite` pour le flot vide, cette inversion permet de terminer la preuve immédiatement.

```
Theorem LNil_not_Infinite :
 ∀A:Set, ~Infinite (LNil (A:=A)).
Proof.
  intros A H; inversion H.
Qed.
```

Exercice 14.15 ** Prouver les résultats suivants (en utilisant diverses techniques de preuves) :

```
Theorem Infinite_of_LCons :
 ∀(A:Set)(a:A)(u:LList A), Infinite (LCons a u)→ Infinite u.

Lemma LAppend_of_Infinite :
 ∀(A:Set)(u:LList A),
   Infinite u → ∀v:LList A, Infinite (LAppend u v).

Lemma Finite_not_Infinite :
 ∀(A:Set)(l:LList A), Finite l → ~Infinite l.

Lemma Infinite_not_Finite :
 ∀(A:Set)(l:LList A), Infinite l → ~Finite l.

Lemma Not_Finite_Infinite :
 ∀(A:Set)(l:LList A), ~Finite l → Infinite l.
```

Exercice 14.16 ** On remarquera dans l'exercice précédent l'absence de deux énoncés :

```
Lemma Not_Infinite_Finite :
 ∀(A:Set)(l:LList A), ~Infinite l → Finite l.

Lemma Finite_or_Infinite :
 ∀(A:Set)(l:LList A), Finite l ∨ Infinite l.
```

Dans le premier cas, aucun argument logique ne nous permet de construire de preuve de “ `Finite l` ” ; nous ne disposons bien sûr pas de récurrence sur `l`, et une analyse par cas sur `l` ne permet de conclure que dans le cas où `l` est vide (*le lecteur pourra procéder à quelques tentatives de preuves pour se convaincre du caractère désespéré de la situation*). Pour le second lemme, un argument très fort (communiqué par Eduardo Gimenez) indique que, si une preuve intuitionniste de `Finite_or_Infinite` pouvait être construite, alors on pourrait en déduire la décidabilité de l'arrêt d'une machine de Turing (*considérer les flots associés aux traces de son exécution*).

Prouver cependant ces résultats en admettant les règles de la logique classique : *on pourra charger le module `Classical` pour cette expérimentation.*

Exercice 14.17 *** On considère les définitions suivantes :

```
Definition Infinite_ok (A:Set)(X:LList A → Prop) :=
  ∀l:LList A,
    X l →  ∃a:A | (∃l':LList A | l = LCons a l' ∧ X l').
Definition Infinite1 (A:Set)(l:LList A) :=
   ∃X:LList A → Prop | Infinite_ok X ∧ X l.
```

Montrer que les prédicats `Infinite` et `Infinite1` sont logiquement équivalents.

14.7 L'égalité extensionnelle (bisimilarité)

14.7.1 Le problème

Nous nous intéressons aux méthodes de preuves d'égalité entre termes de type co-inductif. On peut remarquer que nous avons déjà prouvé un certain nombre de résultats dont la conclusion est une telle égalité ; citons par exemple `LAppend_LNil`, `LAppend_LCons`, `omega_of_Finite`, tous obtenus par une suite finie de dépliages.

Considérons en revanche un énoncé établissant que tout flot infini est absorbant pour la concaténation. Ce serait dans une première tentative le théorème suivant :

```
Lemma Lappend_of_Infinite_0 :
 ∀(A:Set)(u:LList A),
   Infinite u → ∀v:LList A, u = LAppend u v.
```

Le seul moyen dont nous disposons actuellement pour attaquer un tel but est l'analyse par cas sur la variable `u`. Si l'on décompose `u` en “ `LCons a u'` ”, nous obtenons un but similaire au point de départ :

H1 : Infinite u'
v : LList A
============================
u' = LAppend u' v

Exercice 14.18 Écrire le début de preuve menant à cette situation.

Nous voyons bien qu'un nombre fini de telles étapes ne nous avancera pas à grand-chose. Nous pouvons cependant nous restreindre à une relation sur les flots, plus faible que l'égalité et autorisant les raisonnements par co-induction.

14.7.2 Le prédicat bisimilar

NEW: Encore une bizarrerie des args implicites

La définition co-inductive ci-dessous formalise le type des *preuves finies ou infinies* d'égalités entre flots.

```
CoInductive bisimilar (A:Set) : LList A → LList A → Prop :=
  | bisim0 : bisimilar LNil LNil
  | bisim1 :
      ∀(a:A)(l l':LList A),
        bisimilar l l' → bisimilar (LCons a l)(LCons a l').

Hint Resolve bisim0 bisim1  : llists.
```

Une preuve de la proposition “ `bisimilar u v` ” peut alors être vue comme un terme de preuve fini ou infini bâti avec les constructeurs `EqLNil` et `EqLCons`. Bien entendu, ces termes de preuves seront la plupart du temps construits à l'aide de la tactique `cofix`, à condition de respecter les conditions de garde.

Exercice 14.19 Après avoir chargé si nécessaire le module `Relations` de la bibliothèque *Coq*, montrer que `bisimilar` est une relation d'équivalence. Entre autres résultats, la reflexivité de `bisimilar` :

```
Lemma bisimilar_refl : ∀A:Set, reflexive _ (bisimilar (A:=A)).
```

montre bien que `bisimilar` est une relation plus grossière que l'égalité usuelle de *Coq*.

Exercice 14.20 ** Afin de mieux comprendre `bisimilar`, montrer les deux théorèmes suivants établissant une relation entre `bisimilar` et la fonction `LNth` définie en 14.1, page 415 :

```
Lemma bisimilar_LNth :
 ∀(A:Set)(n:nat)(u v:LList A),
   bisimilar u v → LNth n u = LNth n v.

Lemma LNth_bisimilar :
 ∀(A:Set)(u v:LList A),
   (∀n:nat, LNth n u = LNth n v)→ bisimilar u v.
```

Exercice 14.21 On s'amusera à comparer les techniques de preuve des deux théorèmes suivants :

```
Theorem bisimilar_of_Finite_is_Finite :
 ∀(A:Set)(l:LList A),
   Finite l → ∀l':LList A, bisimilar l l' → Finite l'.


Theorem bisimilar_of_Infinite_is_Infinite :
 ∀(A:Set)(l:LList A),
   Infinite l → ∀l':LList A, bisimilar l l' → Infinite l'.
```

Exercice 14.22 Montrer que la restriction du prédicat `bisimilar` aux flots finis est l'égalité de *Coq* ; autrement dit :

```
Theorem bisimilar_of_Finite_is_eq :
 ∀(A:Set)(l:LList A),
   Finite l → ∀l':LList A, bisimilar l l' → l = l'.
```

Exercice 14.23 ** Reprendre l'exercice précédent avec les arbres binaires potentiellement infinis : on définira une égalité extensionnelle `LTree_bisimilar`, qu'on caractérisera au moyen d'une fonction d'accès aux nœuds d'un arbre, d'une façon similaire à l'exercice 14.20.

14.7.3 Quelques résultats intéressants

Nous présentons quelques preuves sur `bisimilar`, pour lesquelles rien ne pouvait être fait avec l'égalité de *Coq*.

Associativité de `LAppend`

L'associativité de `LAppend` — exprimée à l'aide de `bisimilar` se fait par co-induction autour d'une analyse par cas sur le flot `u`.

```
Theorem LAppend_assoc :
 ∀(A:Set)(u v w:LList A),
   bisimilar (LAppend u (LAppend v w))(LAppend (LAppend u v) w).

Proof.
 intro A; cofix H.
 destruct u; intros;
 autorewrite [llists] using auto with llists.
 apply bisimilar_refl.
Qed.
```

Exercice 14.24 * Montrer que toute flot infini est absorbant (à gauche) pour la concaténation :

```
Lemma LAppend_of_Infinite_bisim :
  ∀(A:Set)(u:LList A),
    Infinite u → ∀v:LList A, bisimilar u (LAppend u v).
```

Exercice 14.25 *** Montrer que le flot " `omega` *u* " est un point fixe pour la concaténation :

```
Lemma omega_lappend :
 ∀(A:Set)(u:LList A),
    bisimilar (omega u)(LAppend u (omega u)).
```

Dans cette dernière preuve, il est conseillé de prouver d'abord un lemme sur `general_omega`. On remarquera la différence avec `omega_of_Finite` dont la conclusion est une égalité au sens de *Coq*, au prix de la condition de finitude sur `u`.

Exercice 14.26 ** On continue l'exercice 14.23 ; montrer que si un arbre a toutes ses branches infinies, alors il est absorbant pour la fonction de greffe (`graft`) définie dans l'exercice 14.4.

14.8 Le principe de Park

Nous adaptons aux flots les explications du tutoriel d'Eduardo Gimenez[45].

Une *bisimulation* est une relation binaire R définie sur `(LList` A`)` et telle que si $R(u, v)$, alors,

– soit u et v sont égaux à `LNil`,

– soit il existe a, u' et v' tels que $u =$“ `LCons` a u' ”, $v =$“ `(LCons` a v'`)` ” et $R(u', v')$.

```
Definition bisimulation (A:Set)(R:LList A → LList A → Prop) :=
  ∀l1 l2:LList A,
    R l1 l2 →
    match l1 with
    | LNil ⇒ l2 = LNil
    | LCons a l'1 ⇒
        match l2 with
        | LNil ⇒ False
        | LCons b l'2 ⇒ a = b ∧ R l'1 l'2
        end
  end.
```

Exercice 14.27 *** Prouver le théorème suivant (principe de Park) :

```
Theorem park_principle :
 ∀(A:Set)(R:LList A → LList A → Prop),
   bisimulation R → ∀l1 l2:LList A, R l1 l2 →
   bisimilar l1 l2.
```

Exercice 14.28 * Appliquer le principe de Park pour prouver que les deux flots suivants sont bisimilaires.

```
CoFixpoint alter : LList bool := LCons true (LCons false alter).

Definition alter2 : LList bool :=
  omega (LCons true (LCons false LNil)).
```

On pourra considérer la relation binaire ci-dessous, et prouver que c'est une bisimulation :

```
Definition R (l1 l2:LList bool) : Prop :=
  l1 = alter ∧ l2 = alter2 ∨
  l1 = LCons false alter ∧ l2 = LCons false alter2.
```

14.9 LTL

Afin de finir de nous familiariser avec les définitions inductives et co-inductives, nous proposons un ensemble de définitions empruntées au domaine de la vérification et notamment la logique temporelle linéaire (voir [77]). Les définitions que nous présentons sont une adaptation des travaux menés avec Davy Rouillard en Isabelle/HOL[23]. Le travail de Solange Coupet-Grimal[30], distribué dans les contributions de *Coq*, formalise la notion de formule LTL restreinte aux exécutions infinies (alors que nous considérons des exécutions finies ou infinies) ; un article détaillé se trouve en [31].

Contrairement au développement présenté ci-dessous, focalisé sur les exécutions et leurs propriétés, l'intérêt se porte sur la notion de formule LTL et ses propriétés abstraites. Néanmoins, l'utilisation de la co-induction est la même dans les deux approches, et nous conseillons vivement au lecteur d'étudier cette contribution.

Ouvrons une section contenant la déclaration d'un type A:`Set` et quelques variables de type A pour développer quelques exemples.

```
Section LTL.
Variables (A : Set)(a b c : A).
```

Nous nous intéressons aux propriétés des flots sur A. Afin de disposer d'une notation intuitive, nous écrirons " `satisfies l P` " pour " `P l` " :

```
Definition satisfies (l:LList A)(P:LList A → Prop) : Prop :=
  P l.
Hint Unfold satisfies : llists.
```

Le prédicat `Atomic`

La définition suivante permet de convertir n'importe quel prédicat At défini sur A en un prédicat sur `(llist` A`)` : le flot l satisfait " `Atomic At` " s'il commence par un élément satisfaisant At.

```
Inductive Atomic (At:A→Prop) : LList A → Prop :=
  Atomic_intro :
    ∀ (a:A)(l:LList A), At a → Atomic At (LCons a l).

Hint Resolve Atomic_intro : llists.
```

Le prédicat `Next`

" `Next P` " est le prédicat caractérisant tout flot dont la queue satisfait `P` :

```
Inductive Next (P:LList A → Prop) : LList A → Prop :=
  Next_intro : ∀ (a:A)(l:LList A), P l → Next P (LCons a l).

Hint Resolve Next_intro : llists.
```

À titre d'exemple, nous montrons que le flot commençant par `a` et suivie d'une infinité de `b` satisfait la formule " `Next (Atomic (eq b))` " (propriété caractérisant les flots dont le second élément est égal à `b`) :

```
Theorem Next_example :
 satisfies (LCons a (repeat b))(Next (Atomic (eq b))).
Proof.
  rewrite (repeat_unfold b); auto with llists.
Qed.
```

Le prédicat `Eventually`

" `Eventually` P " est le prédicat caractérisant tout flot dont au moins un suffixe (non vide) satisfait P ; on remarquera que le premier constructeur est écrit de façon à exclure `LNil` des suffixes considérés.

```
Inductive Eventually (P:LList A → Prop) : LList A → Prop :=
  Eventually_here :
   ∀(a:A)(l:LList A), P (LCons a l)→
                      Eventually P (LCons a l)
| Eventually_further :
   ∀(a:A)(l:LList A), Eventually P l →
                      Eventually P (LCons a l).

Hint Resolve Eventually_here Eventually_further.
```

Exercice 14.29 (**) On considère le lemme suivant, exprimant une compatibilité entre `Eventually` et `LAppend`.

```
Theorem Eventually_of_LAppend :
 ∀(P:LList A → Prop)(u v:LList A),
 Finite u → satisfies v (Eventually P)→
 satisfies (LAppend u v)(Eventually P).
Proof.
 unfold satisfies; induction 1; intros;
  autorewrite [llists] using auto with llists.
Qed.
```

Comment est exploitée l'hypothèse de finitude ? Est-elle nécessaire ? Si oui, construire et prouver un contre-exemple.

Le prédicat `Always`

(`Always` P) est le prédicat caractérisant les flots dont tous les suffixes sont non vides et satisfont P. Il est alors naturel de proposer une définition co-inductive à un seul constructeur.

```
CoInductive Always (P:LList A → Prop) : LList A → Prop :=
  Always_LCons :
  ∀(a:A)(l:LList A),
  P (LCons a l)→ Always P l → Always P (LCons a l).
```

Exercice 14.30 Démontrer que tout flot qui satisfait “ `Always P` ” est infinie.

Exercice 14.31 * Prouver que tout suffixe de “ `repeat a` ” commence par `a` ; en d’autres termes, prouver le théorème :

```
Lemma always_a : satisfies (repeat a)(Always (Atomic (eq a))).
```

Le prédicat F^∞

`(F_Infinite P)` est le prédicat caractérisant toute flot infini dont une infinité de suffixes satisfont `P` ; ce prédicat peut facilement se définir à l’aide de `Always` et `Eventually`.

```
Definition F_Infinite (P:LList A → Prop) : LList A → Prop :=
  Always (Eventually P).
```

Exercice 14.32 ** Démontrer que le flot l_ω consistant en une alternance de `a` et `b` possède un nombre infini d’occurrences de `a`.

Le prédicat G^∞

“ `G_Infinite P` ” est le prédicat caractérisant tout flot infini dont tous les suffixes (sauf un nombre fini) satisfont `P` :

```
Definition G_Infinite (P:LList A → Prop) : LList A → Prop :=
  Eventually (Always P).
```

Exercice 14.33 * Démontrer les deux théorèmes suivants, établissant que le prédicat “ `G_Infinite P` ” est stable par ajout ou retrait d’un préfixe fini :

```
Lemma LAppend_G_Infinite :
 ∀(P:LList A → Prop)(u v:LList A),
   Finite u → satisfies v (G_Infinite P)→
   satisfies (LAppend u v) (G_Infinite P).


Lemma LAppend_G_Infinite_R :
 ∀(P:LList A → Prop)(u v:LList A),
 Finite u → satisfies (LAppend u v) (G_Infinite P)→
 satisfies v (G_Infinite P).

End LTL.
```

14.10 Exemple d'étude : systèmes de transitions

Nous présentons ici le squelette d'un développement concernant les systèmes de transitions, ou plus simplement les automates. Nous ne donnons que les définitions et énoncés de théorèmes ; les preuves sont données dans le site consacré aux sources et solutions d'exercices à l'adresse suivante :

`www.labri.fr/Perso/~casteran/EXPORTABLE/co-inductifs/`

14.10.1 Définitions

Automates

Un *automate* se définit par un type pour représenter des *états*, un type pour représenter des *actions*, un *état initial* et un ensemble de *transitions*, chacune étant définie par un état de départ, une action, et un état d'arrivée. L'ensemble des transitions sera représenté par une fonction associant à tout état de départ et à toute action une liste d'états d'arrivée ; le modèle choisi est donc non-déterministe.

Les enregistrements (*records*) de *Coq* sont bien adaptés à cette représentation. Ici l'enregistrement doit être défini dans la sorte `Type` car il contient des champs de type `Set`. Une définition dans la sorte `Set` serait inutilisable, pour les raisons évoquées en section 15.2.4.

```
 Record automaton : Type :=
   mk_auto {
    states : Set;
    actions : Set;
    initial : states;
    transitions : states → actions → list states
}.
```

Traces

Une *trace* est une suite d'actions correspondant à une suite de transitions dans un automate. Cette trace peut être finie ou infinie, et dans le premier cas mène à un état « bloquant ».

Nous définissons de manière co-inductive un prédicat `Traces` tel que la proposition “ `Traces A q l` ” signifie « `l` est la trace d'une exécution dans `A` à partir de `q` ». Dans la définition suivante, la proposition “ `deadlock q` ” signifie « aucune transition n'est issue de q ».

```
Definition deadlock (A:automaton)(q:states A) :=
  ∀a:actions A, @transitions A q a = nil.

Unset Implicit Arguments.

CoInductive Trace (A:automaton) :
```

```
    states A → LList (actions A)→ Prop :=
  empty_trace :
   ∀q:states A, deadlock A q → Trace A q LNil
| lcons_trace :
   ∀(q q':states A)(a:actions A)(l:LList (actions A)),
    In q' (transitions A q a)→ Trace  A  q' l →
    Trace A  q (LCons a l).

Set Implicit Arguments.
```

Exercice 14.34 *** On considère l'automate décrit ci-dessous :

```
(* states *)
Inductive st : Set := q0 | q1 | q2.

(* actions *)
Inductive acts : Set := a | b.

(* transitions *)
Definition trans (q:st)(x:acts) : list st :=
  match q, x with
  | q0, a ⇒ cons q1 nil
  | q0, b ⇒ cons q1 nil
  | q1, a ⇒ cons q2 nil
  | q2, b ⇒ cons q2 (cons q1 nil)
  | _, _ ⇒ nil (A:=_)
  end.

Definition A1 :=  mk_auto q0 trans.
```

1. Dessiner cet automate.
2. Montrer que toute trace dans A_1 comporte un nombre infini d'actions `b` :

   ```
   Theorem Infinite_bs :
    ∀(q:st)(t:LList acts), Trace A1 q t →
    satisfies t (F_Infinite (Atomic (eq b))).
   ```

14.11 Exploration des types co-inductifs

Nous sommes loin d'avoir présenté toutes les possibilités offertes par les constructions co-inductives de *Coq*. Nous invitons le lecteur à lire les travaux d'Eduardo Gimenez [46, 45] et la documentation de *Coq*. Les types co-inductifs sont également utiles pour la vérification de circuits [32].

```
Check and.
(* Fin de chapitre *)
```

```
Definition isEmpty (A:Set)(l:LList A) : Prop :=
  match l with LNil ⇒ True | LCons a l’ ⇒ False end.

Definition LHead (A:Set)(l:LList A) : option A :=
  match l with | LNil ⇒ None | LCons a l’ ⇒ Some a end.

Definition LTail (A:Set)(l:LList A) : LList A :=
  match l with LNil ⇒ LNil | LCons a l’ ⇒ l’ end.

Fixpoint LNth (A:Set)(n:nat)(l:LList A){struct n} : option A :=
  match l with
  | LNil ⇒ None
  | LCons a l’ ⇒
    match n with O ⇒ Some a | S p ⇒ LNth p l’ end
  end.
```

FIGURE 14.1 – Fonctions d’accès dans un flot

Chapitre 15

** Fondements des types inductifs

15.1 Règles de bonne formation

Les définitions de types inductifs présentent plusieurs degrés de liberté. Le type lui même peut être une constante dont le type est l'une des sortes du système, mais il peut aussi être une fonction, cette fonction peut avoir un type dépendant et certains des arguments de cette fonction peuvent être des paramètres. Si l'on passe aux constructeurs, il peuvent être des constantes, mais également des fonctions, éventuellement avec un type dépendant, les arguments de ces fonctions peuvent être dans le type inductif ou non et ils peuvent même être des fonctions. Dans cette section nous allons étudier les limites de cette liberté.

15.1.1 Le type inductif lui-même

La définition d'un type inductif ajoute dans le contexte une nouvelle constante ou fonction dont le type final, après que l'on aura fourni suffisamment d'arguments, se trouve dans l'une des sortes, `Prop`, `Set`, et `Type`.

Lorsque la fonction décrivant le type inductif prend un ou plusieurs arguments, il faut distinguer les arguments *paramétriques* des arguments normaux. Les arguments paramétriques sont ceux qui apparaissent entre parenthèse avant le caractère « deux points », "`:`", dans la ligne de définition du type.

Si une définition de type inductif reçoit un argument paramétrique, lar portée de cet argument paramétrique est l'ensemble de la définition : on peut utiliser cet argument dans les arguments paramétriques suivants, dans le type du type, et dans le type des constructeurs. Reprenons par exemple la définition paramétrique des listes polymorphes, fournie dans les bibliothèques de *Coq* par le module `List` et déjà décrite dans la section 7.4.1 :

```
Set Implicit Arguments.
```

```
Inductive list (A:Set) : Set :=
  nil : list A
| cons : A → list A → list A.

Implicit Arguments nil [A].
```

La déclaration de paramètre `[A:Set]` introduit un type `A` dans la sorte `Set` et on peut donc utiliser cet identificateur dans la description des types pour les deux constructeurs `nil` et `cons`. En fait, le type annoncé pour chacun des constructeurs ne correspond pas exactement au type avec lequel on pourra les utiliser après la déclaration du type inductif. Ainsi, si nous vérifions le type de `nil`, nous obtenons la réponse suivante (nous avons besoin de préfixer l'identificateur avec `@` pour ignorer la spécification des arguments implicites) :

```
Check (@nil).
```
@nil : ∀ A:Set, list A

En pratique, le type de chacun des constructeurs est donc le type donné dans la définition du type inductif, augmenté d'un préfixe correspondant aux paramètres de la définition inductive.

L'utilisation des arguments paramétriques doit respecter une contrainte de stabilité qui se résume dans la phrase suivante : *les arguments paramétriques doivent être inchangés dans toutes les occurrences du type inductif apparaissant dans la définition inductive.*

Dans l'exemple des listes polymorphes, cette contrainte impose que `list` apparaît toujours appliqué à `A` dans la deuxième et la troisième ligne de la définition inductive. Lorsque cette contrainte n'est pas respectée, le système *Coq* rejette la définition avec un message d'erreur assez explicite. Voici par exemple une définition de type inductif paramétré erronée :

```
Inductive T (A:Set) : Set := c : ∀B:Set, B → T B.
```
Error: The 1st argument of T must be A in ∀ B:Set, B → T B

Lorsque le type inductif prend des arguments dont la valeur peut varier à chaque utilisation dans la définition inductive, il est nécessaire de faire apparaître les arguments variables dans la déclaration de type, en dehors de la liste des paramètres, comme nous l'avons vu dans la définition d'arbres de hauteur fixée en section 7.5.2 avec la définition du type `htree` que nous répétons ici :

```
Inductive htree (A:Set) : nat→Set :=
  hleaf : A → htree A 0
| hnode : ∀n:nat, A → htree A n → htree A n → htree A (S n).
```

Ici, le type `htree` prend deux arguments, dont le premier est paramétrique et le second est normal. La contrainte que le type paramétrique réapparaisse inchangé est bien satisfaite et l'on voit également que l'argument normal prend trois valeurs différentes dans les quatre occurrences du type inductif qui apparaissent dans la description des constructeurs.

Dans les deux exemples que nous avons donnés ci-dessus, il apparaît que l'argument paramétrique est lui-même un type. Ceci n'est pas une nécessité, et il est possible de faire des déclarations paramétriques dans lesquelles certains paramètres sont des données, comme nous l'avons vu pour la définition inductive de l'égalité en section 9.2.6 et pour des spécifications fortes de fonctions (sections 7.5.1 et 10.4.2).

De manière générale, il semble toujours possible de construire un type inductif sans paramètres à partir d'un type inductif paramétré, en « déclassant » les paramètres pour en faire des arguments normaux. Cette transformation est rarement intéressante, car les principes de récurrence sont plus simples pour les types inductifs paramétrés.

Par exemple, on aurait pu définir les listes polymorphes de la façon suivante

```
Inductive list' : Type→Type :=
  nil' : ∀A:Type, list' A
| cons' : ∀A:Type, A → list' A → list' A.
```

Mais on obtient alors un principe de récurrence hideux :

```
Check list'_ind.
```

list'_ind
: ∀ P:∀ T:Type, list' T → Prop,
(∀ A:Type, P A (nil' A))→
(∀ (A:Type)(a:A)(l:list' A), P A l → P A (cons' A a l))→
∀ (T:Type)(l:list' T), P T l

Ce principe de récurrence est plus complexe car il quantifie sur un prédicat avec type dépendant à deux arguments au lieu de quantifier sur un prédicat à un seul argument.

15.1.2 Formation des constructeurs

Les constructeurs d'un type inductif T sont des constantes ou des fonctions dont le type final doit être T (lorsque le type est constant) ou une application de T à des arguments (lorsque le type T est une fonction). Ceci se reconnaît syntaxiquement parce que le type de chaque constructeur pour le type T a la forme suivante :

$$c : t_1 \to t_2 \to \cdots \to t_l \to T\ a_1\ \ldots\ a_k \tag{15.1}$$

et l'expression "$T\ a_1\ \ldots\ a_k$" doit déjà être bien formée, c'est-à-dire qu'elle doit satisfaire la contrainte relative aux arguments paramétriques et les contraintes de typage. En outre, le type T ne peut pas réapparaître parmi les arguments $a_1 \ldots a_k$, même si les contraintes de typage le permettaient. Par exemple, la définition suivante est rejetée :

```
Inductive T : Set→Set := c : (T (T nat)).
```

Error: Non strictly positive occurrence of T in T (T nat)

Dans la description de type 15.1, nous avons laissé entendre que le type du constructeur était un type non dépendant. Ce n'est bien sûr pas nécessaire. Il reste néanmoins une contrainte de bonne formation sur les termes $t_1 \ldots t_l$. Chacun de ces termes peut être une constante ou une fonction et le type inductif ne peut apparaître dans ces termes qu'à l'intérieur du type final : si t_i est un type de constante alors t_i peut être de la forme "$g\ (T\ b_{1,1}\ \ldots\ b_{1,k}) \cdots (T\ b_{l,1}\ \ldots\ b_{l,k})$," à condition que les expressions $b_{i,j}$ satisfassent encore les contraintes de typage, que T n'apparaisse ni dans ces expressions ni dans g. Si t_i est un type de fonction, alors ce type de fonction peut être de la forme

$$t'_1 \rightarrow \ldots t'_m \rightarrow g\ (T\ b_{1,1}\ \ldots\ b_{1,k}) \cdots (T\ b_{l,1}\ \ldots\ b_{l,k})$$

mais le type T ne peut pas apparaître ni dans les expressions $t'_1 \ldots t'_m$, ni dans les expressions $b_{i,j}$. On appelle ces contraintes les contraintes de *positivité stricte*.

Par exemple, la définition inductive suivante, qui généralise la définition d'arbres à branchement infini donnée en section 7.3.5.2, est bien formée :

```
Inductive inf_branch_tree (A:Set) : Set :=
  inf_leaf : inf_branch_tree A
| inf_node : A →(nat → inf_branch_tree A)→ inf_branch_tree A.
```

Le premier constructeur a un type de constante et `(inf_branch_tree A)` satisfait bien les contraintes données à la section précédente. Le second constructeur prend deux arguments et le type final est à nouveau bien formé (c'est le même que pour le premier constructeur). Le premier argument du second constructeur ne fait pas intervenir le type `inf_branch_tree` et utilise le paramètre qui est bien un type. Il satisfait bien les contraintes de bonne formation et les contraintes de typage. Le type du second argument du second constructeur est un type de fonction dont le type final est le type `inf_branch_tree` dont l'utilisation est bien formée. Ce type de fonction indique qu'il n'y a qu'un argument, de type `nat`, ce type est bien différent du type inductif.

En revanche, la définition suivante est un exemple simple de définition rejetée par la contrainte que T ne doit pas apparaître dans les expressions t'_i.

```
Inductive T : Set := l : (T→T)→T.
```
Error: Non strictly positive occurrence of T in (T→T)→T

Une description plus précise des règles de bonne formation des types inductifs est donnée dans la documentation du système *Coq* et dans l'article [72], mais nous pouvons déjà tâcher de comprendre les raisons qui font rejeter ce type de définition.

Si le type `T` était accepté, il serait également possible de définir les fonctions suivantes :

```
Definition t_delta : T :=
 (l fun t:T ⇒ match t with (l f) ⇒ f t end).

Definition t_omega: T :=
  match t_delta with l f ⇒ (f t_delta) end.
```

Si l'on cherche à évaluer l'expression `omega`, on s'aperçoit cette expression est réductible par ι-réduction. L'expression obtenue est alors :

```
(fun t:T ⇒ match t with l f ⇒ f t end t_delta)
```

Soit après β-réduction :

```
match t_delta with l f ⇒ f t_delta end
```

Nous avons à nouveau une expression ι-réductible, mais c'est la même qu'au départ. Admettre ce type de construction inductive mènerait donc à perdre l'assurance que les réductions du Calcul des Constructions terminent toujours. On ne serait alors même plus assuré que la vérification qu'un terme a un type donné soit décidable.

Cet exemple montre que les définitions de types inductifs avec des occurrences non strictement positives mettent en danger la terminaison de l'algorithme de vérification des types pour le Calcul des Constructions Inductives. Elles permettraient même de construire une preuve de `False`, ce qui met en danger la cohérence du système. Considérons par exemple la fonction suivante. Voici un exemple de fonction que l'on pourrait définir si ce type était accepté :

```
Definition depth : T→nat :=
  fun t:T ⇒ match t with l f ⇒ S (depth (f t)) end.
```

Pour le calcul de cette fonction sur le terme `(l fun t:T ⇒t)`, on obtiendrait l'égalité suivante en une ι-réduction :

```
    (depth (l (fun t:T ⇒ t)) =
        (S (depth ((fun t:T ⇒ t) (l (fun t:T ⇒ t))))
```

En ajoutant une β-réduction, on obtiendrait alors :

```
(depth (l (fun t:T ⇒ t)) = (S (depth (l (fun t:T ⇒ t))).
```

On pourrait alors conclure à une preuve de `False`, grâce au théorème `n_Sn` :

$n\_Sn : \forall\, n{:}nat,\ n \neq S\ n$

15.1.3 Comment se construit le principe de récurrence

Dans cette section, réservée au lecteur curieux des aspects fondamentaux du Calcul des Constructions Inductives et qui peut être délaissée par l'utilisateur pragmatique, nous résumons la méthode utilisée pour construire le principe de récurrence à partir de la définition inductive. Les principes de récurrence sont des outils pour démontrer que des propriétés sont vérifiées pour tous les éléments d'un type inductif donné. Pour cette raison, chaque principe de récurrence est constitué d'un *en-tête* présentant un certain nombre de quantifications universelles et terminé par une quantification sur un prédicat P. Ce prédicat porte sur les éléments du type inductif. Puis viennent un certain nombre d'implications dont les prémisses sont ce que nous avons appelé les *prémisses principales*. Enfin on trouve un épilogue, qui exprime toujours que la propriété P est vérifiée pour tous les éléments du type inductif. Nous allons décrire l'en-tête et l'épilogue avant de revenir sur les prémisses principales.

Considérons dans la suite de cette section que nous étudions le principe de récurrence pour un type inductif T.

Génération de l'en-tête

Si T est un type dépendant, on construit en réalité une famille de types inductifs, indexés par des éléments apparaissant comme arguments de T. Parmi les arguments de T les paramètres jouent un rôle particulier.

Les définitions inductives avec paramètre se font comme les définitions inductives sans paramètre dans un contexte où le paramètre est figé. La définition est ensuite généralisée à un contexte où le paramètre peut reprendre sa liberté. Cette opération de généralisation entraîne l'apparition d'une quantification universelle devant tous les éléments de la définition inductive. Le principe de récurrence pour un type paramétré commence donc par une quantification universelle sur les paramètres.

Par exemple, le principe de récurrence pour les listes polymorphes (voir section 7.4.1) et le principe de récurrence pour les arbres de hauteur fixée (voir section 7.5.2) commencent par une quantification universelle sur un élément `A` de sorte `Set` :

```
∀A:Set, ...
```

Pour les arguments de T qui ne sont pas des paramètres, il est nécessaire de faire apparaître la possibilité qu'ont ces arguments de varier entre les différents sous-termes d'un même élément du type inductif, comme nous l'avons déjà vu pour le type `htree` en section 7.5.2. Puisqu'ils peuvent varier, il faut exprimer explicitement que la propriété à prouver peut en dépendre. Donc la propriété que l'on cherche à prouver, qui doit normalement porter sur les éléments de T, n'a pas qu'un seul argument, en fait elle a $k+1$ arguments si T a k arguments non paramétriques. Par exemple, pour le principe de récurrence sur les entiers naturels on construit la quantification universelle suivante, car il n'y a pas de paramètres et le type n'est pas dépendant :

```
∀P:nat→Prop, ...
```

Considérons le principe de récurrence associé au type `list` : ce type ne dépend pas d'autres arguments que le paramètre `A`. L'en-tête a donc la forme suivante :

```
∀(A:Set)(P:list A → Prop), ...
```

Le cas des arbres de hauteur fixée est légèrement plus complexe : le type inductif associé dépend d'un paramètre ainsi que d'un argument supplémentaire de type `nat` ; l'en-tête engendré prend alors la forme suivante :

```
∀(A:Set)(P:∀n:nat, htree A n → Prop), ...
```

On le voit, lorsque le type est dépendant, la prédicat P a lui-même un type dépendant.

Génération de l'épilogue

Après l'e-tête et les prémisses principales réunies comme prémisses d'implications successives, il reste à fournir la conclusion du principe de récurrence. Il s'agit alors de dire que le prédicat P est bien satisfait par tout élément t de T. Cette conclusion contient donc une quantification universelle sur un élément t de T, suivie de l'application du prédicat P à cet élément. Tous les arguments dont dépend éventuellement le type de t doivent également être quantifiés unversellement et passés en argument au prédicat P.

Par exemple, le principe de récurrence sur les listes polymorphes termine par l'épilogue suivant :

```
...∀l:list A, P l.
```

Pour le type des arbres de hauteur fixée, l'épilogue prend en compte un argument `n` dont dépend le type `(htree A n)`.

```
...∀n:nat, ∀t:htree A n, P n t
```

Génération des prémisses principales

Intuitivement, les prémisses principales ont pour rôle d'assurer que la prédicat P est satisfait pour tous les cas possibles d'utilisation des constructeurs. Il y a donc une prémisse principale pour chaque constructeur. De plus, il s'agit vraiment d'un principe de récurrence au sens où l'on s'autorise à supposer que le prédicat est déjà satissfait par les sous-termes pour vérifier qu'il est satisfait pour le terme complet. Les prémisses principales vont donc contenir une quantification universelle pour chaque argument possible du constructeur, plus une hypothèse de récurrence pour chaque argument du constructeur qui est dans le type T.

Lorsque l'un des arguments est une fonction, comme nous l'avons vu pour le type `Z_fbtree` en section 7.3.5.1, il suffit de quantifier universellement sur une fonction, mais si cette fonction a son image dans le type T, il est également nécessaire de fournir une hypothèse de récurrence pour toutes les images possibles de cette fonction.

La prémisse principale se termine ensuite avec une application de la propriété à prouver au constructeur, lui-même appliqué à tous les arguments pour lesquels des quantifications universelles ont été préalablement introduites, mais pas aux hypothèses de récurrence.

Par exemple, la prémisse principale pour le constructeur `January` du type `month` (voir section 7.1.1) ne présente pas de quantifications universelles ou d'hypothèses de récurrences, car ce constructeur n'a pas d'arguments

```
P January
```

Pour le constructeur `nil` du type `list`, le paramètre de la définition inductive apparaît naturellement, mais il est ensuite caché par le mécanisme d'arguments implicites.

```
P nil
```

Pour le type `vehicle` vu en section 7.1.6, le constructeur `bicycle` prend un argument qui n'est pas dans le type inductif lui-même. On se contente donc d'une seule quantification universelle.

```
∀n:nat, P (bicycle n)
```

Pour le type `nat`, le constructeur `S` prend un argument dans le type `nat`, il y a donc une quantification universelle et une hypothèse de récurrence.

```
∀n:nat, P n → P (S n)
```

Pour le type `Z_btree` (section 7.3.4), le constructeur `Z_bnode` prend trois arguments, dont le premier n'est pas dans le type `Z_btree` et les deux autres y sont, la prémisse principale fait apparaître deux hypothèses de récurrence.

```
∀z:Z, ∀t0:Z_btree, P t0 →
       ∀t1:Z_btree, P t1 → P (Z_bnode z t0 t1)
```

Pour le type `Z_fbtree`, le constructeur `Z_fnode` prend deux arguments, dont le premier n'est pas dans le type `Z_fbtree` et le second est une fonction qui retourne des éléments de `Z_fbtree`, on a donc deux quantifications universelles et une hypothèse de récurrence portant sur toutes les valeurs possibles de la fonction.

```
∀(z:Z)(f:bool→Z_fbtree), (∀x:bool, P (f x))→ P (Z_fnode z f)
```

Enfin, si l'on considère un type inductif dépendant, les hypothèses de récurrence doivent être adaptées aux bons arguments pour être bien typées. Par exemple, le constructeur `hnode` du type `htree` vu en section 7.5.2 prend cinq arguments, dont le premier est un paramètre et les deux derniers sont dans le type inductif considéré, mais pour une hauteur différente. La prémisse principale contient une quantification universelle pour les quatre arguments qui ne sont pas des paramètres. Pour les arguments du constructeur dans le type inductif lui-même (nommés `t` et `t'`), l'argument dépendant (nommé `n`) est ajusté comme dans la définition du constructeur. Le même choix est fait pour le premier argument de la propriété `P` dans les hypothèses de récurrence.

```
∀(n:nat)(x:A)(t:htree A n), P n t →
   ∀t':htree A n, P n t' → P (S n)(hnode A n x t t')
```

15.1.4 Typage des récurseurs

Les fonctions récursives définies sur un type inductif peuvent avoir un type dépendant. Pour chaque type inductif de sorte `Set`, le système *Coq* engendre d'ailleurs une telle fonction récursive à type dépendant, dont le nom est obtenu en accolant au nom du type le suffixe `_rec`. Par exemple, le terme `nat_rec` donné ci-dessous est fourni pour le type des entiers naturels :

```
nat_rec:∀P:nat→Set, P 0 →(∀n:nat, P n → P (S n))→
   ∀n:nat, P n
```

Ce terme peut être utilisé pour définir des fonctions récursives à la place de la commande `Fixpoint` ou de la construction `fix`, nous l'appellerons le *récurseur* associé au type inductif. L'existence de ce récurseur appelle une question : quelle est la relation entre les récurseurs et la commande `Fixpoint` ou la construction `fix` ? Nous allons répondre à cette question en considérant d'abord des formes simples de récurseurs. Ensuite nous montrerons que cette réponse permet également de comprendre l'origine du principe de récurrence associé à un type inductif.

Récursion non dépendante

La forme la plus fréquente des fonctions récursives sur les entiers naturels est celle exhibée par l'expression suivante :

```
Fixpoint f (x:nat) : A :=
 match x with
   0 ⇒ exp1
 | S p ⇒ exp2
 end.
```

Nous avons volontairement laissé A, exp_1 et exp_2 imprécis dans cette déclaration, pour indiquer que ces éléments varient d'une fonction à l'autre. Néanmoins exp_1 et exp_2 sont contraints à avoir le type A. Pour exp_2 la situation est légèrement plus compliquée, car la construction `S p ⇒ ...` est une construction liante pour la variable `p`, de sorte que cette variable peut être utilisée. En outre, la construction de fonctions récursives nous autorise à utiliser la valeur "f `p`." On peut exprimer cela en disant que la déclaration précédente est pratiquement équivalente à la définition suivante :

```
Fixpoint f (x:nat) : A :=
 match x with
   0 ⇒  exp1
 | S p ⇒  exp'2 p (f p)
end.
```

Dans cette formulation, A, exp_1, et exp'_2 doivent être des expressions closes où f n'apparaît pas et typables de la façon suivante :

$$A : \texttt{Set} \qquad exp_1 : A \qquad exp'_2 : nat{\rightarrow}A{\rightarrow}A$$

En pratique, chaque fonction récursive est définie dès que l'on connaît A, exp_1 et exp'_2, de sorte que l'on pourrait pratiquement les définir en supposant qu'il existe une fonction à type dépendant `nat_simple_rec` qui aurait le type suivant :

```
nat_simple_rec:∀A:Set, A →(nat→A→A)→nat→A
```

En fait, cette fonction `nat_simple_rec` peut effectivement être définie en *Coq*, en utilisant la définition suivante :

```
Fixpoint nat_simple_rec (A:Set)(exp1:A)(exp2:nat→A→A)(x:nat)
 {struct x} : A :=
  match x with
  | O ⇒ exp1
  | S p ⇒ exp2 p (nat_simple_rec A exp1 exp2 p)
  end.
```

L'ensemble des fonctions que l'on pourra décrire à l'aide de `nat_simple_rec` contient l'ensemble des fonctions primitives récursives au sens de Dedekind [36]. Mais, puisque `A` peut aussi représenter un type de fonctions, cet ensemble contient également des fonctions qui ne sont pas primitives récursives comme ce fut annoncé par Hilbert [52] et démontré par Ackermann [2]. Du point de vue du Calcul des Constructions, cette fonction `nat_simple_rec` ne permet pas de définir toutes les fonctions intéressantes sur les entiers naturels, car il manque les fonctions qui ont un type dépendant.

Filtrage dépendant

Pour les fonctions à type dépendant le type d'arrivée n'est plus donné par un simple élément du type `Set`, mais par une fonction associant un type à tout élément du type de départ.

Si l'on cherche à construire une fonction à type dépendant, la construction de filtrage `match` intervient naturellement, puisque c'est cette construction qui permet d'indiquer que l'on fait des calculs différents pour des valeurs différentes. On sera donc amené à construire des expressions de filtrage dans lesquelles différentes branches auront des types différents. Ceci impose une difficulté aux outils de typage, qui doivent disposer de la loi de variation du type retourné en fonction de la valeur. Cette loi de variation doit donc être donnée par l'utilisateur.

Par exemple, sur le type des valeurs booléennes on pourra définir une fonction qui envoie `true` vers `O` (de type `nat`) et `false` vers `true` (de type `bool`) en écrivant d'abord la fonction qui associe à chacune des valeurs booléennes un type différent :

```
Definition example_codomain (b:bool) : Set :=
  match b with true ⇒ nat | false ⇒ bool end.
```

Il est ensuite possible de définir une fonction par cas à type dépendant de la manière suivante :

```
Definition example_dep_function (b:bool) : example_codomain b :=
  match b as x return example_codomain x with
  | true ⇒ O
  | false ⇒ true
  end.
```

Comme on le voit, l'utilisateur est obligé de fournir une indication pour préciser le type attendu en retour de chaque branche ; cette indication est donnée sous la forme d'une expression placée après le mot-clef `return` et dépendant d'une variable fournie après le mot-clef `as` dont le type est le même le type de l'expression sur laquelle on effectue le filtrage. Chaque règle de l'expression de filtrage est de la forme suivante :

$$pattern \Rightarrow exp$$

Si l'indication fournie par l'utilisateur a la forme « `as` x `return` t », alors l'expression *exp* de cette règle doit avoir le type $t\{x/pattern\}$. On peut également associer à la variable x et à l'expression t une fonction F dont la valeur est la suivante :

$$F = \texttt{fun}\ x \Rightarrow \texttt{t}.$$

Cette fonction peut aussi être utilisée pour caractériser le typage dépendant, puisque l'expression *exp* doit avoir le type "*F pattern*".

Lorsque le type de l'expression filtrée est lui-même un type dépendant avec des arguments non paramétriques, la fonction F ne peut pas être une fonction à un seul argument, car il est nécessaire de fournir les arguments dépendants du type de l'expression filtrée. L'utilisateur doit alors fournir une indication de typage de la forme suivante :

```
as x in type_name _ _ c d return t
```

Si l'expression filtrée a le type "*type_name A B c d*," alors la fonction F utilisée pour le typage a la valeur suivante :

```
fun c d ⇒ fun x:type_name A B c d ⇒ t.
```

Comme nous l'avons fait précédemment pour la récursion, nous pouvons décrire le filtrage dépendant sur les booléens par l'existence d'une fonction à type dépendant `bool_case` qui prend en argument la fonction F de type `bool` → `Set` et les deux expressions à fournir pour les constructeurs du type inductif, qui doivent respectivement être de type (F `true`) et (F `false`). La fonction `bool_case` a donc le type suivant :

```
bool_case:∀F:bool→Set, F true → F false → ∀x:bool, F x
```

La fonction `bool_case` peut effectivement être définie dans le système *Coq* en utilisant la commande suivante :

```
Definition bool_case
   (F:bool→Set)(v1:F true)(v2:F false)(x:bool) :=
  match x return F x with true ⇒ v1 | false ⇒ v2 end.
```

Le même type de construction par cas peut être effectué pour les fonctions par cas sur les entiers. On fait alors apparaître une fonction de traitement par cas `nat_case` qui a le type suivant :

```
nat_case:∀F:nat→Set, F 0 →(∀m:nat, F (S m))→∀n:nat, F n
```

Cette fonction peut être définie en *Coq* de la façon suivante :

```
Definition nat_case
  (F:nat→Set)(exp1:F O)(exp2:∀p:nat, F (S p))(n:nat) :=
  match n as x return F x with
  | O ⇒ exp1
  | S p ⇒ exp2 p
  end.
```

Récurseurs à types dépendants

Nous pouvons maintenant allier filtrage dépendant et récursion pour comprendre la forme que doit prendre le type du récurseur le plus général possible associé à chaque type dépendant. D'abord, le premier argument ne doit plus être un type fixe mais une fonction f allant du type inductif considéré vers une sorte `Set`, `Prop`, ou `Type`.

Ensuite, pour chaque constructeur on va devoir fournir une expression dont le type est construit en fonction des arguments de ce constructeur. Si le nom du constructeur est c et les arguments sont $a_1 : t_1, \ldots, a_k : t_k$ alors l'expression associée à ce constructeur pourra utiliser tous les a_i, mais également toutes les valeurs correspondant à des appels récursifs sur les a_i qui sont dans le type inductif considéré. Cette expression sera donc représentée par une fonction de type "$\forall (b_1 : t'_1) \cdots (b_l : t'_l),\ c\ b_{i_1} \ldots b_{i_k}$" où l est k plus le nombre d'indices i tels que t_i contient une instance du type considéré. Chacun des arguments a_i est associé à l'un des b_{j_i} choisi de la manière suivante :

- $j_1 = 1$ et $t'_1 = t_1$.
- Si t_i ne contient pas d'instance du type inductif considéré, alors $j_{i+1} = j_i + 1$ et $t'_{j_i} = t_i$.
- Si t_i est une instance du type inductif considéré, alors $j_{i+1} = j_i + 2$, $t'_{j_i} = t_i$, et $t_{j_i+1} = (f\ b_{j_i})$.
- Si t_i est un type de fonction de la forme $\forall (c_1 : \tau_1) \cdots (c_m : \tau_m),\ \tau$ où τ est une instance du type inductif considéré, alors $j_{i+1} = j_i + 2$, $t_{j_i} = t_i$ et $t_{j_i+1} = \forall (c_1 : \tau_1) \cdots (c_m : \tau_m),\ f\ (b_{j_i}\ c_1 \cdots c_m)$.

Observons le fruit de ce procédé de construction sur le type des entiers naturels. Le récurseur commence par prendre un premier argument *f :nat → Set*. Ensuite il faut fournir deux valeurs, la première doit être de type *(f O)*, ce qui est facile à obtenir car le constructeur O ne prend pas d'arguments. Pour le second constructeur on a seulement un argument, disons $a_1 : nat$, nous avons alors $b_1 : nat$ et $b_2 : (f\ b_1)$ et l'expression complète a le type

$$\forall\,(b_1\texttt{:nat})(b_2\texttt{:(f }b_1\texttt{))},\ \texttt{f (S }b_1\texttt{)}.$$

Si nous choisissons différement les noms de variables liées et tenons compte des produits non dépendants cette expression peut s'écrire :

```
∀n:nat, f n → f (S n)
```

En réunissant les différents éléments de ce récurseur, appelé *nat_rec*, on trouve le type suivant :

```
nat_rec :
  ∀f:nat→Set, f 0 →(∀n:nat, f n → f (S n))→ ∀n:nat, f n.
```

Ce récurseur est construit automatiquement au moment de la définition inductive de `nat` dans le système *Coq*. Cette construction est équivalente à la définition suivante :

```
Fixpoint nat_rec (f:nat→Set)(exp1:f 0)
 (exp2:∀p:nat, f p → f (S p))(n:nat){struct n} : f n :=
  match n as x return f x with
  | O ⇒ exp1
  | S p ⇒ exp2 p (nat_rec f exp1 exp2 p)
  end.
```

La fonction `nat_rec` peut être utilisée à la place de la commande `Fixpoint` pour définir la majeure partie des fonctions récursives. Par exemple la fonction de multiplication d'un entier naturel par 2 peut être décrite de la manière suivante :

```
Definition mult2' :=
  nat_rec (fun n:nat ⇒ nat) 0 (fun p v:nat ⇒ S (S v)).
```

En revanche, on rencontrera des difficultés pour définir des fonctions exhibant une récurrence à pas multiple, comme la fonction `div2` décrite en section 10.3.1.

La fonction *nat_rec* et le principe de récurrence *nat_ind* ont pratiquement le même type, sauf que *Set* dans le type de *nat_rec* est remplacé par *Prop* dans le type de *nat_ind*. On peut ainsi pousser l'isomorphisme de Curry-Howard entre preuves et programmes un cran plus loin : le principe de récurrence est effectivement construit par le même procédé que le récurseur : c'est une fonction qui permet de construire pour tout n une preuve de $(P\ n)$ par un calcul récursif sur n. Cette fonction aurait pu être décrite par la définition suivante :

```
Fixpoint nat_ind (P:nat→Prop)(exp1:P 0)
 (exp2:∀p:nat, P p → P (S p))(n:nat){struct n} : P n :=
  match n as x return P x with
  | O ⇒ exp1
  | S p ⇒ exp2 p (nat_ind P exp1 exp2 p)
  end.
```

Comme deuxième exemple, nous pouvons étudier la construction du récurseur pour les arbres binaires à valeur entière introduits en section 7.3.4 page 195. Ici encore le récurseur prend en argument une fonction `f:Z_btree→Set`, puis les valeurs correspondant à chaque constructeur. Pour le premier constructeur, sans argument, la valeur doit simplement être de type `(f Z_leaf)`. Pour le second constructeur, il y a trois arguments que nous pouvons nommer a_1 : `Z`, a_2 : `Z_btree`, a_3 : `Z_btree`. Nous avançons donc progressivement dans ces arguments de la manière suivante :

1. $j_1 = 1$ et b_1 doit avoir le type `Z`,

2. $j_2 = 2$ et b_2 doit avoir le type `Z_btree`,
3. Comme `Z_btree` est le type inductif considéré, $j_3 = 4$ et b_3 doit avoir le type “ (b_2 ”,
4. L'argument b_4 a le type `Z_btree`,
5. Comme `Z_btree` est le type inductif considéré, il doit y avoir un argument b_5 de type “ $f\ b_4$ ”

Le type de l'expression associée à ce constructeur est alors :

```
∀(b1:Z)(b2:Z_btree)(b3:f b2)
    (b4:Z_btree)(b5:f b4), f (Z_bnode b1 b2 b4)
```

En tenant compte des arguments non dépendants, et en renommant les variables liées pour améliorer la lisibilité ce type est également :

```
∀(z:Z)(t1:Z_btree), f t1 →
    ∀t2:Z_btree, f t2 → f (Z_bnode z t1 t2)
```

En réunissant le tout, le récurseur pour les arbres binaires a le type suivant :

```
Z_btree_rec :
  ∀f:Z_btree→Set,
   f Z_leaf →
   (∀(n:Z)(t1:Z_btree), f t1 →
      ∀t2:Z_btree, f t2 → f (Z_bnode n t1 t2))→
   ∀t:Z_btree, f t.
```

Ce récurseur est construit automatiquement par le système *Coq* à la définition du type inductif. Cette construction automatique fournit un résultat identique à la définition ci-dessous :

```
Fixpoint Z_btree_rec (f:Z_btree→Set)(exp1:f Z_leaf)
 (exp2:∀(n:Z)(t1:Z_btree), f t1 →
        ∀t2:Z_btree, f t2 → f (Z_bnode n t1 t2))
 (t:Z_btree){struct t} : f t :=
  match t as x return f x with
  | Z_leaf ⇒ exp1
  | Z_bnode n t1 t2 ⇒
      exp2 n t1 (Z_btree_rec f exp1 exp2 t1) t2
        (Z_btree_rec f exp1 exp2 t2)
  end.
```

Ici encore le récurseur a pratiquement le même type que le principe de récurrence et nous pouvons voir ce principe comme une fonction construisant des preuves relatives aux arbres par un calcul récursif sur ces arbres.

Comme troisième exemple, considérons le type d'arbres binaires fonctionnels introduit en section 7.3.5.1 page 198. Le deuxième constructeur est

```
Z_fnode:Z→(bool→Z_fbtree)→Z_fbtree
```

nous avons alors $t_1 = $ `Z` et $t_2 = $ `bool→Z_fbtree`. La construction de l'expression associée à ce constructeur suit le schéma suivant :

1. $j_1 = 1$ et $t'_1 = $ `Z`
2. $j_2 = 2$ et $t'_2 = $ `bool→Z_fbtree`,
3. Comme t_2 contient le type `Z_fbtree` $t'_3 = $ `(`c_1`:bool)(`f `(`b_2 c_1`))`

Le type de l'expression associée à ce constructeur est alors :

```
∀(b1:Z)(b2:bool→Z_fbtree)(b3:∀c1:bool, f (b2 c1)),
       f (Z_fnode b1 b2)
```

En tenant compte des arguments non dépendants, et en renommant les variables liées pour améliorer la lisibilité ce type est également :

```
∀(z:Z)(g:bool→Z_fbtree), (∀b:bool, f (g b))→ f (Z_fnode z g)
```

Nous arrêtons ici notre description, bien qu'elle reste incomplète, puisque nous n'avons pas décrit les récurseurs pour les types inductifs dépendants. Le lecteur intrigué pourra se reporter aux travaux de Christine Paulin sur ce sujet [72, 73]. Notons toutefois que cette section nous a montré la relation intime qui existe entre le type d'un récurseur dépendant et le principe de récurrence associé au type inductif.

Exercice 15.1 Décrire la fonction d'addition sur les entiers naturels sans utiliser `Fixpoint` ou `fix` mais à l'aide de `nat_rec`.

Exercice 15.2 * Donner la définition de `Z_btree_ind` à l'aide de la construction `fix`.

15.1.5 Principes de récurrence des propriétés inductives

Le principe de récurrence engendré par défaut pour un prédicat inductif (de sorte `Prop`) est différent du principe de récurrence engendré pour le type de données inductif jumeau de sorte `Set`. En effet, le principe de récurrence associé à un prédicat inductif est simplifié pour exprimer la non-pertinence des preuves.

Dans la suite de cette section, nous allons décrire les différences entre *principes de récurrence maximaux* et *principes de récurrence simplifiés*. Le principe de récurrence maximal est le principe de récurrence construit suivant la technique décrite dans la section 15.1.3. Le principe de récurrence simplifié est le principe de récurrence construit par défaut dans le système *Coq* pour les types inductifs de sorte `Prop`.

Les types inductifs de sortes `Prop` sont habituellement des types dépendants. Lorsque l'on considère un type inductif dépendant avec n arguments non-paramétriques, le principe de récurrence maximal quantifie sur un prédicat à $n+1$ arguments, car l'argument de rang $n+1$ est dans le type considéré et les n autres arguments sont nécessaires pour construire un terme bien formé. Pour le principe de récurrence simplifié, on utilise un prédicat à n arguments : l'argument de rang $n+1$ est tout simplement oublié.

Le principe de récurrence simplifié est obtenu en instanciant le principe maximal pour un prédicat à $n+1$ arguments qui oublie le dernier et fait référence au prédicat à n arguments.

Par exemple, le principe de récurrence maximal pour le prédicat inductif `le` devrait avoir le type suivant :

∀ (n:nat)(P:∀ n0:nat, n ≤ n0 → Prop),
P n (le_n n)→(∀ (m:nat)(l:n ≤ m), P m l → P (S m)(le_S n m l))→
∀ (n0:nat)(l:n ≤ n0), P n0 l

L'instanciation de ce type pour le prédicat

```
End redefine_well_founded_induction.
```

donne le type que nous connaissons pour le principe de récurrence usuel des prédicats inductifs, ce que nous pouvons vérifier par l'expérience suivante :

```
Eval compute in
  (∀ (n:nat)(P:nat→Prop),
     (fun (n':nat)(P:∀m:nat, n' ≤ m → Prop) ⇒
        P n' (le_n n')→
        (∀ (m:nat)(h:n' ≤ m), P m h → P (S m)(le_S n' m h))→
        ∀ (m:nat)(h:n' ≤ m), P m h) n
       (fun (m:nat)(_:n ≤ m) ⇒ P m)).
   = ∀ (n:nat)(P:nat→Prop),
       P n →(∀ m:nat, n ≤ m → P m → P (S m))→
       ∀ m:nat, n ≤ m → P m
   : Prop
```

Comme nous l'avons indiqué en section 15.1.4, le principe de récurrence maximal peut toujours être obtenu en utilisant la commande `Fixpoint` dans la déclaration suivante :

```
Fixpoint even_ind_max (P:∀n:nat, even n → Prop)
 (exp1:P 0 O_even)
 (exp2:∀ (n:nat)(t:even n),
         P n t → P (S (S n))(plus_2_even n t))(n:nat)(t:even n)
 {struct t} : P n t :=
  match t as x0 in (even x) return P x x0 with
  | O_even ⇒ exp1
  | plus_2_even p t' ⇒
    exp2 p t' (even_ind_max P exp1 exp2 p t')
  end.
```

Le principe de récurrence simplifié peut être obtenu par la déclaration suivante. Encore une fois, ce principe de récurrence est une fonction dont l'argument est une preuve de propriété inductive et dont le résultat est une conséquence de cette propriété.

```
Fixpoint even_ind' (P:nat→Prop)(exp1:P 0)
 (exp2:∀n:nat, even n → P n → P (S (S n)))(n:nat)(t:even n)
 {struct t} : P n :=
  match t in (even x) return P x with
  | O_even ⇒ exp1
  | plus_2_even p t' ⇒ exp2 p t' (even_ind' P exp1 exp2 p t')
  end.
```

Le principe de récurrence simplifié pour le prédicat `clos_trans` (section 9.1.1, page 243) peut être reproduit avec la déclaration suivante :

```
Fixpoint clos_trans_ind' (A:Set)(R P:A→A→Prop)
 (exp1:∀x y:A, R x y → P x y)
 (exp2:∀x y z:A,
        clos_trans A R x y →
        P x y → clos_trans A R y z → P y z → P x z)(x y:A)
 (p:clos_trans A R x y){struct p} : P x y :=
 match p in (clos_trans _ _ x x0) return P x x0 with
 | t_step x' y' h ⇒ exp1 x' y' h
 | t_trans x' y' z' h1 h2 ⇒
   exp2 x' y' z' h1 (clos_trans_ind' A R P exp1 exp2 x' y' h1)
        h2 (clos_trans_ind' A R P exp1 exp2 y' z' h2)
  end.
```

Exercice 15.3 ** Construire manuellement le principe de récurrence simplifié pour le prédicat `sorted` donné en section 9.1.

15.1.6 La commande `Scheme`

Dans le système *Coq*, le principe de récurrence simplifié est engendré automatiquement pour toutes les définitions de prédicats inductifs. Le principe de récurrence maximal peut être obtenu, soit en fournissant la construction manuellement comme nous l'avons fait pour `even_ind_max`, soit en appelant la commande `Scheme`, par exemple avec la commande suivante :

```
Scheme even_ind_max := Induction for even Sort Prop.
```

Le mot-clef `Induction` indique que c'est un principe d'induction maximal que l'on veut obtenir. Ce mot-clef peut être remplacé par `Minimality` pour obtenir un principe d'induction simplifié.

La command `Scheme` peut aussi être utilisée pour produire des récurseurs simplifiés, comme le récurseur `nat_simple_rec` que nous avons vue en section 15.1.4.

```
Scheme nat_simple_rec := Minimality for nat Sort Set.
```

Exercice 15.4 * Une autre technique pour prouver le théorème `le_2_n_pred` donné en page 10.2.4 est d'utiliser le principe de récurrence maximal pour la proposition inductive "`le`". Effectuer cette démonstration.

15.2 *** Filtrage et récursion sur des preuves

À une exception près, les règles de bonne formation des expressions de filtrage interdisent que l'on construise un terme de sorte `Set` en effectuant un filtrage sur des expressions de sorte `Prop`. La raison pour cette restriction est que l'on veut maintenir le principe de non pertinence des preuves : la seule chose qui compte lorsque l'on dispose d'une preuve ne devrait être que son existence, mais pas sa forme. Si l'on était en mesure de distinguer entre plusieurs preuves d'un même énoncé pour obtenir des données différentes à l'aide de la construction de filtrage, alors ce principe serait violé.

15.2.1 Restriction sur le filtrage

Si l'on cherche à construire une fonction de type $(x : A)(P\ x)\to(T\ x)$, où $(P\ x)$ est de sorte `Prop` et $(T\ x)$ est de sorte `Set`, alors la restriction sur le filtrage rend très difficile de construire une fonction récursive par rapport à une preuve de $(P\ x)$. La solution usuelle est de construire une fonction récursive par rapport à x, de déterminer la valeur du résultat dans chaque cas, et d'utiliser des inversions sur $(P\ x)$ pour en tirer les conséquences utiles au calcul.

Pour illustrer ce problème, nous considérons une spécification riche de la soustraction des entiers naturels utilisant la définition suivante :

```
Definition rich_minus (n m:nat) := {x : nat | x+m = n}.

Definition le_rich_minus : ∀m n:nat, n ≤ m → rich_minus m n.
```

Cette construction ne peut se faire par élimination de l'hypothèse "`le n m`", mais nous pouvons utiliser un raisonnement par récurrence sur les arguments de type `nat`.

```
induction m.
intros n Hle; exists 0.
...
Hle : n ≤ 0
============================
 0+n = 0
```

À ce stade de la construction, le but est de sorte `Prop` et il est licite de faire un traitement par cas sur l'hypothèse `H`, ce que fait la tactique `inversion`. Cette tactique retourne un seul but trivial.

```
inversion Hle; trivial.
```

La preuve continue avec le cas de récurrence :

```
intros n; case n.
intros Hle; exists (S m).
...
IHm : ∀ n:nat, n ≤ m → rich_minus m n
```

```
  n : nat
  Hle : 0 ≤ S m
  ============================
   S m + 0 = S m
```

Ici encore le but est de sorte `Prop` et nous pouvons construire une preuve sans contrainte.

```
 auto with arith.
 intros n' Hle.
  ...
  IHm : ∀ n:nat, n ≤ m → rich_minus m n
  n : nat
  n' : nat
  Hle : S n' ≤ S m
  ============================
   rich_minus (S m)(S n')
```

Ici, la conclusion du but est de sorte `Set` et l'hypothèse `IHm` est également de sorte `Set`, un traitement par cas sur une instance est licite :

```
 elim (IHm n').
 intros r Heq.
 exists r.
 rewrite <- Heq; auto with arith.
```

À cette étape, le but est à nouveau de sorte `Prop` et la tactique d'inversion est à nouveau utilisable.

```
 inversion Hle; auto with arith.
Defined.
```

Exercice 15.5 ** On considère le prédicat inductif sur les listes polymorphes « u est un préfixe de v » :

```
Set Implicit Arguments.

Inductive lfactor (A:Set) : list A → list A → Prop :=
  lf1 : ∀u:list A, lfactor nil u
| lf2 : ∀(a:A)(u v:list A),
       lfactor u v → lfactor (cons a u)(cons a v).
```

Construire une fonction réalisant la spécification suivante :

```
∀(A:Set)(u v:list A), lfactor u v → {w : list A | v = app u w}
```

15.2.2 Relâchement de la restriction

Lorsqu'une définition inductive présente un seul constructeur et que ce constructeur est une fonction qui ne prend que des preuves en argument, il reste raisonnable d'autoriser un filtrage sur cette définition inductive, puisqu'aucune information de sorte `Set` ne sera obtenue. Dans le cas d'une définition inductive paramétrique, ce sont seulement les arguments non-paramétriques qui sont contraints à habiter un type de sorte `Prop`.

Ainsi, il est remarquable que la possibilité de faire des réécritures soit obtenue par l'application directe du principe de récurrence associé à l'égalité. Nous rappelons que l'égalité est décrite par la définition inductive suivante :

```
Inductive eq (A:Type)(x:A) : A→Prop :=
    refl_equal : eq A x x.
```

Dans cette définition, le constructeur `refl_equal` est une constante si l'on fait abstraction des arguments paramétriques. Le système de type autorise un traitement par cas sur les éléments de ce type même pour construire une valeur dont le type est de sorte `Set`. Ainsi, il est possible de définir la fonction `eq_rect` et d'en déduire une fonction `eq_rec` :

```
Definition eq_rect (A:Type)(x:A)(P:A→Type)(f:P x)(y:A)(e:x = y)
  : P y := match e in (_ = x) return P x with
           | refl_equal ⇒ f
           end.

Definition eq_rec (A:Type)(x:A)(P:A→Set) :
  P x → ∀y:A, x = y → P y := eq_rect A x P.

Implicit eq_rec [A].
```

Ici, la définition de `eq_rec` repose sur l'extension de la notion de convertibilité présentée en section 3.5.2.

La fonction `eq_rec` est utile principalement lorsque l'on utilise des types dépendants. Si la spécification `P:A→Set` exige que la valeur retournée soit égale à `x` pour permettre un bon typage et que l'on dispose d'une preuve de `x=y` et de la valeur `y`, la fonction `eq_rec` peut permettre de montrer que la valeur `y` est acceptable.

Par exemple, on peut envisager un type `A` pour lequel on dispose d'une fonction qui décide l'égalité de deux expressions, un type dépendant `B:A→Set` et une fonction `f:(x:A)(B x)`. Si `a:A` est une valeur pour laquelle on dispose d'une valeur `v:(B a)`, on peut vouloir construire la fonction qui coïncide avec `f` pour tout élément de `A`, sauf en `a`, pour lequel la valeur retournée est `v`. Cette fonction peut s'écrire de la façon suivante :

```
Section update_def.
  Variables (A : Set)(A_eq_dec : ∀x y:A, {x = y}+{x ≠ y}).
  Variables (B : A→Set)(a : A)(v : B a)(f : ∀x:A, B x).
```

```
  Definition update (x:A) : B x :=
    match A_eq_dec a x with
    | left h ⇒ eq_rec A a B v x h
    | right h' ⇒ f x
    end.
End update_def.
```

Raisonner sur la valeur de la fonction `eq_rec` est difficile car on atteint les limites du pouvoir expressif des types inductifs. Pour résoudre ces difficultés, le système *Coq* fournit un axiome dans le module `Eqdep`, appelé `eq_rec_eq`, qui exprime le concept intuitif, c'est à dire que la valeur retournée par `eq_rec` est la même que l'un des arguments de cette fonction :

```
eq_rec_eq
     :∀(U:Set)(p:U)(Q:U→Set)(x:Q p)(h:p = p),
                  x = eq_rec U p Q x p h
```

Exercice 15.6 ** Démontrer que la fonction `update` vérifie le théorème suivant :

```
update_eq
     :∀(A:Set)(eq_dec:∀x y:A, {x = y}+{x ≠ y})
                  (B:A→Set)(a:A)(v:B a)(f:∀x:A, B x),
                  update A eq_dec B a v f a = v.
```

15.2.3 Récursion

On peut également définir des fonctions récursives dont l'argument principal est dans la sorte `Prop` et dont le résultat est dans la sorte `Set`. Dans ce cas, l'argument de sorte `Prop` est une propriété qui assure la terminaison de l'algorithme mais dont la valeur effective n'est pas utilisée pour sélectionner la valeur trouvée.

L'exemple le mieux connu de récursion sur un prédicat inductif pour un calcul de sorte `Set` est l'utilisation de l'accessibilité pour définir les fonctions récursives bien fondées, suivant une notion décrite dans [4, 55]. Cette notion d'accessibilité est décrite par la définition inductive suivante [1] :

```
Inductive Acc (A:Set)(R:A→A→Prop) : A→Prop :=
  Acc_intro : ∀x:A, (∀y:A, R y x → Acc R y)→ Acc R x.
```

Si Φ est le prédicat « ne pas faire partie d'une suite infinie décroissante pour R », il apparaît que la propriété suivante est satisfaite :

$$\forall x.(\forall y.(R\ y\ x)\rightarrow(\Phi\ y)) \leftrightarrow (\Phi\ x),$$

1. Cette définition est effectuée dans le mode d'arguments implicites automatiques.

puisque si x faisait partie d'une suite infinie décroissante, le successeur y dans cette suite ferait également partie d'une suite infinie décroissante. Par conséquent un élément accessible ne peut appartenir à aucune suite infinie décroissante. Ceci montre que ce prédicat d'accessibilité approche de façon constructive la notion d'absence de suite infinie décroissante. Cette notion sera exploitée pour décrire des fonctions récursives générales au prochain chapitre, puisqu'elle permettra d'indiquer que certains calculs ne présentent pas de suites infinis d'appels récursifs.

Si nous disposons d'un élément x, d'une preuve h_x que x est accessible, d'un élément y et d'une preuve $h_r : (R\ y\ x)$, nous pouvons facilement construire une preuve que y est accessible, tout simplement par filtrage ; de plus, la nouvelle preuve obtenue est *structurellement plus petite* que h_x. Les bibliothèques de *Coq* contiennent la fonction `Acc_inv` qui remplit cet office :

```
Print Acc_inv.
Acc_inv =
fun (A:Set)(R:A→A→Prop)(x:A)(H:Acc R x) ⇒
  match H in (Acc _ a) return (∀ y:A, R y a → Acc R y) with
  | Acc_intro x0 H0 ⇒ H0
  end
    : ∀ (A:Set)(R:A→A→Prop)(x:A),
        Acc R x → ∀ y:A, R y x → Acc R y

Arguments A, R, x are implicit
Argument scopes are [type_scope _ _ _ _ _]
```

Ce filtrage est parfaitement licite parce que la valeur retournée est encore de sorte `Prop`.

Puisque `(Acc_inv A R x H y Hr)` est une preuve d'accessibilité structurellement plus petite, nous pouvons l'utiliser dans la construction d'une fonction récursive dont l'argument principal est la preuve d'accessibilité `H`, ce qui se produit dans la définition inductive suivante [2] :

```
Fixpoint Acc_iter (A:Set)(R:A→A→Prop)(P:A→Set)
 (f:∀x:A, (∀y:A, R y x → P y)→ P x)(x:A)(H:Acc R x)
 {struct H} : P x :=
  f x (fun (y:A)(Hr:R y x) ⇒ Acc_iter P f (Acc_inv H y Hr)).
```

Cette fonction récursive est remarquable : elle contient un appel récursif, mais apparemment pas de filtrage. En fait, le filtrage est isolé dans la fonction `Acc_inv` qui est utilisée seulement pour fournir un argument de sorte `Prop` qui est nécessaire pour la récursion.

Ceci n'est possible que parce que `Acc_inv` est définie de façon transparente et *Coq* est capable de vérifier que l'appel récursif respecte les contraintes de la récursion structurelle. Ainsi, nous avons effectivement obtenu une fonction

2. La définition de `Acc_iter` utilise les arguments implicites automatiques.

récursive sur une proposition de sorte `Prop` dont le résultat est une donnée usuelle de sorte `Set`.

Ceci est utilisé pour définir des fonctions *récursives bien-fondées*. Une relation sur un type A est nœtherienne si tout élément du type A est accessible. Toute relation noethérienne est bien fondée : aucun élément de A ne participe à une suite infinie décroissante. Ceci s'exprime par la définition de `well_founded` fournie dans les bibliothèques de *Coq*. Par abus de notation, les bibliothèques de *Coq* utilisent le terme anglais `well_founded` pour désigner une relation nœthérienne. En logique classique ces notions sont équivalentes (voir exercice 16.7).

```
Print well_founded.
well_founded =
fun (A:Set)(R:A→A→Prop) ⇒ ∀ a:A, Acc R a
    : ∀ A:Set, (A→A→Prop)→Prop
Argument A is implicit
Argument scopes are [type_scope _]
```

Lorsqu'une relation est bien fondée, il est possible de définir des fonctions récursives en contrôlant les appels récursifs par cette relation, à l'aide d'une fonction comme la fonction `well_founded_induction` suivante :

```
Definition well_founded_induction (A:Set)(R:A→A→Prop)
  (H:well_founded R)(P:A→Set)
  (f:∀x:A, (∀y:A, R y x → P y)→ P x)(x:A) : P x :=
  Acc_iter P f (H x).
```

Les fonctions `Acc_iter` et `well_founded_induction` données dans le système *Coq* peuvent être différentes des fonctions fournies ici, mais avec le même type. L'utilisation de ces fonctions pour définir des fonctions récursives générales sera détaillée dans la section 16.2.

La forme de récursion sur des propriétés décrite dans cette section pourra être utilisée à volonté pour des fonctions retournant des valeurs de sorte `Set` si l'on prend bien garde d'isoler les filtrages sur les propriétés comme cela a été fait ici à l'aide de la fonction `Acc_inv`. Nous donnons un exemple supplémentaire en section 16.4.

15.2.4 Élimination forte

L'élimination forte correspond à la possibilité de construire une expression par filtrage dont le type est de sorte `Type`. Ce type d'élimination est utilisé dans la tactique `discriminate` pour démontrer que deux constructeurs différents sont distinguables (voir section 7.2.3).

Pour l'étude de structures mathématiques, principalement les structures algébriques telles que *groupes*, or *anneaux*, il peut être utile de regrouper dans une même structure, représentée comme un type inductif, le type des éléments du groupe, les opérations de ce groupe et les propriétés de ces opérations. Ceci se fait généralement à l'aide de la commande `Record`, en construisant un type de sorte `Type`. Pour cet ouvrage, nous avons utilisé cette possibilité en section 14.10.1 page 413. Par exemple, on pourra définir une notion abstraite de groupe commutatif avec la commande suivante :

```
Record group : Type :=
  {A : Type;
   op : A→A→A;
   sym : A→A;
   e : A;
   e_neutral_left : ∀x:A, op e x = x;
   sym_op : ∀x:A, op (sym x) x = e;
   op_assoc : ∀x y z:A, op (op x y) z = op x (op y z);
   op_comm : ∀x y:A, op x y = op y x}.
```

Cette définition n'est pas imprédicative, car le type `group` est de sorte `Type`. Le système *Coq* acceptera donc de construire la fonction `A:group→Set`, dont le type est bien de sorte `Type`.

15.2.4.1 Quelques restrictions

Une difficulté peut apparaître à l'usage parce que l'élimination forte n'est pas fournie pour tous les types inductifs. En particulier, elle n'est pas fournie pour les types de sorte `Prop` parce que cela contredirait le principe de non-pertinence des preuves.

Elle n'est pas fournie pour certains types inductifs de sorte `Set`, ceux dont certains constructeurs sont imprédicatifs (dans la littérature sur le Calcul des Constructions ces types inductifs sont appelés des « grands » types).

Les règles de formation des types telles que nous les avons décrites dans le tableau 5.4 page 120 autorisent les constructeurs de types inductifs définis dans la sorte `Set` à être imprédicatifs, c'est-à-dire qu'un constructeur peut comporter une quantification sur tous les types de sorte `Set` alors même que l'on est en train de définir un type de sorte `Set`. L'ennui est qu'une telle construction prépare le terrain pour un paradoxe connu depuis la fin du dix-neuvième siècle : le paradoxe de Burali-Forti [3]. L'étude de ce paradoxe dans le cadre de la théorie des types a été effectuée par Girard [47] et par Coquand [27].

Le paradoxe de Burali-Forti se résume de la façon suivante : on peut associer à toute relation bien fondée un ordinal. Il existe une relation d'inclusion entre les ordres qui est bien fondée. L'ordinal associé à cette relation bien fondée doit être supérieur à tous les ordinaux pour cette relation bien fondée. Il doit donc être supérieur à lui-même, ce qui contredit le fait que cette relation d'inclusion est bien fondée. Le point clef de ce paradoxe est que l'on raisonne sur un type (le type des ordinaux) qui peut potentiellement contenir des éléments (distinguables) correspondant à tous les types de la sorte de ce type. En particulier, il contient un élément correspondant à un ordre sur lui-même. C'est en cherchant à distinguer cet élément (par exemple en énonçant qu'il est supérieur à tous les autres) que l'on crée les conditions du paradoxe.

Lorsque l'on construit un type inductif de sorte `Set` avec un constructeur imprédicatif, on autorise la construction d'un élément de ce type pour chaque type de sorte `Set`. Si l'on autorise également de distinguer les différents éléments du type en fonction de l'argument de sorte `Set` fourni pour le constructeur imprédicatif on indique que le constructeur est injectif. Une injection de la sorte `Set` vers un type inclus dans `Set` posera naturellement un problème [3].

Nous pouvons soumettre au système *Coq* une définition de la structure de groupe où `Type` est remplacé par `Set`. Les messages d'erreurs produits montrent que cette structure de groupe sera pratiquement inutilisable.

```
Record group : Set :=
  {A : Set;
   op : A→A→A;
   sym : A→A;
   e : A;
   e_neutral_left : ∀x:A, op e x = x;
   sym_op : ∀x:A, op (sym x) x = e;
   op_assoc : ∀x y z:A, op (op x y) z = op x (op y z);
   op_comm : ∀x y:A, op x y = op y x}.
```

...
Warning: A cannot be defined because it is large and group' is not.
Warning: op cannot be defined because the projections A, A, A haven't.

De manière générale, le caractère imprédicatif de `Set` est très controversé et cette particularité est susceptible de disparaître dans les versions futures de *Coq*.

Une étude formelle du paradoxe de Burali-Forti, décrit comme un développement *Coq* et mis au point par Bruno Barras, Thierry Coquand et Benjamin Werner, est disponible dans les contributions des utilisateurs accessibles depuis le site internet du système *Coq*, sous le titre `paradoxes`.

15.3 Types mutuellement inductifs

Le système *Coq* fournit la possibilité de définir des types mutuellement inductifs. Il s'agit de permettre à au moins deux types définis inductivement de faire référence l'un à l'autre et réciproquement.

15.3.1 Arbres et forêts

Un exemple typique est celui où l'on définit un type d'arbres où les nœuds peuvent avoir un nombre arbitraire et toujours fini de descendants. On peut obtenir ce résultat en disant que chaque nœud porte une liste d'arbres.

```
Inductive ntree (A:Set) : Set :=
```

3. En théorie des ensembles, Cantor a montré qu'il n'était pas possible de disposer d'une injection de l'ensemble des parties d'un ensemble vers cet ensemble : l'ensemble des parties est toujours beaucoup plus « grand ».

```
  nnode : A → nforest A → ntree A
with nforest (A:Set) : Set :=
  nnil : nforest A | ncons : ntree A → nforest A → nforest A.
```

Dans cette définition le type `ntree` fait référence au type `nforest` et le type `nforest` fait référence au type `ntree`. C'est en ce sens que l'on a défini deux types mutuellement inductifs.

Pour calculer et raisonner sur ces types mutuellement inductifs, le système *Coq* fournit le moyen de construire des fonctions structurellement récursives mutuelles. Par exemple la fonction qui compte le nombre de nœuds d'un arbre peut-être écrite de la façon suivante :

```
Open Scope Z_scope.

Fixpoint count (A:Set)(t:ntree A){struct t} : Z :=
  match t with
  | nnode a l ⇒ 1 + count_list A l
  end
 with count_list (A:Set)(l:nforest A){struct l} : Z :=
  match l with
  | nnil ⇒ 0
  | ncons t tl ⇒ count A t + count_list A tl
  end.
```

Il faut noter que le principe de récurrence engendré par défaut dans le système *Coq* ne tient pas compte de la structure mutuellement récursive des types :

```
ntree_ind :
  ∀(A:Set)(P:ntree A → Prop),
    (∀(a:A)(l:nforest A), P (nnode A a l))→
    ∀t:ntree A, P t.
```

Ce principe de récurrence ne prend pas en compte le fait que la liste qui apparaît comme composante peut contenir des sous-termes pour lesquels on pourrait utiliser une hypothèse de récurrence. En fait, ce principe de récurrence est pratiquement inutile. Le principe de récurrence associé par défaut au type `nforest` est également simplifié et souvent insuffisant.

Des principes de récurrence plus complets peuvent être obtenus en utilisant la commande `Scheme`, déjà vue en section 15.1.6.

```
Scheme ntree_ind2 :=
   Induction for ntree Sort Prop
 with nforest_ind2 :=
   Induction for nforest Sort Prop.
```

Cette commande engendre deux principes de récurrence qui sont nommés comme la commande l'indique (ici `ntree_ind2` et `nforest_ind2`). Observons principalement le premier :

ntree_ind2
  $: \forall$ *(A:Set)(P:ntree A* $\to$ *Prop)(P0:nforest A* $\to$ *Prop),*
    ($\forall$ *(a:A)(n:nforest A), P0 n* $\to$ *P (nnode A a n))*$\to$
    P0 (nnil A)$\to$
    ($\forall$ *n:ntree A,*
      P n $\to$ $\forall$ *n0:nforest A, P0 n0* $\to$ *P0 (ncons A n n0))*$\to$
    $\forall$ *n:ntree A, P n*

Ce principe de récurrence quantifie sur deux prédicats, `P` un prédicat sur le type `ntree` et `P0`, une prédicat sur le type `nforest`. Le prédicat `P` est utilisé pour les expressions de type `ntree` et le prédicat `P0` est utilisé pour les expressions de type `nforest`. Ensuite on trouve trois prémisses principales correspondant aux trois constructeurs des deux types, enfin l'épilogue exprime que le prédicat `P` est satisfait pour tous les arbres de type `ntree`. Le principe de récurrence pour les forêts a la même forme, seulement l'épiloque change. Nous voyons plus tard une preuve qui utilise se principe de récurrence.

Les définitions inductives mutuelles peuvent aussi s'utiliser pour des propositions, par exemple nous pourrons construire les propositions `occurs` et `occurs_forest` de la façon suivante :

```
Inductive occurs (A:Set)(a:A) : ntree A → Prop :=
  occurs_root : ∀l, occurs A a (nnode A a l)
| occurs_branches :
   ∀b l, occurs_forest A a l → occurs A a (nnode A b l)
with occurs_forest (A:Set)(a:A) : nforest A → Prop :=
  occurs_head :
   ∀t tl, occurs A a t → occurs_forest A a (ncons A t tl)
| occurs_tail :
   ∀t tl,
    occurs_forest A a tl → occurs_forest A a (ncons A t tl).
```

Ici aussi, il pourra être nécessaire d'engendrer les bons principes de récurrence à l'aide de la commande `Scheme` pour raisonner par récurrence sur ces types inductifs.

15.3.2 Démonstration par récurrence mutuelle

Pour comprendre l'utilisation de principes de récurrence mutuels, nous allons démontrer un petit théorème sur les arbres de type `ntree`. Pour exprimer ce théorème, nous allons définir deux autres fonctions sur les types `ntree` et `nforest` qui calculent la somme de toutes les valeurs portées dans un arbre de type `(ntree Z)` et dans une liste d'arbres de type `(nforest Z)`.

```
Fixpoint n_sum_values (t:ntree Z) : Z :=
  match t with
  | nnode z l ⇒ z + n_sum_values_l l
  end
```

```
with n_sum_values_l (l:nforest Z) : Z :=
 match l with
 | nnil ⇒ 0
 | ncons t tl ⇒ n_sum_values t + n_sum_values_l tl
 end.
```

Nous pouvons maintenant démontrer que si toutes les valeurs qui apparaissent dans un arbre sont supérieures à 1, alors la somme de toutes ces valeurs est supérieure au nombre de nœuds dans l'arbre.

```
Theorem greater_values_sum :
 ∀t:ntree Z,
   (∀x:Z, occurs Z x t → 1 ≤ x)→ count Z t ≤ n_sum_values t.
```

Ici nous voulons démontrer cette propriété par récurrence sur `t`. Puisque cette variable est dans le type `(ntree Z)` et qu'il n'y a qu'un seul constructeur, cette preuve par récurrence ne donnera qu'un seul cas insolvable si nous utilisons le principe de récurrence `ntree_ind`. En revanche, nous verrons bien apparaître une récurrence sur la structure de l'arbre si nous utilisons une récurrence basée sur le principe `ntree_ind2`. L'utilisation de ce principe de récurrence est délicate, car nous devons indiquer comment s'instancie la variable `P0` :

```
Proof.
 intros t; elim t using ntree_ind2 with
  (P0 := fun l:nforest Z ⇒
           (∀x:Z, occurs_forest Z x l → 1 ≤ x)→
           count_list Z l ≤ n_sum_values_l l).
```

La valeur que nous avons donnée pour `P0` est simplement la propriété jumelle de la propriété que nous voulons prouver. Cette propriété est obtenue en remplaçant `ntree` par `nforest`, `count` par `count_list`, etc.

La preuve par récurrence fournit trois buts. Le premier est plus lisible une fois que l'on a introduit les différentes hypothèses dans le contexte et que l'on a simplifié l'expression à démontrer par les différentes règles de conversion. Nous utilisons la tactique `lazy` plutôt que `simpl` car l'expansion de toutes les fonctions récursives mène à un terme illisible.

```
intros z l Hl Hocc; lazy beta iota delta -[Zplus Zle];
 fold count_list n_sum_values_l.
```

...
t : ntree Z
z : Z
l : nforest Z
Hl : (∀ x:Z, occurs_forest Z x l → 1 ≤ x)→
    count_list Z l ≤ n_sum_values_l l
Hocc : ∀ x:Z, occurs Z x (nnode Z z l)→ 1 ≤ x
============================
 1 + count_list Z l ≤ z + n_sum_values_l l

Ici nous cherchons un théorème qui permet de décomposer la comparaison sur les deux sommes en une comparaison sur les termes de la somme. La commande suivante permet de trouver un théorème adapté :

```
SearchPattern (_ + _ ≤ _ + _).
```

...

Zplus_le_compat: $\forall$ *n m p q:Z, n* $\leq$ *m* $\rightarrow$ *p* $\leq$ *q* $\rightarrow$ *n+p* $\leq$ *m+q*

Ce théorème est bien adapté pour une utilisation avec `apply`, il décompose le but en deux buts qui sont prouvés aisément à l'aide de l'hypothèse `Hocc` et des constructeurs de la proposition inductive `occurs`. Dans le deuxième but engendré par la preuve par récurrence, il s'agit de vérifier que la propriété recherchée sur les listes d'arbres est bien satisfaite lorsque la liste est vide.

```
auto with *.
...
t : ntree Z
============================
 (∀ x:Z, occurs_forest Z x (nnil Z)→ 1 ≤ x)→
 count_list Z (nnil Z) ≤ n_sum_values_l (nnil Z)
```

```
auto with zarith.
```

Le but suivant est le troisième but engendré par la preuve par récurrence, il s'agit de vérifier le cas de récurrence sur les listes d'arbres, en utilisant une hypothèse de récurrence sur les arbres et une hypothèse de récurrence sur les listes d'arbres. Ce but est plus lisible après l'introduction des variables et hypothèses dans le contexte et la simplification des fonctions `count_list` et `n_sum_values_l` :

```
intros t1 Hrec1 tl Hrec2 Hocc;
 lazy beta iota delta -[Zplus Zle];
 fold count count_list n_sum_values n_sum_values_l.
...
Hrec1 : (∀ x:Z, occurs Z x t1 → 1 ≤ x)→
        count Z t1 ≤ n_sum_values t1
tl : nforest Z
Hrec2 : (∀ x:Z, occurs_forest Z x tl → 1 ≤ x)→
        count_list Z tl ≤ n_sum_values_l tl
Hocc : ∀ x:Z, occurs_forest Z x (ncons Z t1 tl)→ 1 ≤ x
============================
 count Z t1 + count_list Z tl ≤ n_sum_values t1 + n_sum_values_l tl
```

Ici encore le théorème `Zle_plus_plus` est le mieux adapté et les buts qu'il produit sont faciles à démontrer.

15.3.3 *** Arbres et listes d'arbres

Un défaut important des types inductifs `ntree` et `nforest` est que le type `nforest` est un type de liste spécialisé pour ne contenir que des arbres. Ainsi, toutes les fonctions déjà définies sur les listes vont devoir être programmées de nouveau pour ce type `nforest` et leurs propriétés re-démontrées. On peut éviter cela en utilisant simplement le type des listes polymorphes, ce qui mène à la définition suivante :

```
Inductive ltree (A:Set) : Set :=
    lnode : A → list (ltree A)→ ltree A.
```

Cette déclaration est acceptée par *Coq*, mais les principes de récurrence qui sont construits automatiquement ne permettent pas de faire de calcul récursif sur les sous termes apparaissant dans la liste d'arbres fournie en deuxième argument du constructeur `lnode`, ce que nous pouvons vérifier avec la commande `Check` :

```
Check ltree_ind.
```

ltree_ind
: ∀ (A:Set)(P:ltree A → Prop),
(∀ (a:A)(l:list (ltree A)), P (lnode A a l))→
∀ l:ltree A, P l

Il est possible de construire un principe de récurrence mieux adapté en utilisant la commande `Fixpoint`, comme nous l'avons fait dans la section 15.1.4, mais il est nécessaire d'utiliser une construction `fix` imbriquée pour que la définition soit acceptée par *Coq* comme une définition bien formée. Cette imbrication permet d'assurer que les appels récursifs ont bien lieu sur des termes structurellement plus petits.

```
Section correct_ltree_ind.
Variables
  (A : Set)(P : ltree A → Prop)(Q : list (ltree A)→ Prop).

Hypotheses
 (H : ∀ (a:A)(l:list (ltree A)), Q l → P (lnode A a l))
 (H0 : Q nil)
 (H1 : ∀t:ltree A, P t →
      ∀l:list (ltree A), Q l → Q (cons t l)).


Fixpoint ltree_ind2 (t:ltree A) : P t :=
  match t as x return P x with
  | lnode a l ⇒
      H a l
        (((fix l_ind (l':list (ltree A)) : Q l' :=
             match l' as x return Q x with
             | nil ⇒ H0
             | cons t1 tl ⇒ H1 t1 (ltree_ind2 t1) tl (l_ind tl)
             end)) l)
  end.
End correct_ltree_ind.
```

Dans cette expression, appelons « point fixe interne » l'expression

```
fix l_ind ... end.
```

La variable `l` est reconnue comme un sous-terme structurel de `t`, `l'` est également reconnue comme un sous-terme structurel de `t` parce que c'est un sous-terme de `l` par application du point fixe interne, la variable `t1` est naturellement reconnue comme un sous-terme de `l'` et c'est donc un sous-terme structurel de `l` et de `t`. Ainsi l'application de `ltree_ind2` à `t1` se fait bien sur un sous-terme strict de `t`.

Cette technique pour construire un bon principe de récurrence pour cette structure de données fournit également le principe pour écrire de véritables fonctions récursives sur les arbres de type `ltree` et les listes d'arbres de ce type. Alors que la commande `Scheme` donnait simultanément un principe de récurrence sur les arbres et les forêts, nous allons devoir également construire un principe de récurrence sur les listes d'arbres, cette fois-ci en imbriquant une fonction récursive sur les arbres à l'intérieur de la définition d'une fonction récursive sur les listes d'arbres.

Exercice 15.7 ** Construire le principe de récurrence `list_ltree_ind2` qui permet un véritable raisonnement par récurrence sur les listes d'arbres.

Exercice 15.8 Définir la fonction `lcount` de type "$\forall$ A :Set, ltree A $\to$ nat," qui compte le nombre de nœuds dans un arbre de type `ltree`.

Exercice 15.9 ** Définir la fonction `ltree_to_ntree` qui traduit les arbres d'un type dans l'autre en respectant la structure. Définir ensuite la fonction `ntree_to_ltree` et démontrer que les deux fonctions sont les inverses l'une de l'autre.

Chapitre 16

* Récursivité générale

La récursivité structurelle est puissante, surtout grâce à l'ordre supérieur, comme nous l'avons vu avec l'exemple sur la fonction d'Ackermann. En revanche, elle n'est pas toujours adaptée pour décrire des algorithmes dont la terminaison ne se justifie pas par la décroissance structurelle de l'un des arguments de l'algorithme considéré.

Plusieurs méthodes permettent de contourner cette difficulté. La première consiste à ajouter un argument artificiel à la fonction définie, et à s'assurer que la décroissance structurelle se fera sur cet argument. En pratique cela revient à se contraindre à calculer une forme de complexité avant d'appeler la fonction, puis à exécuter cette fonction avec l'argument de complexité qui décroît strictement à chaque appel récursif de la fonction.

La seconde méthode consiste à faire apparaître une relation « bien fondée », qui ne contient donc pas de chaîne infinie décroissante et à montrer que les arguments successifs des appels récursifs sont sur une chaine décroissante pour cette relation, ce qui assure assez simplement que la récursion ne peut pas boucler indéfiniment. Cette méthode est plus difficile à mettre en œuvre car elle fait intervenir plusieurs concepts assez complexes, mais elle permet de fabriquer des fonctions dont le code extrait est plus fidèle aux intentions du programmeur.

L'un des défauts importants de la seconde méthode est qu'il est difficile de raisonner sur les fonction obtenues par cette méthode lorsqu'elle sont faiblement spécifiées. Nous décrivons une troisième méthode qui permet de contourner cette difficulté. Cette méthode repose en partie sur la deuxième méthode mais permet d'obtenir une *équation de point fixe* qui est utile pour les raisonnements ultérieurs sur la fonction considérée.

Nous décrivons enfin une quatrième méthode qui repose sur la définition d'un prédicat inductif particulièrement conçu pour décrire le domaine de définition de la fonction considérée. La définition de la fonction récursive se fait alors par récurrence structurelle sur les preuves de ce prédicat.

Pour chacune de ces méthodes nous donnons un exemple de développement.

16.1 Récursion bornée

Il est possible de définir des fonctions récursives en ajoutant un argument artificiel, dont le seul intérêt est de rendre la fonction récursive structurelle par rapport à cet argument. L'argument supplémentaire sera typiquement un nombre entier, servant de borne pour indiquer le nombre maximal d'appels récursifs. Par nature, la fonction ainsi définie effectue un traitement par cas sur cet argument artificiel. Lorsque le cas de base est atteint, une valeur par défaut est calculée.

Par exemple, la fonction de division par soustractions successives peut être définie de la façon suivante :

```
Fixpoint bdiv_aux (b m n:nat){struct b} : nat*nat :=
  match b with
  | O ⇒ (0, 0)
  | S b’ ⇒
      match le_gt_dec n m with
      | left H ⇒
          match bdiv_aux b’ (m-n) n with
          | pair q r ⇒ (S q, r)
          end
      | right H ⇒ (0, m)
      end
  end.
```

Pour construire une fonction de division bien spécifiée, nous devons maintenant démontrer que la fonction `bdiv_aux` calcule les bonnes valeurs si la borne de récursion est assez grande. Il faut donc savoir préciser ce que l'on entend par « assez grande ». Ici, avec un diviseur supérieur ou égal à 1 nous savons que le nombre d'appels récursifs, comme le quotient, est nécessairement inférieur au dividende. Nous pouvons décomposer les propriétés à démontrer en deux parties, étudions la démonstration de la première :

```
Theorem bdiv_aux_correct1 :
 ∀b m n:nat, m ≤ b → 0 < n →
 m = fst (bdiv_aux b m n) * n + snd (bdiv_aux b m n).
```

La démonstration de ce théorème se fait assez facilement, en utilisant notre premier principe guide (voir page 200), par récurrence sur l'argument principal, c'est à dire la borne.

```
Proof.
 intros b; elim b; simpl.
 intros m n Hle; inversion Hle; auto.
 intros b’ Hrec m n Hleb Hlt; case (le_gt_dec n m); simpl; auto.
 intros Hle; generalize (Hrec (m-n) n);
  case (bdiv_aux b’ (m-n) n); simpl; intros q r Hrec’.
 rewrite <- plus_assoc; rewrite <- Hrec’; auto with arith.
```

L'étape " `rewrite <- Hrec'` " crée un but qui demande de vérifier que la borne est encore bien choisie pour l'appel récursif. Ce but a la forme suivante :

...
Hleb : m ≤ S b'
Hlt : 0 < n
Hle : n ≤ m
...
============================
m-n ≤ b'

Ce but se résout automatiquement avec la tactique `omega`. Nous laissons le lecteur construire une démonstration de la deuxième partie, que nous prenons simplement comme hypothèse pour la suite de notre exposé :

```
Hypothesis bdiv_aux_correct2 :
 ∀b m n:nat, m ≤ b → 0 < n → snd (bdiv_aux b m n) < n.
```

Maintenant, nous pouvons construire une fonction de division répondant à une spécification raisonnable, c'est-à-dire qui ne prend que deux arguments numériques et un argument de preuve assurant la précondition que le diviseur est non nul. Il est seulement nécessaire de fournir une fonction qui calcule la borne avant d'appeler la fonction auxiliaire récursive. Ici, le calcul de la borne se réduit à sa plus simple expression, puisque cette borne est l'un des arguments naturels de la fonction.

Nous utilisons une construction par preuve, pour séparer le contenu calculatoire du contenu logique de cette fonction. Nous nous reposons sur la tactique `refine` déjà décrite dans la section 10.2.7.

```
Definition bdiv :
  ∀m n:nat, 0 < n → {q:nat &{r:nat | m = q*n+r ∧ r < n}}.
 refine
 (fun (m n:nat)(h:0 < n) ⇒
    let p := bdiv_aux m m n in
    existS (fun q:nat ⇒ {r : nat | m = q*n+r ∧ r < n})
      (fst p)(exist _ (snd p) _)).
 unfold p; split.
 apply bdiv_aux_correct1; auto.
 intros; eapply bdiv_aux_correct2; eauto.
Defined.
```

Les fonctions à récursion bornée présentent un léger inconvénient, car elles contiennent des calculs supplémentaires relatifs à la borne et il est nécessaire de déterminer cette borne avant d'appeler la fonction principale. Ces calculs supplémentaires impliquent une distance entre la fonction effectivement étudiée dans les preuves et l'algorithme visé. Ceci peut être gênant si l'objectif est de fournir l'étude formelle d'un algorithme. De plus, les calculs supplémentaires seront conservés par le procédé d'extraction que nous étudions dans le chapitre 11.

En revanche, les calculs effectués par cette fonction peuvent être effectués par le moteur de réduction de *Coq*, ce que l'on mettra à profit dans les tactiques basées sur la réflexion que nous étudierons dans le chapitre 17. On pourra par exemple calculer effectivement une division en appelant la commande d'évaluation `Eval` comme c'est visible ici pour la division de 2000 par 31. Le temps de calcul observé est justifié par le fait que l'on effectue une réduction symbolique du calcul des constructions plutôt qu'un calcul direct de division.

```
Time Eval lazy beta iota zeta delta in (bdiv_aux 2000 2000 31).
     = (64, 16) : (nat*nat)%type
Finished transaction in 1. secs (0.73u,0.02s)
```

La fonction `bdiv` peut aussi être utilisée, car nous avons pris garde de la définir comme une fonction transparente (en utilisant la commande `Defined`). Néanmoins, il faut lui fournir une preuve que le diviseur est supérieur à zéro et prendre soin d'extraire les valeurs numériques du résultat ; il est aussi judicieux de demander à *Coq* d'effectuer les calculs de façon paresseuse pour éviter de passer du temps à construire un terme de preuve qui sera ensuite oublié [1].

```
Time Eval lazy beta delta iota zeta in
  match bdiv 2000 31 (lt_O_Sn 30) with
    existS q (exist r h) ⇒ (q,r)
  end.
     = ((64),(16)) : (nat*nat)%type
Finished transaction in 0 secs (0.03u,0.s)
```

Nous avons également essayé la commande suivante où seule la stratégie change, mais nous n'avons pas reçu de réponse, même après plusieurs heures de calcul.

```
Time Eval compute in
  match bdiv 2000 31 (lt_O_Sn 30) with
    existS q (exist r h) ⇒ (q,r)
  end.
```

Exercice 16.1 Démontrer l'hypothèse `bdiv_aux_correct2`.

Exercice 16.2 * Écrire une variante bien spécifiée de la fonction `bdiv_aux`.

Exercice 16.3 ** L'algorithme de fusion de deux listes ordonnées est particulièrement adapté pour trier des listes. Sa structure peut être résumée par les égalités suivantes (dans les deux cas on suppose que $a \leq b$) :

```
merge (cons a l)(cons b l') = cons a (merge l (cons b l'))
merge (cons b l)(cons a l') = cons a (merge (cons b l) l')
```

1. Les temps décrits ici ont été calculé à l'aide d'un processeur `Intel Pentium II` cadencé à 400 MHz.

Cette fonction n'est récursive structurelle ni par rapport à la première liste (à cause de la deuxième équation) ni par rapport à la deuxième liste (à cause de la première équation), mais la somme des longueurs des deux listes décroît dans les deux équations. Construire un algorithme qui effectue le tri par fusion des listes dont les éléments sont de types `A`, en travaillant dans une section dont voici l'en-tête :

```
Section merge_sort.
 Variables (A : Set)(Ale : A→A→Prop)
   (Ale_dec : ∀x y:A, {Ale x y}+{Ale y x}).
```

Écrire une fonction `merge` à trois arguments, dont les deux premiers sont de type "`list A`" et le troisième est un nombre naturel par rapport auquel la fonction est récursive structurelle, la valeur calculée étant encore une liste d'éléments de type `A`.

Exercice 16.4 ** *Calcul de racine carrée.* Étant donné x, il s'agit de calculer simultanément les nombres s et r tels que $s^2 \leq x < (s+1)^2$ et $x = s^2 + r$. L'algorithme procède en calculant d'abord la racine carrée de x divisé par 4 : si (q, r_0) est le résultat de la division de x par 4, si s' est la racine carrée entière de q et si r' est le reste $q - s'^2$, deux cas peuvent alors se présenter :

1. si $4r' + r_0 \geq 4s' + 1$ alors $s = 2s' + 1$ et $r = 4r' + r_0 - 4s' - 1$,
2. sinon $s = 2s'$ et $r = 4r' + r0$.

Construire la fonction par récursion bornée qui effectue ce calcul. Démontrer qu'elle est correcte : l'utiliser pour définir la fonction

```
sqrt_nat
   : ∀n:nat, {s:nat &{r:nat | n = s*s+r ∧ n < (s+1)*(s+1)}}
```

16.2 ** Fonctions récursives bien fondées

16.2.1 Relations bien fondées

La notion de relation bien fondée est définie à l'aide de la notion d'accessibilité que nous avons décrite dans la section 15.2.3 page 437. Rappelons l'interprétation intuitive de l'accessibilité. Pour une relation binaire R sur un type A, le prédicat d'accessibilité caractérise l'ensemble des éléments de A par lesquels ne passe aucune suite infinie décroissante pour la relation R, c'est à dire qu'il n'existe pas de suite u_n telle que $u_0 = x$ et "$R\ u_{j+1}\ u_j$" pour tout j. Ce prédicat est défini inductivement, et nous avons déjà montré qu'il était possible de définir des fonctions récursives dont l'argument principal est une preuve de ce prédicat pour un élément arbitraire de A. Intuitivement, si la fonction est appelée sur un élément x de A et une preuve que x est accessible, alors les appels récursifs de la fonction ne peuvent avoir lieu que sur des éléments y tels que "$R\ y\ x$" soit satisfait. Comme la suite d'appels récursifs ne peut pas être infinie, on est certain que la réduction de la fonction va terminer.

Une relation bien fondée est une relation pour laquelle tous les éléments sont accessibles. Le système *Coq* fournit des outils pour utiliser les relations bien fondées, principalement sous la forme d'un récurseur et d'un principe de récurrence. Pour utiliser ces outils il faut connaître quelques relations bien fondées et savoir en construire de nouvelles. Il peut être nécessaire de démontrer manuellement que les éléments d'un ensemble sont accessibles, mais il est également possible d'utiliser quelques relations bien fondées existantes et de construire de nouvelles relations bien fondées à partir d'autres, en utilisant quelques théorèmes généraux. Les deux prochaines sections illustrent ces possibilités.

16.2.2 Preuves d'accessibilité

Pour A et R donnés, une preuve d'accessibilité de x est donc construite à partir d'une preuve d'accessibilité de tous les prédécesseurs de x par R. Mais comment démarrer ? On peut remarquer que tous les éléments minimaux (sans prédécesseur par R) sont trivialement accessibles : la preuve se fait par élimination de la proposition — fausse — "$R\ x\ y$". De la même façon, pour les éléments non minimaux x, on doit prouver à partir des propriétés de la relation R que tous les prédécesseurs de x sont accessibles.

Illustrons ce propos en prouvant que tous les entiers naturels sont accessibles pour la relation d'ordre strict `lt`. La stratégie est alors claire : on prouve ce résultat par récurrence : le cas 0 se traite par une simple inversion de la proposition (fausse) "`y < 0`". Si l'on suppose n accessible, alors son successeur l'est aussi ; en effet, une simple technique d'inversion permet de montrer que si $y < n + 1$, alors :

– soit $y < n$ et donc — comme n est accessible — y l'est aussi,
– soit $y = n$, et — par réécriture de y en n — y est accessible.

Ce raisonnement fait appel à une étape que l'on retrouvera systématiquement : le prédécesseur pour la relation bien fondée d'un élément accessible est forcément accessible. Cette étape est décrite dans les bibliothèques de *Coq* par le théorème suivant :

Acc_inv
  : ∀ (A:Set)(R:A→A→Prop)(x:A),
    Acc R x → ∀ y:A, R y x → Acc R y

Nous montrons ci-dessous le script complet de cette preuve :

```
Require Import Lt.

Theorem lt_Acc : ∀n:nat, Acc lt n.
Proof.
 induction n.
 split; intros p H; inversion H.
 split.
 intros y H0.
 case (le_lt_or_eq _ _ H0).
```

```
 intro; apply Acc_inv with n; auto with arith.
 intro e; injection e; intro e1; rewrite e1; assumption.
Qed.
```

On en déduit immédiatement que la relation `lt` est bien fondée :

```
Theorem lt_wf : well_founded lt.
Proof.
 exact lt_Acc.
Qed.
```

Ce théorème est présent dans le module `Wf_nat` de la bibliothèque standard de *Coq*.

De manière générale, la relation qui associe à tout élément d'un type inductif ses sous-termes directs est une relation bien fondée, et ceci se démontre aisément par une récurrence structurelle. Par exemple, considérons le type `positive` présenté en section 7.3.4 ; la relation liant tout terme à ses sous-termes stricts se définit ainsi :

```
Inductive Rpos_div2 : positive→positive→Prop :=
  Rpos1 : ∀x:positive, Rpos_div2 x (xO x)
| Rpos2 : ∀x:positive, Rpos_div2 x (xI x).
```

La démonstration que cette relation est bien fondée tient dans les quelques lignes suivantes, qui pourraient également s'utiliser aisément pour de nombreux autres types récursifs.

```
Theorem Rpos_div2_wf : well_founded Rpos_div2.
Proof.
 unfold well_founded; intros a; elim a;
  (intros; apply Acc_intro; intros y Hr; inversion Hr; auto).
Qed.
```

16.2.3 Construction de relations bien fondées

Le module `Wellfounded` des bibliothèques de *Coq* fournit une collection de théorèmes pour construire de nouvelles relations bien fondées à partir de relations existantes. Citons pour mémoire le théorème `wf_clos_trans` qui exprime que la clôture transitive d'une relation bien fondée est aussi bien fondée :

```
Check wf_clos_trans.
```

wf_clos_trans
: ∀ (A:Set)(R:relation A),
well_founded R → well_founded (clos_trans A R)

et le théorème `wf_inverse_image` qui indique comment construire une relation bien fondée sur un type `A` à partir d'une relation bien fondée sur un type `B` et d'une fonction de `A` dans `B` :

```
Check wf_inverse_image.
```

wf_inverse_image
: ∀ (A B:Set)(R:B→B→Prop)(f:A→B),
well_founded R → well_founded (fun x y:A => R (f x)(f y))

Par exemple, nous avons utilisé ce dernier théorème dans la section 11.2.2.5 page 337 pour construire une relation bien-fondée sur le type associé à l'état d'un programme impératif, où deux états σ_1 et σ_2 sont reliés si la valeur associée à la variable `x` dans l'état σ_1 est positive mais strictement inférieure à la valeur de la même variable dans l'état σ_2.

Il est aussi aisé de construire de nouvelles relations sur des produits ou des sommes disjointes de types. La bibliothèque `Wellfounded` contient les théorèmes exprimant que ces constructions préservent le caractère bien-fondé des relations.

Exercice 16.5 Considérer le type des arbres binaires dont les sommets sont étiquetés par un type quelconque A : `Set`, et définir la relation « t est le sous-arbre gauche ou le sous-arbre droit de t' » ; démontrer que cette relation est bien fondée.

16.2.4 Récursion bien fondée

La récursion sur des preuves d'accessibilité fournit une nouvelle méthode pour definir des fonctions récursives (voir section 15.2.3 pour en comprendre les fondements théoriques). Quand la relation considérée est bien fondée, tout élément est accessible et la notion d'accessibilité peut masquée au programmeur. Le récurseur `well_founded_induction` et le principe de récurrence `well_founded_ind` remplissent cet office.

16.2.5 Le récurseur `well_founded_induction`

Voici le type de `well_founded_induction` :

well_founded_induction
: ∀ (A:Set)(R:A→A→Prop),
well_founded R →
∀ P:A→Set,
(∀ x:A, (∀ y:A, R y x → P y)→ P x)→
∀ a:A, P a

Il peut sembler difficile à lire, mais une bonne explication de texte nous permettra de comprendre comment ce récurseur peut permettre de construire une fonction récursive générale. Ce récurseur prend six arguments, mais nous préférerons dire qu'elle prend cinq arguments et retourne une fonction de type ∀`x:A,P x`. Étudions séparément les arguments :

1. Le premier argument est `A`, de type `Set`, c'est le domaine de la fonction que l'on cherche à définir.
2. Le second argument est `R` de type `A→A→Prop`, c'est une relation binaire.

3. Le troisième argument est de type "`well_founded A R`", c'est une preuve que la relation `R` est bien fondée.
4. Le quatrième argument est `P`, de type `A→Set`, c'est une fonction utilisée pour décrire le type de la valeur calculée. En effet, nous pourrons utiliser `well_founded_induction` pour définir des fonctions ayant un type dépendant.
5. Le cinquième argument est lui-même une fonction prenant deux arguments :
   (a) le premier argument est un élément `x` de `A`,
   (b) le second argument est une fonction de type dépendant, qui prend deux arguments : le premier est un élément `y` de `A` et le second est une preuve que `y` et `x` sont reliés par `R`, la valeur calculée par cette fonction est de type "`P y`". Le type un peu particulier de cette fonction indique que cette fonction ne peut être appelée que sur des valeurs reliées avec `x` par la relation `R`, c'est-à-dire en quelque sorte sur des valeurs *plus petites* que `x`. Cette fonction représente les appels récursifs de la fonction en cours de définition, le fait que la relation `R` soit bien fondée décrit bien que l'on assure ainsi qu'il n'y aura pas de chaîne infinie d'appels récursifs.

   La valeur calculée par la fonction donnée en cinquième argument est de type `(P x)`, cette fonction décrit donc bien le procédé de calcul de la valeur pour `x` en fonction des valeurs possibles de la fonction récursive elle-même, mais en s'autorisant seulement à utiliser ces valeurs pour des arguments plus petits que `x` vis-à-vis de la relation `R`.

Les deux exercices suivants sont des applications de l'induction bien fondée, représentée par `well_founded_ind`*; dans les deux cas, le lecteur devra considérer un prédicat Q :* `A→Prop` *bien choisi ; une preuve (par induction bien fondée) de la proposition "*`∀a:A,` Q `a`*" doit permettre de conclure le théorème demandé.*

Exercice 16.6 Montrer le théorème suivant (la constante `inclusion` est définie dans le module `Relations`) :

```
Lemma wf_inclusion :
 ∀(A:Set)(R S:A→A→Prop),
   inclusion A R S → well_founded S → well_founded R.
```

Exercice 16.7 ** Montrer que si la relation R sur A est bien fondée, alors il n'existe pas de suite infinie « décroissante » dans A. Autrement dit, prouver l'énoncé ci-dessous :

```
Theorem not_decreasing :
 ∀(A:Set)(R:A→A→Prop),
   well_founded R →
   ~(∃seq:nat→A | (∀i:nat, R (seq (S i))(seq i))).
```

En déduire que les deux relations `le:nat→nat→Prop` et `Zlt : Z→Z→Prop` ne sont pas bien fondées.

Pensez-vous pouvoir démontrer la réciproque de `not_decreasing` ? Pourquoi ? Et en logique classique (en chargeant le module `Classical`) ?

16.2.6 Division euclidienne bien fondée

Étudions maintenant l'utilisation de `well_founded_induction` pour définir une fonction récursive simple. Nous prendrons la fonction de division euclidienne, car il est assez simple d'en décrire la spécification et d'en construire l'algorithme. La division euclidienne sur les nombres naturels est une fonction qui prend en argument deux nombres naturels `m` et `n`, tels que `n` > 0 et calcule un couple de nombres `q` et `r`, tels que la propriété suivante est satisfaite :

$$\mathtt{m} = \mathtt{q} \times \mathtt{n} + \mathtt{r} \ \wedge \ \mathtt{r} < n.$$

L'algorithme que nous allons décrire procède par soustractions successives, à chaque fois que l'on soustrait `n` de `m` on ajoute une unité au quotient. Si `n` est supérieur à `m`, alors on a terminé : le quotient est `0` et le reste est `m`. La relation bien fondée utilisée est l'ordre $<$ sur les entiers naturels. Lorsque l'on charge le module `Wf_nat` dans le système *Coq* on dispose d'un théorème `lt_wf` qui exprime que cet ordre est bien fondé. Nous aurons besoin d'un théorème qui exprime qu'un nombre naturel m peut se décomposer comme la somme d'un autre nombre n et de $m - n$, lorsque n est plus petit que m :

```
Check le_plus_minus.
```
le_plus_minus : ∀ n m:nat, n ≤ m → m = n+(m-n)

Nous utiliserons aussi le théorème d'associativité de l'addition :

```
Check plus_assoc.
```
plus_assoc : ∀ n m p:nat, n+(m+p) = n+m+p

Passons en revue les arguments de `well_founded_induction` pour définir la division euclidienne par soustractions successives :

1. le type de départ est `nat`,
2. la relation est `lt`,
3. la preuve que `lt` est bien fondée est le théorème `lt_wf` (fourni dans le module `Wf_nat`),
4. la fonction calculant le type de la valeur retournée est

   ```
   fun m:nat ⇒
    ∀n:nat, 0 < n → {q:nat &{r:nat | m = q*n+r ∧ r < n}}.
   ```

   Pour rendre notre définition plus lisible, nous allons donner un nom à cette fonction et à ses fragments :

```
Definition div_type (m:nat) :=
  ∀n:nat, 0 < n → {q:nat &{r:nat | m = q*n+r ∧ r < n}}.

Definition div_type' (m n q:nat) :=
  {r:nat | m = q*n+r ∧ r < n}.

Definition div_type'' (m n q r:nat) := m = q*n+r ∧ r < n.
```

5. La fonction calculant la valeur pour un `m` arbitraire doit avoir le type suivant

   `∀x:nat, (∀y:nat, y < x → div_type y)→ div_type x.`

   Cette fonction est déjà trop complexe pour que nous essayions d'en donner le texte en construisant directement le terme du Calcul des Constructions. Nous allons plutôt utiliser une construction par preuve en utilisant la tactique `refine` que nous avons décrite dans la section 10.2.7.

```
Definition div_F :
  ∀x:nat, (∀y:nat, y < x → div_type y) → div_type x.
unfold div_type at 2.
refine
  (fun m div_rec n Hlt ⇒
     match le_gt_dec n m with
     | left H_n_le_m ⇒
         match div_rec (m-n) _ n _ with
         | existS q (exist r H_spec) ⇒
             existS (div_type' m n)(S q)
               (exist (div_type'' m n (S q)) r _)
         end
     | right H_n_gt_m ⇒
         existS (div_type' m n) 0
            (exist (div_type'' m n 0) m _)
     end); unfold div_type''; auto with arith.
```

   Ici, la tactique `refine` devrait engendrer quatre buts, mais certains sont résolus automatiquement par la tactique `auto with arith` qui est placée juste derrière. En fait il ne reste qu'un seul but, qui correspond à la vérification que la spécification est bien satisfaite par les données fournies dans la première clause du traitement par cas sur la comparaison entre `m` et `n`.

   ...
   m : nat
   div_rec : $\forall$ *y:nat,* $y < m \rightarrow$ *div_type y*
   n : nat
   Hlt : $0 < n$
   H_n_le_m : $n \leq m$
   q : nat
   s : $\{r : nat \mid m\text{-}n = q*n+r \wedge r < n\}$

r : nat
*H_spec : m-n = q*n+r* $\wedge$ *r* $<$ *n*
```
============================
```
$m = S\ q\ *\ n + r \wedge r < n$

Ce but est assez aisé à résoudre, par exemple avec les tactiques suivantes :

```
 elim H_spec; intros H1 H2; split; auto.
 rewrite (le_plus_minus n m H_n_le_m); rewrite H1; ring_nat.
Qed.
```

La description de la fonction de division se termine avec la définition suivante :

```
Definition div :
  ∀m n:nat, 0 < n → {q:nat &{r:nat | m = q*n+r ∧ r < n}} :=
  well_founded_induction lt_wf div_type div_F.
```

Il est généralement assez difficile de raisonner directement sur les fonctions définies par récursion bien fondée. Les raisonnements sur les fonctions récursives reposent toujours sur un raisonnement par récurrence sur l'argument principal. Mais ici l'argument principal est une preuve d'accessibilité, et les théorèmes d'accessibilité sont habituellement masqués dans la démonstration que la relation choisie est bien fondée et cette démonstration est souvent sauvegardée de façon opaque ce qui interdit tout raisonnement.

Si l'on arrive à se placer dans un contexte où toutes les démonstrations d'accessibilité sont transparentes, il faut utiliser le principe de récurrence maximal pour cette propriété inductive, fourni dans les bibliothèques de *Coq* sous le nom de `Acc_inv_dep`. Une solution alternative est d'utiliser une équation de point fixe associée à définition inductive et démontrer d'abord cette équation de point fixe, comme c'est décrit dans [6]. Une autre méthode pour obtenir une fonction récursive et l'équation de point fixe associée est de mettre en œuvre la technique de « récursion par itération » que nous décrivons dans la section 16.3.

Il est souvent préférable de construire du premier coup une fonction bien spécifiée par son type. À ce titre, la fonction `div` est définie avec une bonne spécification.

Nous verrons en section 16.3 une technique permettant de définir des fonctions récursives générales, en s'approchant du style de programmation des langages fonctionnels usuels. Cette technique permettra en outre d'obtenir une équation de point fixe qui permettra de faciliter les démonstrations de propriétés, même si la fonction construite n'est pas bien spécifiée.

Exercice 16.8 * Nous reprenons la suite de fibonacci décrite dans l'exercice 10.8 page 10.8. Démontrer les théorèmes suivants :

$$\begin{aligned}
\forall n : nat.\ n > 0 \Rightarrow u_{2n} &= {u_n}^2 + {u_{n-1}}^2 \\
\forall n : nat.\ n > 0 \Rightarrow u_{2n+1} &= {u_n}^2 + 2u_n u_{n-1} \\
\forall n : nat.\ n > 0 \Rightarrow u_{2n+2} &= 2{u_n}^2 + 2u_n u_{n-1} + {u_{n-1}}^2
\end{aligned}$$

En déduire une fonction qui calcule les valeurs u_n et u_{n+1} avec un nombre logarithmique d'appels (par exemple, pour calculer u_{17} et u_{18}, on ne devrait calculer que u_7 et u_8 puis u_2 et u_3).

Exercice 16.9 * En considérant que "`two_power` n" correspond à l'expression 2^n (voir exercice 7.17), définir la fonction de logarithme discret à base 2 ayant le type suivant :

```
log2:
 ∀n:nat, n ≠ 0 →
        {p : nat | two_power p ≤ n ∧ n < two_power (p + 1)}.
```

Exercice 16.10 Retrouver les ordres bien fondés fournis dans les bibliothèques de *Coq* pour les nombres entiers (type `Z`). Les utiliser pour définir la fonction qui coïncide avec la fonction factorielle sur les nombres entiers positifs et qui vaut 0 ailleurs.

Exercice 16.11 ** Construire une fonction de calcul de racine carrée utilisant l'algorithme déjà décrit dans l'exercice 16.4. Cette fonction devra avoir le type suivant :

```
sqrt_nat':
∀n:nat, {s:nat &{r:nat | n = s*s+r ∧ n ≤ (s+1)*(s+1)}}.
```

Exercice 16.12 ** En s'inspirant de l'exercice 16.11, construire une fonction de calcul de racine cubique.

16.2.7 Récursion imbriquée

On parle de récursion imbriquée lorsque la fonction est définie par une équation de la forme suivante :

$$f(x) = \cdots f(g(f(y))) \cdots$$

Cette forme de fonction pose régulièrement des problèmes pour la formalisation, car il est difficile d'établir que ces fonctions sont bien définies partout.

La fonction `well_founded_induction` est bien adaptée pour définir des fonctions récursives imbriquées. Il suffit que la fonction soit définie avec une spécification assez forte pour exprimer que les arguments de chaque appel récursif seront plus petits que l'argument initial. Nous allons illustrer cette remarque sur une fonction conçue pour servir d'exemple, définie par les équations suivantes, où les divisions sont des divisions par défaut :

$$f(0) = 0 \tag{16.1}$$

$$f(x+1) = 1 + f(\frac{x}{2} + f(\frac{x}{2})) \tag{16.2}$$

La méthode que nous proposons est de montrer que cette fonction est bien définie en démontrant en même temps qu'elle calcule toujours une valeur inférieure ou égale à son argument. Nous procédons comme pour les autres définitions de fonctions récursives bien fondées, en fournissant la fonction qui devra être passée en argument au récurseur. Nous utilisons la fonction `div2` décrite dans la section 10.3.1, et le théorème `div2_le`, et quelques hypothèses que la lectrice pourra démontrer en exercice :

```
Hypothesis double_div2_le : ∀x:nat, div2 x + div2 x ≤ x.

Hypothesis f_lemma :
  ∀x v:nat, v ≤ div2 x → div2 x + v ≤ x.

Hint Resolve div2_le f_lemma double_div2_le.

Definition nested_F :
  ∀x:nat, (∀y:nat, y < x→{v:nat|v ≤ y})→{v:nat | v ≤ x}.
refine
  (fun x ⇒ match x return (∀y:nat, y < x →{v:nat | v ≤ y})→
                          {v:nat | v ≤ x} with
             O ⇒ fun f ⇒ exist _ O _
            | S x' ⇒
             fun f ⇒ match f (div2 x') _ with
                       exist v H1 ⇒
                       match f (div2 x' + v) _ with
                         exist v1 H2 ⇒ exist _ (S v1) _
                       end
                     end
            end); eauto with arith.
Defined.
```

La tactique `refine` engendre quatre buts, dont deux correspondent aux préconditions de décroissance pour les appels récursifs sur les valeurs “`div2 x'`” et “`plus (div2 x') v`,” mais ces préconditions sont faciles à démontrer, en particulier à l'aide de l'hypothèse `H1` qui exprime que la valeur calculée par l'appel récursif sur “`div2 x'`” est plus petite que “`div2 x'`.” Toutes les préconditions sont traitées par la tactique `eauto with arith`.

Pour compléter notre définition, il ne reste qu'à faire appel à la fonction `well_founded_induction` :

```
Definition nested_f :=
  well_founded_induction
    lt_wf (fun x:nat ⇒ {v:nat | v ≤ x}) nested_F.
```

Cet exemple n'est pas entièrement satisfaisant parce que la fonction que nous obtenons n'est pas fortement spécifiée. Les équations 16.1 et 16.2 ne peuvent pas être utilisée pour décrire une spécification forte parce qu'elles nécessiteraient que la fonction soit déjà définie. L'exercice 16.15 est plus satisfaisant. Il montre que nous pouvons définir une fonction récursive imbriquée et fortement spécifiée.

Exercice 16.13 Démontrer les hypothèses `f_lemma` et `double_div2_le`.

Exercice 16.14 * Définir en *Coq* la fonction récursive f_1 qui vérifie les équations suivantes :

$$\begin{array}{rcll} f_1(0) & = & (0) & \\ f_1(1) & = & (0) & \\ f_1(x+1) & = & 1 + f_1(1 + f_1(x)) & \quad (x \neq 0) \end{array}$$

Exercice 16.15 *** Cet exercice fait suite à l'exercice 9.24 page 269. Définir une fonction d'analyse syntaxique qui satisfait la spécification suivante :

```
∀l:list par, {l':list par &{t:bin | parse_rel l l' t}}.
```

16.3 ** Récursion générale par itération

Les techniques que nous avons décrites jusqu'à présent reposent de façon importante sur les types dépendants et le filtrage dépendant. Il est aussi possible de définir des fonctions récursives générales en *Coq*, lorsque l'on dispose déjà du codage sous forme d'un programme purement fonctionnel dans un langage fonctionnel tel que *ML*, *OCAML*, ou *Haskell*. Nous allons décrire dans cette section une technique pour effectuer ce genre de définition, qui est facile à suivre tant que la fonction ne comporte pas de récursion imbriquée [7]. Cette technique fonctionne en trois étapes :

1. Déterminer la fonctionnelle associée à une fonction récursive,
2. Démontrer la terminaison de la fonction visée,
3. Construire la fonction visée et l'équation de point fixe adaptée.

Cette technique est automatisable et un prototype d'outil la mettant en œuvre est fourni dans les contributions des utilisateurs de *Coq* (module `Recursive-Definition`).

16.3.1 Fonctionnelle associée à une fonction récursive

En général, la définition d'une fonction récursive fait intervenir une expression de la forme suivante :

$$f\ x = expr$$

où f et x sont autorisées à apparaître dans l'expression *expr*. En fait, les noms f et x sont conventionnels, et correspondent donc à des variables liées dans cette définition. La fonctionnelle associée à une définition récursive est simplement la fonction qui à f et x associe l'expression *expr*. C'est une fonction, que nous qualifions de « fonctionnelle » parce qu'elle prend une fonction en argument. Notons F cette fonction. L'équation définissant notre fonction récursive devient alors l'équation suivante :

$$f\ x = F\ f\ x$$

Lue autrement, cette équation indique que f est un point fixe de F. Des considérations théoriques permettent de préciser cette remarque : la fonction f est en fait le plus petit point fixe de F pour un certain ordre sur l'ensemble des fonctions calculables. De plus F est une fonction continue par construction, car l'expression *expr* ne peut être construite qu'en composant des constructions élémentaires qui sont toutes continues. Le fait que F soit continue indique aussi que si deux fonctions g_1 et g_2 vérifient que g_1 est définie partout où g_2 est définie, alors "$F\ g_1$" est définie partout où "$F\ g_2$" est définie. En sémantique dénotationnelle, il est connu que le plus petit point fixe de la fonction F peut être construit comme la limite lorsque n tend vers l'infini de la suite

$$F^n(\perp)$$

Dans cette suite, le terme $\perp$ est la fonction qui n'est définie nulle part. Cette fonction n'étant pas définissable en *Coq* (qui ne considère que des fonctions totales, voir section 3.2.1), on pourra remplacer sans problème la fonction $\perp$ par une fonction arbitraire et totale $g : A{\rightarrow}B$. Dans la mesure où la fonction f qu'on veut définir est totale, si le terme "$F^k\ \perp\ x$" est défini, il est égal à un terme "$F^k\ g\ x$."

Par exemple, nous pouvons considérer la définition suivante de la division euclidienne :

```
div_it m n =
   match le_gt_dec n m with
   | left _ ⇒ let (q, r) := div_it (m-n) n in (S q, r)
   | right _ ⇒ (0, m)
   end.
```

La fonctionnelle associée peut être construite par la définition suivante :

```
Definition div_it_F (f:nat→nat→nat*nat)(m n:nat) :=
  match le_gt_dec n m with
  | left _ ⇒ let (q, r) := f (m-n) n in (S q, r)
  | right _ ⇒ (0, m)
  end.
```

Preuve de terminaison

Si f est une fonction récursive générale totale et qu'elle est le point fixe de la fonctionnelle F, alors pour tout x, il existe un n tel que pour tout k plus grand que n et pour toute valeur par défaut g on ait l'égalité suivante :

$$f\ x = F^k\ g\ x$$

Donc si la fonction récursive générale est totale, il existe un nombre d'itérations de F qui permet de calculer "$f\ x$." Ceci est donc une preuve que la définition récursive de f définit effectivement une fonction qui termine.

Prise à l'envers, cette remarque permet également de construire f. Montrer que f est bien définie partout, c'est justement montrer qu'on peut associer à toute valeur en entrée x une valeur v telle que "$F^k\ g\ x = v$" ne dépende pas de g (et on peut alors en déduire qu'elle ne dépend pas de k dès qu'il est assez grand). Nous pourrons nous contenter de prouver l'existence de cette valeur. Ainsi on est amené à construire une fonction `f_terminates` spécifiée de la façon suivante :

```
Fixpoint iter (A:Set)(n:nat)(F:A→A)(g:A){struct n} : A :=
  match n with O ⇒ g | S p ⇒ F (iter A p F g) end.

Implicit Arguments iter [A].

Definition f_terminates:
  (n:A)
   {v: B| (Ex [p:nat]
                (k:nat)(g:A→B)(iter (A→B) k F g x)=v)}.
```

Pour construire cette `f_terminates` nous procédons par preuve et cette preuve est basée sur une récurrence bien fondée sur la variable qui décroît à chaque appel. Ensuite la structure de la preuve suit exactement la structure de la fonction `F`.

Par exemple, pour la fonction de division, on écrit de la façon suivante :

```
Definition div_it_terminates :
  ∀n m:nat, 0 < m →
    {v : nat * nat |
     exists p : nat |
      (∀k:nat, p < k → ∀g:nat → nat → nat * nat,
           iter k div_it_F g n m = v)}.
 intros n; elim n using (well_founded_induction lt_wf).
 intros n' Hrec m Hlt.
 caseEq (le_gt_dec m n'); intros H Heq_test.
```

Comme nous avons suivi la structure de la fonctionnelle `div_it_F`, nous avons effectué un traitement par cas sur la valeur de "`le_gt_dec m n'`." Nous n'avons pas effectué ce traitement directement avec la tactique `case`, mais avec la tactique `caseEq` que nous avons définie en section 7.2.7. En effet, nous aurons plusieurs fois besoin de raisonner sur la façon dont ce traitement par cas se réduit. Le premier des deux cas est exprimé par le but suivant :

...
H : m ≤ n'
Heq_test : le_gt_dec m n' = left (m > n') H
============================
*{v:nat*nat |*
∃ p:nat
| (∀ k:nat,
p < k →
*∀ g:nat→nat→nat*nat, iter k div_it_F g n' m = v)}*

Quand `m` est plus petit que `n'`, l'algorithme repose sur un appel récursif de la fonction de division. Ici, cet appel récursif est représenté par une utilisation de l'hypothèse de récurrence.

```
case Hrec with (y := n' - m)(2 := Hlt); auto with arith.
```

L'hypothèse de récurrence retourne un couple quotient-reste et une propriété exprimant que ce couple sera celui fourni par tout calcul contenant assez d'itérations, quelle que soit la fonction fournie pour la fin des itérations. Selon la fonction `div_it_F` la prochaine opération est un traitement par cas sur cette valeur, nous effectuons le même traitement par cas pour obtenir séparément le quotient et le reste qui seront utilisés pour la valeur finale.

```
intros [q r]; intros Hex; exists (S q, r).
```

Il ne nous reste plus qu'à démontrer qu'un nombre suffisant d'itérations permettra de toujours atteindre cette valeur. Notons que le reste de la démonstration à partir de cette étape ne dépend pas de l'algorithme étudié : le schéma de raisonnement se reproduira systématiquement pour toute fonction définie par cette méthode.

La borne inférieure pour ce nombre d'itérations est calculée facilement à partir de la borne inférieure qui était nécessaire pour l'appel récursif : c'est le successeur de cette borne.

```
elim Hex; intros p Heq.
exists (S p).
```

Raisonnons maintenant sur le nombre d'itérations : s'il est nul, il y a une contradiction avec le fait que ce nombre est strictement supérieur à `p`, ce qui s'exprime par les lignes suivantes.

```
intros k.
case k.
intros; elim (lt_n_O (S p)); auto.
```

Si le nombre d'itérations est supérieur à 1, alors la fonction `div_it_F` est exécutée au moins une fois, nous pouvons utiliser l'égalité `Heq` issue de l'hypothèse de récurrence et l'égalité `Heq_test` issue du traitement par cas pour établir l'égalité que nous devons démontrer.

```
intros k' Hplt g; simpl; unfold div_it_F at 1.
rewrite Heq; auto with arith.
rewrite Heq_test; auto.
```

Ici, nous avons terminé le premier cas issu du traitement par cas sur l'expression "`le_gt_dec n' m`," qui est aussi le seul cas contenant un appel récursif.

Notons que les huit dernières lignes du script de preuve (à partir de "`intros Hex`") seraient réutilisées presque sans changement si nous étudiions une autre fonction récursive. Les seules variations seraient dans la tactique

“ `unfold div_it_F 1` ” et dans la réécriture par l'hypothèse `Heq_test`. Ici le code de la fonction ne contient qu'une construction de filtrage et nous avons une hypothèse de la forme de `Heq_test` issue de notre analyse de cette construction de filtrage. Dans le cas général, l'appel récursif pourra se trouver inclus dans plusieurs constructions de filtrage et plusieurs hypothèses `Heq_test` devront être utilisées.

Le filtrage sur l'expression “`le_gt_dec m n'`” fournit un autre cas. Celui-ci ne contient pas d'appel récursif, le calcul de la valeur retournée est plus simple. Le résultat de la division est le couple `(O,p)`, le nombre d'itérations nécessaires pour être sûr que cette valeur est celle retournée par la fonction `div_it_F` itérée est 1, puisqu'il n'y a pas d'appel récursif, et il suffit donc de donner 0 pour la valeur de `p`.

```
exists (0, n'); exists 0; intros k; case k.
intros; elim (lt_irrefl 0); auto.
intros k' Hltp g; simpl; unfold div_it_F at 1.
rewrite Heq_test; auto.
Defined.
```

Dans les cinq tactiques composées qui précèdent seule la valeur `(O,n')` variera si nous étudions une autre fonction par la même méthode.

Nous avons donc réussi à obtenir une fonction `div_it_terminates` qui effectue les calculs attendus et fournit également une preuve qu'elle les fait.

16.3.2 Construction de la fonction cherchée

Il est ensuite possible de construire une fonction dont le type n'exprime que les types de données dans lesquels se trouvent les arguments en entrée et les valeurs en sortie. Pour la division, le type ne peut pas être complètement non dépendant, car la terminaison de la fonction est réellement basée sur le fait que le second argument est non nul : il faut éviter les divisions par 0.

```
Definition div_it (n m:nat)(H:0 < m) : nat*nat :=
  let (v, _) := div_it_terminates n m H in v.
```

16.3.3 Démonstration de l'équation de point fixe

Il s'agit maintenant de démontrer l'énoncé suivant, qui n'est pas entièrement équivalent à l'énoncé donné en section 16.3.1 parce que nous devons tenir compte du fait que la fonction `div_it` n'est bien définie que si le diviseur est non nul :

```
∀(m n:nat)(h:0 < n),
  div_it m n h =
     match le_gt_dec n m with
     | left H ⇒ let (q, r) := div_it (m-n) n h in (S q, r)
     | right H ⇒ (0, m)
     end.
```

Cette démonstration est basée sur la nature même de la fonction `div_it` et la spécification de la fonction `div_it_terminates`. Cette spécification exprime qu'il existe une valeur p pour chaque occurrence de (`div_it` a b c) apparaissant dans l'égalité, telle que pour toute valeur de $k > p$ et toute fonction g on ait :

```
div_it a b c=
  iter k div_it_F g a b c.
```

Deux occurrences de `div_it` apparaissent dans cette égalité. D'après les définitions de `div_it` et `div_it_terminates`, il existe donc deux valeurs p et p' telles que pour toute valeur k simultanément supérieure à p et p' et pour toute fonction g les égalités suivantes soient satisfaites :

```
div_it m n h = iter (S k) div_it_F g m n h

div_it (m-n) n h = iter k div_it_F g (minus m n) n h
```

Nous prenons garde de prendre un nombre d'occurrence supérieur pour le membre de gauche de l'égalité parce que le membre droit correspond déjà à une itération de la fonctionnelle. La démonstration se termine par une utilisation de la réflexivité.

Pour calculer le maximum des variables p, nous utilisons une fonction `max` et les théorèmes la décrivant, dont voici la définition et les preuves :

```
Definition max (m n:nat) : nat :=
  match le_gt_dec m n with left _ ⇒ n | right _ ⇒ m end.

Theorem max1_correct : ∀n m:nat, n ≤ max n m.
intros n m; unfold max; case (le_gt_dec n m); auto with arith.
Qed.

Theorem max2_correct : ∀n m:nat, m ≤ max n m.
intros n m; unfold max; case (le_gt_dec n m); auto with arith.
Qed.
Hint Resolve max1_correct max2_correct : arith.
```

Nous donnons ici l'ensemble du script de démonstration que le lecteur pourra tester et modifier à loisir.

```
Theorem div_it_fix_eqn :
 ∀(n m:nat)(h:(0 < m)),
   div_it n m h =
   match le_gt_dec m n with
   | left H ⇒ let (q,r) := div_it (n-m) m h in (S q, r)
   | right H ⇒ (0, n)
   end.
Proof.
```

```
 intros n m h.
 unfold div_it; case (div_it_terminates n m h).
 intros v Hex1; case (div_it_terminates (n-m) m h).
 intros v' Hex2.
 elim Hex2; elim Hex1; intros p Heq1 p' Heq2.
 rewrite <- Heq1 with
   (k := S (S (max p p')))(g := fun x y:nat ⇒ v).
 rewrite <- Heq2 with (k := S (max p p'))
                      (g := fun x y:nat ⇒ v).
 reflexivity.
 eauto with arith.
 eauto with arith.
Qed.
```

16.3.4 Utilisation de l'équation de point fixe

Maintenant que nous disposons de l'équation de point fixe, il est assez aisé de démontrer que notre fonction de division satisfait la spécification usuelle de la division. Nous ne montrons ici que la première partie de la spécification. Cette démonstration se fait par récurrence bien fondée sur l'argument qui doit décroître entre chaque appel. Après l'étape de récurrence, la démonstration suit la structure de la fonction.

```
Theorem div_it_correct1 :
 ∀(m n:nat)(h:0 < n),
   m = fst (div_it m n h) * n + snd (div_it m n h).
Proof.
 intros m; elim m using (well_founded_ind lt_wf).
 intros m' Hrec n h; rewrite div_it_fix_eqn.
 case (le_gt_dec n m'); intros H; trivial.
 pattern m' at 1; rewrite (le_plus_minus n m'); auto.
 pattern (m'-n) at 1.
 rewrite Hrec with (m'-n) n h; auto with arith.
 case (div_it (m'-n) n h); simpl; auto with arith.
Qed.
```

Exercice 16.16 * Démontrer la deuxième partie de la correction de `div_it` :

```
∀(m n:nat)(h:0 < n), snd (div_it m n h) < n.
```

16.3.5 Discussion

La technique présentée dans cette section permet de travailler séparément sur l'algorithme (représenté par la fonctionnelle `F`), la preuve de terminaison de cet algorithme (représenté par la construction de la fonction `f_terminates`), et la vérification que la fonction satisfait sa spécification, où l'équation de point fixe joue un rôle important. Nous avons donné entièrement les démonstrations dans le cas de la fonction de division par soustractions successives. Nous espérons avoir réussi à convaincre que les démonstrations nécessaires pour prouver la terminaison et l'équation de point fixe peuvent être obtenues systématiquement, voire automatiquement au travers d'une analyse de la structure de la fonctionnelle `F`.

À la réflexion, la technique par itération et la technique de la récursion bornée sont très similaires : les calculs effectués pour évaluer les expressions "`bdiv_aux k n m`" et "`iter k div_it_F (fun n m:nat ⇒(0,0)) m`" sont les mêmes. La démonstration que la fonction répond à la spécification fait même intervenir la propriété que l'argument "`n-m`" lors de l'appel récursif est strictement plus petit que l'argument initial `n`. Ainsi, `n` n'est pas l'argument principal de récursion pour la fonction `bdiv_aux`, mais parmi les arguments secondaires, il joue un rôle privilégié. Toutefois, la similitude s'arrête là, c'est la fonction `div_it_terminates` qui est utilisée pour définir la fonction `div_it` et cette fonction a bien séparé les calculs effectués sur les arguments de terminaison des arguments de calcul effectif de la valeur retournée, ce que ne faisait pas la fonction `bdiv_aux`.

Exercice 16.17 ** Définir la fonction factorielle sur les nombres entiers relatifs (avec la valeur zero pour les arguments négatifs) en utilisant cette méthode et la relation bien fondée `Zwf`.

Exercice 16.18 ** En suivant la technique décrite dans cette section, décrire la fonction `log2` qui satisfait la spécification suivante :

$$\forall n : nat, n > 0 \to \ 2^{(log2\ n)} \leq n < 2 \times 2^{(log2\ n)}$$

16.4 *** Récursion sur un prédicat ad-hoc

Dans la section 15.2.3, nous montrons que la récursion bien fondée repose sur une récursion structurelle sur un prédicat inductif, le prédicat d'accessibilité `Acc`. De manière générale, il est possible de faire reposer la définition d'une fonction sur une récursion sur un prédicat inductif arbitraire. Ana Bove [16] propose même de définir un nouveau prédicat inductif pour chaque fonction que l'on veut définir. Les démonstrations par récurrence vis-à-vis de ce prédicat peuvent ensuite être utilisées pour prouver des propriétés sur la fonction.

Les travaux d'Ana Bove se placent dans une théorie des types différente du Calcul des Constructions inductives et il faut quelques efforts pour les transposer dans notre contexte. Les limitations que nous rencontrons sont les suivantes : si le prédicat ad-hoc est de sorte `Prop` et la fonction doit calculer une valeur de sorte `Set`, la récursion est possible, mais aucune construction de filtrage sur la preuve ne peut être utilisée pour construire une donnée de sorte `Set`. Pourtant les appels récursifs sur la structure de la preuve nécessitent bien l'utilisation de construction de filtrage. Nous isolons ces constructions de filtrage dans des fonctions d'inversions qui sont utilisées seulement au moment de l'appel récursif.

Une compréhension intuitive du prédicat ad-hoc utilisé est que ce prédicat décrit le domaine de définition de la fonction. Comme ce prédicat est distinct du type sur lequel la fonction récursive est définie, cette méthode est particulièrement bien adaptée pour décrire des fonctions partielles.

Pour illustrer cette méthode nous reprenons l'exemple déjà utilisé d'une fonction qui calcule le logarithme discret en base 2 d'un nombre naturel. En *OCAML* cette fonction s'écrirait de la façon suivante :

```
let rec log x = match x with
  S O -> O
| S (S p) -> S (log (S (div2 p)))
```

Cette fonction est visiblement définie pour 1 et si x est de la forme "$S\ (S\ p)$" alors il suffit qu'elle soit définie pour "$div2\ (S\ (S\ p))$" pour être définie pour x. Nous pouvons exprimer ce raisonnement dans une définition inductive :

```
Inductive log_domain : nat→Prop :=
  log_domain_1 : log_domain 1
| log_domain_2 :
    ∀p:nat, log_domain (S (div2 p))→ log_domain (S (S p)).
```

Apparemment, la définition inductive du domaine peut toujours être déduite du texte « attendu » de la fonction, exprimé dans un langage fonctionnel (seulement s'il n'y a pas de récursion imbriquée).

Pour exprimer la fonction dans le Calcul des Constructions, il faut couvrir tous les cas de filtrage sur l'argument de la fonction, mais ici il sera facile de démontrer que zéro n'est pas dans le domaine de définition :

```
Theorem log_domain_non_O : ∀x:nat, log_domain x → x ≠ 0.
Proof.
 intros x H; case H; intros; discriminate.
Qed.
```

Pour chacun des appels récursifs de la fonction, nous devons produire un *théorème d'inversion* qui exprime que l'on peut déduire que l'argument de l'appel récursif est bien dans le domaine de définition du fait que l'argument initial y était. Ici, le principe de non-pertinence des preuves ne s'applique pas ! Il est impératif que la démonstration soit faite par filtrage sur l'hypothèse (donc soit en utilisant la tactique `case` soit en utilisant la tactique `inversion`), que la preuve produite apparaisse bien comme une sous-preuve structurelle de la preuve initiale (l'utilisation de la tactique `injection` et de la tactique `rewrite` le permettent) et que la démonstration soit transparente. .

Dans notre cas d'exemple, il faut exprimer que si "$x = S\ (S\ p)$" est dans le domaine de définition, c'est que "$S\ (div2\ p)$" y était déjà. L'égalité est aussi utilisée pour montrer que le premier constructeur du prédicat inductif ne peut pas avoir été utilisé pour démontrer "`log_domain x`" :

```
Theorem log_domain_inv :
```

```
 ∀x p:nat, log_domain x → x = S (S p)→
          log_domain (S (div2 p)).
Proof.
 intros x p H; case H; try (intros H'; discriminate H').
 intros p' H1 H2; injection H2; intros H3;
 rewrite <- H3; assumption.
Defined.
```

Nous pouvons maintenant définir la fonction `log` comme une fonction récursive structurelle usuelle. Nous devons introduire une égalité pour permettre les raisonnements dans chaque cas ; ce procédé reproduit ce qui est effectué dans les tactiques `inversion` et `caseEq` (voir 9.5.2 page 279 et 7.2.7 page 185).

```
Fixpoint log (x:nat)(h:log_domain x){struct h} : nat :=
  match x as y return x = y → nat with
  | 0 ⇒ fun h' ⇒ False_rec nat (log_domain_non_O x h h')
  | S 0 ⇒ fun h' ⇒ 0
  | S (S p) ⇒
       fun h' ⇒ S (log (S (div2 p))(log_domain_inv x p h h'))
  end (refl_equal x).
```

L'introduction d'une égalité par `refl_equal` et l'utilisation d'un typage dépendant sont imposés par le fait que le traitement sur x porte sur l'argument du prédicat contrôlant la récursion. Lorsque le filtrage ne porte pas sur cet argument mais sur le résultat d'une fonction bien spécifiée, cette étape n'est pas nécessaire.

Par exemple, une autre fonction calculant le logarithme d'un nombre naturel peut utiliser la fonction `eq_nat_dec` pour tester son argument. Cette fonction peut alors être définie de la façon suivante, sans utiliser de filtrage dépendant :

```
Inductive log2_domain : nat→Prop :=
| l21 : log2_domain 1
| l22 : ∀x:nat,
        x ≠ 1 → x ≠ 0 → log2_domain (div2 x) → log2_domain x.

Hypothesis log2_domain_non_zero :
 ∀x:nat, log2_domain x → x ≠ 0.

Theorem log2_domain_invert :
 ∀x:nat, log2_domain x → x ≠ 0 → x ≠ 1 →
  log2_domain (div2 x).
Proof.
 intros x h; case h.
 intros h1 h2; elim h2; reflexivity.
 intros; assumption.
Defined.
```

```
Fixpoint log2 (x:nat)(h:log2_domain x){struct h} : nat :=
  match eq_nat_dec x 0 with
  | left heq ⇒ False_rec nat (log2_domain_non_zero x h heq)
  | right hneq ⇒
      match eq_nat_dec x 1 with
      | left heq1 ⇒ 0
      | right hneq1 ⇒
          S (log2 (div2 x)(log2_domain_invert x h hneq hneq1))
      end
  end.
```

Lorsqu'une fonction est définie par récurrence sur un prédicat inductif ad-hoc, les démonstrations sur cette fonction se font naturellement par récurrence sur ce prédicat inductif. Néanmoins il faut bien faire attention d'utiliser le principe de récurrence maximal, car ici le principe de non-pertinence des preuves n'est pas satisfait. Ce principe de récurrence maximal est obtenu à l'aide de la commande `Scheme` (voir section 15.1.6).

Voici par exemple, comment nous démontrons l'une des propriétés fondamentales de notre fonction logarithme.

```
Scheme log_domain_ind2 := Induction for log_domain Sort Prop.

Fixpoint two_power (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S p ⇒ 2 * two_power p
  end.

Section proof_on_log.

Hypothesis mult2_div2_le : ∀x:nat, 2 * div2 x ≤ x.

Theorem pow_log_le :
 ∀(x:nat)(h:log_domain x), two_power (log x h) ≤ x.
Proof.
 intros x h; elim h using log_domain_ind2.
 simpl; auto with arith.

 intros p l Hle.
 lazy beta iota zeta delta [two_power log_domain_inv log];
  fold log two_power.
 apply le_trans with (2 * S (div2 p)); auto with arith.
 exact (mult2_div2_le (S (S p))).
Qed.

End proof_on_log.
```

Bien sûr, il est aussi possible et recommandé de construire des fonctions bien spécifiées, comme c'est proposé dans l'exercice 16.19.

Dans l'article [17], Ana Bove et Venanzio Capretta montrent que la méthode peut s'appliquer également pour décrire des fonctions récursives imbriquées, mais ceci requiert un outil qui n'est pas fourni dans le système *Coq* : la possibilité de définir simultanément un prédicat inductif et une fonction récursive [40]. L'article [10] propose une méthode pour se passer de cet outil. La récursion sur un prédicat ad-hoc est également un aspect central du langage de programmation avec types dépendants proposé par Connor McBride et James McKinna dans [66].

Exercice 16.19 * Définir par récurrence sur `log_domain` une fonction ayant le type suivant :

```
∀x:nat, {y : nat | two_power y ≤ x ∧ x < two_power (S y)}.
```

Exercice 16.20 *** Nous reprenons ici le langage de programmation défini en section 9.4.2. La propriété inductive `forLoops` caractérise des programmes dans lesquels il est possible de reconnaître que toutes les boucles ont une variable qui décroît en restant un nombre entier positif à chaque itération. L'exécution de ces programmes est garantie de terminer. Le but de l'exercice est de décrire une fonction qui exécute de tels programmes lorsque l'exécution peut se produire sans erreur. Réussir à écrire une telle fonction est particulièrement remarquable parce que la sémantique du langage `inst` en fait un langage commplet au sens de Turing.

```
Open Scope Z_scope.

Definition extract_option (A:Set)(x:option A)(def:A) : A :=
  match x with
  | None ⇒ def
  | Some v ⇒ v
  end.
Implicit Arguments extract_option [A].
Implicit Arguments Some [A].

Inductive forLoops : inst→Prop :=
  | aForLoop :
      ∀(e:bExp)(i:inst)(variant:aExp),
        (∀s s':state,
           evalB s e = Some true → exec s i s' →
           Zwf 0 (extract_option (evalA s' variant) 0)
             (extract_option (evalA s variant) 0))→
        forLoops i → forLoops (WhileDo e i)
  | assignFor : ∀(v:Var)(e:aExp), forLoops (Assign v e)
  | skipFor : forLoops Skip
  | sequenceFor :
```

```
   ∀i1 i2:inst,
     forLoops i1 → forLoops i2 → forLoops (Sequence i1 i2).
```

Écrire une fonction dont le type est le suivant :

```
∀(s:state)(i:inst), forLoops i →
   {s':state | exec s i s'}+{∀s':state, ~exec s i s'}.
```

Chapitre 17

* Démonstration par réflexion

La démonstration par réflexion est caractéristique de la démonstration en théorie des types : puisque l'on dispose d'un langage de programmation à l'intérieur du langage logique, on va utiliser ce langage de programmation pour décrire des procédures de décision ou des méthodes systématiques de raisonnement. Toutefois, nous savons déjà que la programmation en *Coq* est une opération coûteuse et cette approche n'est justifiée que parce que les preuves construites sont effectivement plus efficaces : on pourra remplacer des dizaines d'opérations de réécriture par un petit nombre d'applications de théorèmes et une réduction du Calcul des Constructions pour assurer la convertibilité de deux termes. Puisque les calculs dans ce langage de programmation n'apparaissent pas dans les termes de preuve, on obtiendra des preuves qui sont plus petites et souvent plus rapides à vérifier.

Dans ce chapitre, nous allons décrire le principe général de la méthode et donner trois exemples simples portant sur la vérification qu'un nombre entier est premier et sur des égalités entre expressions algébriques.

17.1 Présentation générale

La démonstration par réflexion n'est pas très éloignée des techniques de démonstration que nous avons déjà rencontrées pour traiter les fonctions de *Coq*. Pour gérer de telles fonctions, nous faisons régulièrement appel aux réductions de termes fournies par le calcul des constructions : la $\beta\delta\zeta$-réduction pour les fonctions simples et la ι-réduction pour les fonctions récursives.

Observons de près une démonstration simple, la démonstration que l'addition des entiers naturels est associative :

```
Theorem plus_assoc : ∀x y z:nat, x+(y+z) = x+y+z.
Proof.
 intros x y z; elim x.
```

Ici, la preuve par récurrence demandée par la tactique `elim` produit deux buts. Le premier a la forme suivante :

```
  ...
  ============================
   0+(y+z) = 0+y+z
```

Nous avons l'habitude d'appeler `auto` pour résoudre ce but, mais si nous y regardons de plus près (par exemple grâce à la commande `Info`, ou en observant le terme construit), nous nous apercevons que cette tactique automatique effectue en fait l'opération suivante :

```
exact (refl_equal (y+z)).
```

C'est surprenant, car ni le membre gauche, ni le membre droit de l'égalité à prouver ne sont de la forme "`y + z`". En revanche, ils sont *convertibles* avec cette expression. Ainsi, pour vérifier cette étape de démonstration, le système *Coq* doit effectuer un petit calcul, qui mène à remplacer toutes les instances de "`0 +` m" par m, comme le préconise la définition de `plus`. Cette opération doit être effectuée deux fois (une fois dans le membre de gauche pour "m=`y + z`" et une fois dans le membre droit pour m=`y`), mais c'est entièrement transparent pour l'utilisateur. Nous ne terminerons pas cette démonstration, seule cette étape étant intéressante.

Cet exemple montre que la simplification de formules par conversion joue un rôle important dans le processus de preuve, car elle permet de remplacer quelques étapes de raisonnement par de simples étapes de calcul. Ici, les étapes de raisonnement qui sont traitées sont élémentaires, mais le but de la démonstration par réflexion est de faire effectuer des raisonnements complexes par calcul. En fait, nous allons utiliser la simplification pour construire de véritables procédures de décision. Pour effectuer des démonstrations par réflexion, nous allons décrire explicitement dans le langage logique les calculs normalement effectués dans les outils de preuve automatique. Nous serons naturellement amenés à démontrer que ces calculs représentent correctement les raisonnements représentés, mais comme pour le typage, ces démonstrations ne seront effectués qu'une fois. Les outils de démonstration automatique ainsi obtenus n'auront plus pour tâche de construire une preuve différente pour chaque donnée.

Il existe deux grandes classes de problèmes où ce type de démonstration par calcul s'avère utile. Dans la première classe de problèmes on travaille avec une propriété `C:T`→`Prop` où `T` est un type de donnée et l'on est capable de fournir une fonction `f:T`→`bool` telle que le théorème suivant soit vérifié :

```
 f_correct : ∀x:T, f x = true → C x.
```

Si `f` est définie de telle sorte que les règles de réduction permettent effectivement de réduire `f` t vers `true` pour une catégorie assez large de termes t, alors pour l'un de ces termes, le terme suivant sera une preuve de `C` t :

`f_correct` t `(refl_equal true):C` t

Mise à part l'occurrence de t apparaissant en premier argument de `f_correct`, la taille de ce terme de preuve ne dépend pas de t. En pratique, cette tactique s'applique seulement si t est un terme sans variable, c'est à dire un terme construit uniquement avec les constructeurs d'un type inductif. Dans la prochaine section, nous allons donner un exemple de ce procédé de démonstration pour la vérification qu'un nombre donné est premier et nous verrons que nous obtenons une tactique beaucoup plus rapide que celle décrite en section 8.5.2.

La deuxième classe de problèmes qui peuvent résolus par calcul est la classe des preuves algébriques, comme les preuves reposant sur la réécriture modulo l'associativité ou la commutativité de certains opérateurs. Pour ces preuves, nous considérons de nouveaux un type T and nous exhibons un typ abstrait A et deux fonctions $i : A \rightarrow T$ et $f : A \rightarrow A$. La fonction i est une fonction d'interprétation et permet d'associer des termes du type concret T avec des termes du type abstrait A. La fonction f effectue des raisonnements sur le type abstrait A. Le processus de réflexion repose sur un théorème qui exprime que la fonction f ne change pas l'interprétation du terme qu'elle manipule :

```
f_ident : ∀x:A, i (f x)= i x
```

Ainsi, pour prouver que deux termes t_1 et t_2 sont égaux dans T, nous pouvons montrer qu'il sont les images par la fonction d'interpretation de deux termes abstraits a_1 et a_2 tels que "$f\ \ a_1 = f\ \ a_2$". Nous donnons un exemple de ce type de raisonnement algébrique par réflexion dans la trois section de ce chapitre où nous étudions les preuves d'égalités modulo associativité et commutativité.

17.2 Démonstrations par calcul direct

Les fonctions utilisées dans le schéma de preuve par réflexion sont des fonctions récursives d'une classe particulière. En effet, il est important que la ι-réduction, telle qu'elle est opérée par la tactique `simpl` ou par les tactiques `lazy` et `cbv` suffise à ramener les expressions traitées dans la forme voulue. En pratique, il faudra donc que toutes ces fonctions soient programmées de façon structurelle récursive, au besoin en utilisant la technique de récursion bornée présentée en section 16.1.

Il faut bien sûr se préoccuper de considérations de complexité : la fonction développée en *Coq* sera exécutée dans le système de preuve lui-même, en utilisant les mécanismes de réduction internes qui n'ont pas une efficacité comparable à l'exécution d'un programme écrit dans un langage de programmation usuel et compilé. Pour des algorithmes dont on prévoit une utilisation massive sur des données parfois importantes, il faudra envisager le développement et la vérification formelle d'algorithmes efficaces.

Par exemple, si l'on veut obtenir une démonstration en *Coq* qu'un nombre de taille raisonnable est premier [1], il faudra montrer que ce nombre n'est pas divisible par une grande collection d'autres nombres. Les divisions par soustractions successives sont relativement peu efficaces et on pourra avoir avantage à d'abord convertir les données en représentation binaire avant de faire tous les tests de divisibilité.

1. Cet exemple nous a été suggéré par Martijn Ostdijk et Herman Geuvers.

Calcul de modulo

Les bibliothèques de *Coq* fournissent une fonction de division euclidienne sur les entiers relatifs dans le module `Zdiv`. Pour charger cette fonction dans *Coq*, il suffit d'entrer la commande suivante :

```
Require Export Zdiv.
```

La fonction de division s'appelle `Zdiv_eucl` et a le type `Z→Z→Z*Z`. Son type ne suffit donc pas à décrire son comportement, mais le théorème suivant fournit une information plus adaptée.

Z_div_mod : ∀ a b:Z, (b > 0)%Z →
let (q, r) := Zdiv_eucl a b in
*a = (b*q+r)%Z ∧ (0 ≤ r < b)%Z*

Le module `Zdiv` fournit également une fonction `Zmod` qui retourne uniquement le deuxième élément de la paire retournée par `Zdiv_eucl`.

Mise en œuvre de la réflexion

Nous avons besoin d'un premier théorème qui établit la correspondance entre l'existence d'un diviseur et le calcul de reste en représentation binaire. Nous ne détaillons pas ici la démonstration de ce théorème et nous nous contentons de l'admettre comme axiome :

```
Axiom verif_divide :
    ∀m p:nat, 0 < m → 0 < p →
    (∃q:nat | m = q*p)→(Z_of_nat m mod Z_of_nat p = 0)%Z.
```

Notre intention est de vérifier qu'un nombre est premier en vérifiant que le reste de la division de ce nombre par tout nombre plus petit que lui est non nul. Il nous faut justifier qu'il suffit de regarder les divisions par les nombres plus petits que lui. Ici encore nous admettons ce résultat, sans inclure la démonstration ici.

```
Axiom divisor_smaller :
    ∀m p:nat, 0 < m → ∀q:nat, m = q*p → q ≤ m.
```

Nous pouvons maintenant écrire la fonction qui va tester si un nombre est divisible par l'un de ses prédécesseur.

```
Fixpoint check_range (v:Z)(r:nat)(sr:Z){struct r} : bool :=
  match r with
    O ⇒ true
  | S r' ⇒
    match (v mod sr)%Z with
      Z0 ⇒ false
    | _ ⇒ check_range v r' (Zpred sr)
    end
  end.
```

```
Definition check_primality (n:nat) :=
  check_range (Z_of_nat n)(pred (pred n))(Z_of_nat (pred n)).
```

Nous pouvons maintenant tester cette fonction sur quelques valeurs.

```
Eval compute in (check_primality 2333).
   = true : bool

Eval compute in (check_primality 2330).
   = false : bool
```

Il aurait été plus simple de décrire cette fonction avec deux arguments, comme dans le texte suivant, où l'on refait la conversion du diviseur à chaque étape.

```
Fixpoint check_range' (v:Z)(r:nat){struct r} : bool :=
  match r with
    0 ⇒ true | 1 ⇒ true
  | S r' ⇒
      match (v mod Z_of_nat r)%Z with
      | 0%Z ⇒ false
      | _ ⇒ check_range' v r'
      end
  end.

Definition check_primality' (n:nat) :=
  check_range' (Zpos (P_of_succ_nat (pred n)))(pred (pred n)).
```

Mais cette fonction a une complexité inacceptable. En effet, chaque appel à la fonction `inject_nat` a un coût linéaire dans le nombre représenté. Dans la fonction `check_range`, ces conversions sont évitées et on fait seulement une soustraction sur un nombre binaire à chaque étape, ce qui a un coût linéaire dans le logarithme du nombre représenté. Le coût est bien moins élevé. Il est souvent utile de tester la complexité et la validité des fonctions avant de commencer la démonstration de leur correction. Des essais nous ont permis d'établir que la fonction `check_range` était préférable à la fonction `check_range'`.

Nous devons maintenant démontrer le théorème qui permet de déduire des preuves à partir des calculs effectués par ces fonctions. Nous utilisons deux lemmes dont nous ne donnons ici que l'énoncé.

```
Axiom check_range_correct :
  ∀ (v:Z)(r:nat)(rz:Z),
  (0 < v)%Z →
  Z_of_nat (S r) = rz →
  check_range v r rz = true →
  ~(∃k:nat | k ≤ S r ∧ k ≠ 1 ∧
                     (∃q:nat | Zabs_nat v = q*k)).
```

```
Axiom check_correct :
  ∀p:nat, 0 < p → check_primality p = true →
  ~(∃k:nat | k ≠ 1 ∧ k ≠ p ∧ (∃q:nat | p = q*k)).
```

Nous pouvons maintenant vérifier qu'un nombre arbitraire est premier comme dans l'exemple suivant : [2]

```
Theorem prime_2333 :
 ~(∃k:nat | k ≠ 1 ∧ k ≠ 2333 ∧ (∃q:nat | 2333 = q*k)).
Time apply check_correct; auto with arith.
```
Proof completed.
Finished transaction in 132. secs (131.01u,0.62s)
```
Time Qed.
```
...
Finished transaction in 59. secs (56.79u,0.4s)

Cette preuve prend quelques dizaines de secondes (temps cumulés de construction et de vérification du terme de preuve), tandis que la procédure naïve décrite dans la section 8.5.2 était incapable de traiter un nombre de cette taille. Il existe deux moyens simples d'améliorer encore l'efficacité de cette procédure : le premier consiste à ne vérifier que les nombres impairs et le nombre 2, le second est de se limiter à la vérification des nombres inférieurs à la racine carrée entière du nombre cherché (le module `ZArith` fournit une fonction de calcul de racine carrée appelée `Zsqrt`).

Martijn Oostdijk a développé une tactique encore plus élaborée, basée sur le lemme de Pocklington [22] qui permet de traiter des nombres entiers dont l'écriture décimale contient plusieurs dizaines de chiffres.

Exercice 17.1 ** Démontrer les théorèmes `verif_divide`, `divisor_smaller`, `check_range_correct`, `check_correct`.

Exercice 17.2 ** Démontrer que si un nombre n est le produit de deux autres nombres p et q alors l'un des deux nombres est inférieur à la partie entière de la racine carrée de n. En déduire une méthode par réflexion pour montrer qu'un nombre est premier en ne vérifiant que les divisions par des nombres impairs inférieurs à cette racine carrée.

17.3 ** Démonstrations par calcul algébrique

La réduction permet de calculer sur autre chose que seulement des nombres. Dans cette section, nous étudions des exemples où il s'agit de calculs symbolique sur des termes algébriques.

2. Les temps de calculs ont été obtenus sur un processor `Intel Pentium II` cadencé à 400 MHz.

17.3.1 Démonstrations modulo associativité

Nous nous intéressons ici tout particulièrement aux démonstrations que deux expressions de type `nat` sont égales lorsque ces démonstrations ne font intervenir que l'associativité de l'addition. Nous parlerons de démonstrations d'égalité modulo associativité. Ces démonstrations font intervenir des expressions de type `nat` qui sont simplement des arbres binaires, dans lesquels tous les nœuds sont occupés par la fonction `plus` et toutes les feuilles sont occupées par des expressions arithmétiques arbitraires. Informellement, des démonstrations d'égalité modulo associativité se font simplement en oubliant les parenthèses associées à toutes les additions dans les expressions à comparer, puis en vérifiant que les mêmes termes apparaissent dans le même ordre dans les deux expressions à comparer. Ainsi, il est évident au premier coup d'oeil que les expressions

$$x + ((y + z) + w) \qquad \text{et} \qquad (x + y) + (z + w)$$

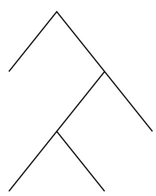
sont égales.

La démonstration de ce théorème sans utiliser la technique de réflexion a la forme suivante :

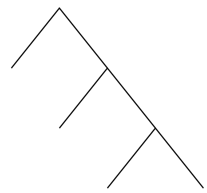
```
Theorem reflection_test :
 ∀x y z t u:nat, x+(y+z+(t+u)) = x+y+(z+(t+u)).
Proof.
 intros; repeat rewrite plus_assoc; auto.
Qed.
```

Ici, le script de preuve ne dépend pas des expressions de part et d'autre de l'égalité, mais la preuve construite par ce script grossit beaucoup avec le nombre d'additions présentes dans ces expressions. En pratique ceci signifie que le temps pris par cette tactique va croître rapidement avec la taille des expressions comparées. Le temps pris par la sauvegarde de la démonstration (commande `Qed`) croît également de façon importante. Lorsque les expressions comparées ont une grande taille, les temps de calcul deviennent inacceptables.

Oublier toutes les parenthèses reliées aux additions dans une expression revient à réécrire répétitivement avec le théorème d'associativité de l'addition jusqu'à ce que toutes additions soient repoussées vers la droite. Graphiquement, ceci revient à faire passer un arbre de la forme suivante :



à un arbre de la forme suivante :



Nous voulons donc construire une fonction qui transformerait, par exemple, l'expression $x+((y+z)+w)$ en $x+(y+(z+w))$, mais cette fonction ne peut pas s'écrire facilement comme une fonction récursive structurelle de `nat` vers `nat`, parce que l'addition n'est pas un constructeur du type `nat` et cela n'a pas de sens de se demander si x peut être considéré comme le résultat d'une addition. Nous allons donc la définir comme une fonction sur un type abstrait d'arbres binaires, que nous définissons par la commande suivante :

```
Inductive bin : Set := node : bin→bin→bin | leaf : nat→bin.
```

Nous construisons d'abord une fonction qui réorganise un arbre vers la droite en connaissant déjà le sous-arbre qui doit apparaître en dernière position, puis une fonction qui réorganise un arbre vers la droite sans connaitre la dernière feulle :

```
Fixpoint flatten_aux (t fin:bin){struct t} : bin :=
  match t with
  | node t1 t2 ⇒ flatten_aux t1 (flatten_aux t2 fin)
  | x ⇒ node x fin
  end.

Fixpoint flatten (t:bin) : bin :=
  match t with
  | node t1 t2 ⇒ flatten_aux t1 (flatten t2)
  | x ⇒ x
  end.
```

On peut vérifier directement en *Coq* que `flatten` construit bien des arbres qui dérivent vers la droite :

```
Eval compute in
  (flatten
    (node (leaf 1) (node (node (leaf 2)(leaf 3)) (leaf 4)))).
```
= node (leaf 1) (node (leaf 2) (node (leaf 3) (leaf 4))) : bin

L'étape suivante est de montrer comment les arbres binaires sont utilisés pour représenter des expressions de type `nat`. Ceci se fait à l'aide d'une fonction qui sera appelée la *fonction d'interprétation* :

```
Fixpoint bin_nat (t:bin) : nat :=
  match t with
  | node t1 t2 ⇒ bin_nat t1 + bin_nat t2
```

```
  | leaf n ⇒ n
  end.
```

Cette fonction d'interprétation indique bien que l'on va interpréter les arbres d'opérateur `node` comme des additions.

Nous pouvons utiliser les mécanismes d'évaluation d'expressions pour tester cette fonction :

```
Eval lazy beta iota delta [bin_nat] in
 (bin_nat
   (node (leaf 1) (node (node (leaf 2) (leaf 3)) (leaf 4)))).
    = 1+(2+3+4) : nat
```

L'étape suivante est de montrer que la mise en forme à droite ne change pas la valeur numérique représentée. On commence par démontrer que la fonction `flatten_aux` représente bien une addition :

```
Theorem flatten_aux_valid :
 ∀t t':bin, bin_nat t + bin_nat t' = bin_nat (flatten_aux t t').
```

Cette démonstration, que nous ne détaillons pas ici, se fait en suivant la structure de la fonction `flatten_aux`. Avec l'aide de ce théorème, nous avons un résultat similaire pour la fonction `flatten` :

```
Theorem flatten_valid : ∀t:bin, bin_nat t = bin_nat (flatten t).
```

Nous pouvons maintenant développer un théorème qui correspond à l'application de `flatten_valid` des deux cotés d'une égalité :

```
Theorem flatten_valid_2 :
  ∀t t':bin, bin_nat (flatten t) = bin_nat (flatten t')→
  bin_nat t = bin_nat t'.
Proof.
 intros; rewrite (flatten_valid t); rewrite (flatten_valid t');
 auto.
Qed.
```

Nous avons maintenant tous les ingrédients pour effectuer la démonstration que $x + ((y + z) + w)$ et $(x + y) + (z + w)$ sont égaux :

```
Theorem reflection_test' :
 ∀x y z t u:nat, x+(y+z+(t+u))=x+y+(z+(t+u)).
Proof.
 intros.
 change
   (bin_nat
      (node (leaf x)
         (node (node (leaf y) (leaf z))
               (node (leaf t)(leaf u)))) =
```

```
    bin_nat
      (node (node (leaf x)(leaf y))
         (node (leaf z)(node (leaf t)(leaf u))))).
 apply flatten_valid_2; auto.
Qed.
```

Cette démonstration utilise donc l'application de seulement deux théorèmes. Néanmoins, l'utilisateur doit encore donner l'expression qui est fournie à la tactique `change`. Cette opération manuelle peut elle-même être effectuée automatiquement grâce au langage $\mathcal{L}$tac décrit dans la section 8.5 :

```
Ltac model v :=
  match v with
  | (?X1 + ?X2) ⇒
    let r1 := model X1 with r2 := model X2 in
    constr:(node r1 r2)
  | ?X1 ⇒ constr:(leaf X1)
  end.

Ltac assoc_eq_nat :=
  match goal with
  | [ |- (?X1 = ?X2 :>nat) ] ⇒
   let term1 := model X1 with term2 := model X2 in
   (change (bin_nat term1 = bin_nat term2);
    apply flatten_valid_2;
    lazy beta iota zeta delta [flatten flatten_aux bin_nat];
    auto)
  end.
```

Maintenant, la tactique `AssocEqNat` condense en un seul mot clef l'ensemble des actions à effectuer pour prouver l'égalité selon cette méthode, comme le montre la session suivante.

```
Theorem reflection_test'' :
 ∀x y z t u:nat, x+(y+z+(t+u)) = x+y+(z+(t+u)).
Proof.
 intros; assoc_eq_nat.
Qed.
```

Nous laissons la lectrice tester cette tactique sur une collection de cas plus large.

Exercice 17.3 Démontrer `flatten_aux_valid`, `flatten_valid`, et `flatten_valid_2`.

17.3.2 Abstraire sur le type et l'opérateur

Les démonstrations que nous avons effectuées dans la section précédente sur les nombres entiers sont valides pour toutes les fonctions à deux arguments qui sont associatives. En restant dans les nombres naturels la méthode devrait s'appliquer aussi pour la multiplication, mais en changeant de type on peut aussi considérer l'addition et la multiplication sur les entiers relatifs, les nombres rationnels...Nous allons montrer dans cette section comment décrire une fois pour toute la tactique à un niveau abstrait, de façon à pouvoir la spécialiser à des cas variés.

Pour effectuer ce travail d'abstraction, il est préférable d'utiliser le mécanisme de sections fourni dans *Coq*.

```
Section assoc_eq.
```

Nous allons maintenant donner les différents éléments qui peuvent varier dans les différents usages de la même tactique. Bien sûr ces éléments doivent être cohérents entre eux. Nous devons disposer d'un type de données (un type de sorte `Set`), d'une fonction à deux arguments de ce type dans lui-même et d'un théorème qui exprime que cette fonction est associative.

```
Variables (A : Set)(f : A→A→A)
  (assoc : ∀x y z:A, f x (f y z) = f (f x y) z).
```

Nous devons maintenant construire la fonction qui associe un terme de type `A` à un terme de type `bin`. Ici, il faut faire attention : les feuilles des termes de type `bin` portent des valeurs de type `nat` et non des valeurs de type `A`. Nous pourrions utiliser une structure d'arbres polymorphes telle que chaque feuille porte une valeur d'un type choisi et appliqué cette structure à `A`. Nous préférons continuer avec une structure d'arbre portant des valeurs entières et établir la correspondance par l'intermédiaire d'une liste de données de type `A`. Chaque entier dans l'arbre représente donc une position dans une liste de données. Ce choix est justifié dans une section ultérieure où nous étendons notre outil pour prendre en compte les opérations associatives et commutatives. Pour interpréter les nombres naturels comme des positions dans la liste, nous utilisons une fonction `nth` qui prend trois arguments, un nombre naturel, une liste de termes de type `A` et une valeur par défaut qui sera utilisée dans le cas où le nombre naturel choisi est plus grand que la longueur de la liste. Cette fonction `nth` est fournie dans les bibliothèques de *Coq* (module `List`). La fonction d'interprétation que nous définissons ici est la transposée abstraite de la fonction `bin_nat` que nous avons décrite plus tôt.

```
Fixpoint bin_A (l:list A)(def:A)(t:bin){struct t} : A :=
  match t with
  | node t1 t2 ⇒ f (bin_A l def t1)(bin_A l def t2)
  | leaf n ⇒ nth n l def
  end.
```

Les théorèmes de validité doivent aussi être transposés. Voici leurs types :

```
Theorem flatten_aux_valid_A :
 ∀(l:list A)(def:A)(t t':bin),
```

```
 f (bin_A l def t)(bin_A l def t') =
       bin_A l def (flatten_aux t t').

Theorem flatten_valid_A :
 ∀(l:list A)(def:A)(t:bin),
   bin_A l def t = bin_A l def (flatten t).

Theorem flatten_valid_A_2 :
 ∀(t t':bin)(l:list A)(def:A),
   bin_A l def (flatten t) = bin_A l def (flatten t')→
   bin_A l def t = bin_A l def t'.
```

Nous pouvons maintenant fermer la section pour obtenir un théorème abstrait.

```
End assoc_eq.
Check flatten_valid_A_2.
```

flatten_ valid_ A_ 2:
∀ (A:Set)(f:A→A→A),
(∀ x y z:A, f x (f y z) = f (f x y) z)→
∀ (t t':bin)(l:list A)(def:A),
bin_ A A f l def (flatten t) = bin_ A A f l def (flatten t')→
bin_ A A f l def t = bin_ A A f l def t'

La mise en œuvre à l'aide du langage de programmation de tactiques est un peu plus complexe maintenant, car la liste de termes de type `A` doit d'abord être construite, puis il faut déterminer la position de chaque élément dans cette liste.

```
Ltac term_list f l v :=
  match v with
  | (f ?X1 ?X2) ⇒
    let l1 := term_list f l X2 in term_list f l1 X1
  | ?X1 ⇒ constr:(cons X1 l)
  end.

Ltac compute_rank l n v :=
  match l with
  | (cons ?X1 ?X2) ⇒
    let tl := constr:X2 in
    match constr:(X1 = v) with
    | (?X1 = ?X1) ⇒ n
    | _ ⇒ compute_rank tl (S n) v
    end
  end.

Ltac model_aux l f v :=
  match v with
```

```
  | (f ?X1 ?X2) ⇒
    let r1 := model_aux l f X1 with r2 := model_aux l f X2 in
      constr:(node r1 r2)
  | ?X1 ⇒ let n := compute_rank l 0 X1 in constr:(leaf n)
  | _ ⇒ constr:(leaf 0)
  end.

Ltac model_A A f def v :=
  let l := term_list f (nil (A:=A)) v in
  let t := model_aux l f v in
  constr:(bin_A A f l def t).

Ltac assoc_eq A f assoc_thm :=
  match goal with
  | [ |- (@eq A ?X1 ?X2) ] ⇒
  let term1 := model_A A f X1 X1
  with term2 := model_A A f X1 X2 in
  (change (term1 = term2);
   apply flatten_valid_A_2 with (1 := assoc_thm); auto)
  end.
```

La tactique `AssocEq` doit s'utiliser de la même manière que la tactique `AssocEqNat`, mais nous devons indiquer comment elle est spécialisée pour chaque utilisation, en donnant le type, l'opérateur binaire, et le théorème d'associativité. Voici un exemple d'utilisation sur les nombres entiers.

```
Theorem reflection_test3 :
 ∀x y z t u:Z, (x*(y*z*(t*u)) = x*y*(z*(t*u)))%Z.
Proof.
 intros; assoc_eq Z Zmult Zmult_assoc.
Qed.
```

Exercice 17.4 Donner les preuves des théorèmes `flatten_aux_valid_A`, `flatten_valid_A` et `flatten_valid_A_2` : elles requièrent l'utilisation de l'hypothèse `f_assoc`.

Exercice 17.5 On suppose maintenant que l'opération binaire admet un élément neutre comme 0 pour l'addition. Adapter la tactique pour qu'elle permette aussi de prouver des égalités de la forme "`(x+0)+(y+(z+0))=x+(y+(z+0))`".

17.3.3 *** Tri de variables pour la commutativité

Lorsque l'on met des données dans une liste, comme nous l'avons fait dans la section précédente, on établit un ordre sur ces valeurs. Cet ordre est arbitraire, mais il peut servir pour certaines applications. Un exemple de telles applications est celui où l'on effectue des démonstrations modulo commutativité. Dans ce cas, on ne se contente pas d'aplatir la structure de l'expression, mais on essaie également de l'ordonner de façon que les termes qui apparaissent des deux cotés de l'égalité apparaissent dans la même position. L'exemple développé ici est inspiré du développement de la tactique `field` par D. Delahaye et M. Mayero.

Nous travaillons encore avec une seule opération binaire dont nous voulons capturer les propriétés algébriques, et nous utilisons encore le type de donnée `bin` et une propriété d'associativité pour l'opérateur binaire. Ceci nous permettra d'obtenir des arbres binaires dans lesquels tous les fils gauches seront des feuilles. Pour traiter la commutativité, nous trions ces feuilles dans l'ordre fourni par le stockage dans la liste.

Nous allons procéder à un tri par insertion. Pour ce tri nous avons besoin d'une fonction de comparaison qui permette de comparer deux feuilles en utilisant le nombre porté par ces feuilles.. C'est une simple fonction structurelle récursive et son exécution sur deux entiers connus se réduira toujours bien à une valeur booléenne `true` ou `false` :

```
Fixpoint nat_le_bool (n m:nat){struct m} : bool :=
  match n, m with
  | O, _ ⇒ true
  | S _, O ⇒ false
  | S n, S m ⇒ nat_le_bool n m
  end.
```

Lorsque nous allons procéder à notre tri, nous allons devoir insérer successivement des feuilles annotées par des entiers dans des arbres déjà triés. Dans la fonction d'insertion suivante, nous repérons la feuille à insérer par la valeur (un nombre naturel) qu'elle porte. Pour l'arbre dans lequel on insère, on ne considère que le cas où cet arbre est bien formé, c'est à dire que son sous-arbre de gauche est réduit à une feuille. Si ce n'est pas le cas, on se contente d'adjoindre la feuille à l'arbre. Lorsque l'arbre est bien formé, on compare la valeur entière à insérer avec la valeur portée par le sous-arbre de gauche, qui est une feuille. Il faut quand même faire attention au cas de base, où l'arbre dans lequel on insère est une feuille.

```
Fixpoint insert_bin (n:nat)(t:bin){struct t} : bin :=
  match t with
  | leaf m ⇒ match nat_le_bool n m with
              | true ⇒ node (leaf n)(leaf m)
              | false ⇒ node (leaf m)(leaf n)
              end
  | node (leaf m) t' ⇒
    match nat_le_bool n m with
    | true ⇒ node (leaf n) t
    | false ⇒ node (leaf m)(insert_bin n t')
    end
  | t ⇒ node (leaf n) t
  end.
```

Avec cette fonction d'insertion, nous pouvons maintenant construire une fonction de tri.

```
Fixpoint sort_bin (t:bin) : bin :=
  match t with
  | node (leaf n) t' ⇒ insert_bin n (sort_bin t')
  | t ⇒ t
  end.
```

Il ne nous reste qu'à démontrer que ce tri ne change pas la valeur de l'expression représentée par un arbre.

```
Section commut_eq.
 Variables (A : Set)(f : A→A→A).
 Hypothesis comm : ∀x y:A, f x y = f y x.
 Hypothesis assoc : ∀x y z:A, f x (f y z) = f (f x y) z.
```

Ici, nous pouvons reprendre les fonctions `flatten_aux`, `flatten`, `bin_A` et les théorèmes `flatten_aux_valid`, `flatten_valid` et `flatten_valid_2` déjà utilisés dans la section précédente (voir page 487, les lecteurs qui exécutent les différentes commandes à l'aide de *Coq* au fur et à mesure devraient reprendre ce fragment de code ici). Nous devons maintenant démontrer que le tri d'une expression ne change pas sa valeur et pour cela, nous devons commencer par démontrer que l'insertion d'un index dans un arbre binaire correspond à la composition de la valeur indexée par l'opération binaire.

```
Theorem insert_is_f :
 ∀(l:list A)(def:A)(n:nat)(t:bin),
  bin_A l def (insert_bin n t) = f (nth n l def)(bin_A l def t).
```

Avec ce théorème, il est maintenant aisé de démontrer que le tri ne change pas la valeur d'une expression.

```
Theorem sort_eq : ∀(l:list A)(def:A)(t:bin),
    bin_A l def (sort_bin t) = bin_A l def t.
```

Comme dans la section précédente, nous construisons également un théorème qui applique l'opération de tri des deux cotés d'une égalité.

```
Theorem sort_eq_2 :
 ∀(l:list A)(def:A)(t1 t2:bin),
   bin_A l def (sort_bin t1) = bin_A l def (sort_bin t2)→
   bin_A l def t1 = bin_A l def t2.
```

Nous pouvons maintenant rendre ces théorèmes plus généraux en fermant la section.

```
End commut_eq.
```

Ces théorèmes peuvent être utilisés dans une tactique générale dont le schéma est le même que pour la section précédente. Notez que le théorème `sort_eq_2` est appliqué après le théorème `flatten_valid_A_2`, de sorte que la fonction `sort_bin` est appliquée à l'expression retournée par la fonction `flatten`, qui est donc un arbre bien formé (les sous-expressions de gauche sont toujours des feuilles).

```
Ltac comm_eq A f assoc_thm comm_thm :=
  match goal with
  | [ |- (?X1 = ?X2 :>A) ] ⇒
    let l := term_list f (nil (A:=A)) X1 in
    let term1 := model_aux l f X1
    with term2 := model_aux l f X2 in
    (change (bin_A A f l X1 term1 = bin_A A f l X1 term2);
      apply flatten_valid_A_2 with (1 := assoc_thm);
      apply sort_eq_2 with (1 := comm_thm)(2 := assoc_thm);
      auto)
  end.
```

Voici un exemple de but qui est résolu par cette tactique :

```
Theorem reflection_test4 : ∀x y z:Z, (x+(y+z) = (z+x)+y)%Z.
Proof.
 intros x y z. comm_eq Z Zplus Zplus_assoc Zplus_comm.
Qed.
```

Exercice 17.6 Démontrer `insert_is_f`, `sort_eq`, `sort_eq_2`.

17.4 Conclusion

Les bibliothèques de *Coq* fournissent d'autres exemples plus élaborés. Les tactiques `ring` et `field` sont basées sur cette technique et l'utilisateur aura avantage à s'inspirer de ces exemples pour développer ses propres tactiques.

Dans les tactiques réflexives, les considérations d'efficacité des algorithmes décrits sont très importantes, car ces tactiques seront exécutées dans un moteur d'exécution comparativement lent par rapport aux moteurs fournis pour les langages de programmation usuels. Pour une tactique utilisée de façon intensive, on aura donc avantage à utiliser des algorithmes de tris plus efficaces que le tri par insertion et un stockage de données plus efficace que le stockage dans une liste linéaire. Par exemple, on pourra utiliser une bibliothèque de stockage dans des arbres binaires, où chaque position peut être repérée par un nombre de type `positive`. Ce stockage fournit encore une notion d'ordre et l'opération de recherche d'une valeur a un coût bien moindre que dans le cas de la recherche dans une structure de liste.

Exercice 17.7 ** En utilisant la notion de permutation vue dans l'exercice 9.4, page 247 et la fonction de comptage définie dans l'exercice 10.5 page 286, montrer que si l' est une permutation de l, alors tout naturel n a autant d'occurrences dans l que dans l'.

Écrire une tactique " `NoPerm` n " permettant de montrer que l' n'est pas une permutation de l, en s'appuyant sur le nombre d'occurrences de n dans l et l' (l et l' étant déterminées par le but courant.)

Annexes

Tri par insertion : le code

Nous présentons ci-dessous le listing du développement d'un tri par insertion commenté page 27. On peut télécharger ce code depuis [11] (rubrique « A brief presentation of *Coq* ») ; ce site contient également une description détaillée de ce code (en anglais).

```
(* A sorting example :
   (C) Yves Bertot, Pierre Castéran
*)

Require Import List.
Require Import ZArith.
Open Scope Z_scope.

Inductive sorted : list Z -> Prop :=
  | sorted0 : sorted nil
  | sorted1 : forall z:Z, sorted (z :: nil)
  | sorted2 :
      forall (z1 z2:Z) (l:list Z),
        z1 <= z2 ->
        sorted (z2 :: l) -> sorted (z1 :: z2 :: l).

Hint Resolve sorted0 sorted1 sorted2 : sort.

Lemma sort_2357 :
 sorted (2 :: 3 :: 5 :: 7 :: nil).
Proof.
 auto with sort zarith.
Qed.


Theorem sorted_inv :
 forall (z:Z) (l:list Z), sorted (z :: l) -> sorted l.
Proof.
```

```
 intros z l H.
 inversion H; auto with sort.
Qed.

(*  Number of occurrences of z in l *)

Fixpoint nb_occ (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil => 0%nat
  | (z' :: l') =>
      match Z_eq_dec z z' with
      | left _ => S (nb_occ z l')
      | right _ => nb_occ z l'
      end
  end.

Eval compute in (nb_occ 3 (3 :: 7 :: 3 :: nil)).

Eval compute in (nb_occ 36725 (3 :: 7 :: 3 :: nil)).


(* list l' is a permutation of list l *)

Definition equiv (l l':list Z) :=
    forall z:Z, nb_occ z l = nb_occ z l'.

(* equiv is an equivalence ! *)

Lemma equiv_refl : forall l:list Z, equiv l l.
Proof.
 unfold equiv; trivial.
Qed.

Lemma equiv_sym : forall l l':list Z, equiv l l' -> equiv l' l.
Proof.
  unfold equiv; auto.
Qed.

Lemma equiv_trans :
 forall l l' l'':list Z, equiv l l' ->
                         equiv l' l'' ->
                         equiv l l''.
Proof.
 intros l l' l'' H H0 z.
 eapply trans_eq; eauto.
Qed.
```

```
Lemma equiv_cons :
 forall (z:Z) (l l':list Z), equiv l l' ->
                            equiv (z :: l) (z :: l').
Proof.
 intros z l l' H z'.
 simpl; case (Z_eq_dec z' z); auto.
Qed.


Lemma equiv_perm :
 forall (a b:Z) (l l':list Z),
   equiv l l' ->
   equiv (a :: b :: l) (b :: a :: l').
Proof.
 intros a b l l' H z; simpl.
 case (Z_eq_dec z a); case (Z_eq_dec z b);
  simpl; case (H z); auto.
Qed.

Hint Resolve equiv_cons equiv_refl equiv_perm : sort.


(* insertion of z into l at the right place
   (assuming l is sorted)
*)

Fixpoint aux (z:Z) (l:list Z) {struct l} : list Z :=
  match l with
  | nil => z :: nil
  | cons a l' =>
      match Z_le_gt_dec z a with
      | left _ =>  z :: a :: l'
      | right _ => a :: (aux z l')
      end
  end.


Eval compute in (aux 4 (2 :: 5 :: nil)).

Eval compute in (aux 4 (24 :: 50 ::nil)).

(* the aux function seems to be a good tool for sorting ... *)
```

```
Lemma aux_equiv : forall (l:list Z) (x:Z),
                  equiv (x :: l) (aux x l).
Proof.
 induction l as [|a l0 H]; simpl ; auto with sort.
 intros x; case (Z_le_gt_dec x a);
   simpl; auto with sort.
 intro; apply equiv_trans with (a :: x :: l0);
   auto with sort.
Qed.


Lemma aux_sorted :
 forall (l:list Z) (x:Z), sorted l -> sorted (aux x l).
Proof.
 intros l x H; elim H; simpl; auto with sort.
 intro z; case (Z_le_gt_dec x z); simpl;
   auto with sort zarith.
 intros z1 z2; case (Z_le_gt_dec x z2); simpl; intros;
  case (Z_le_gt_dec x z1); simpl; auto with sort zarith.
Qed.

(* the sorting function *)

Definition sort :
  forall l:list Z, {l' : list Z | equiv l l' /\ sorted l'}.
 induction l as [| a l IHl].
 exists (nil (A:=Z)); split; auto with sort.
 case IHl; intros l' [H0 H1].
 exists (aux a l'); split.
 apply equiv_trans with (a :: l'); auto with sort.
 apply aux_equiv.
 apply aux_sorted; auto.
Defined.

Extraction "insert-sort" aux sort.
```

Index

Coq et bibliothèques

Exemples du livre

Bibliographie

[1] Users contributions to the *Coq* system. `http://coq.inria.fr/`.

[2] Wilhelm Ackermann. On Hilbert's construction of the real numbers. In van Heijenoort [86], pages 493–507.

[3] Wilhelm Ackermann. A question on transfinite numbers. In van Heijenoort [86], pages 104–112.

[4] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1977.

[5] Cuihtlauac Alvarado. *Réflexion pour la réécriture dans le calcul des constructions inductives*. PhD thesis, Université de Paris XI, 2002. `http://perso.rd.francetelecom.fr/alvarado/publi/these.ps.gz`.

[6] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[7] Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.

[8] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2) :125–154, April 1991.

[9] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[10] Yves Bertot, Venanzio Capretta, and Kuntal Das Barman. Type-theoretic functional semantics. In *Theorem Proving in Higher Order Logics (TPHOLS'02)*, volume 2410 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[11] Yves Bertot and Pierre Castéran. Coq'Art : examples and exercises. `http://www.labri.fr/Perso/~casteran/CoqArt`.

[12] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 531–545, 1995.

[13] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29 :225–252, 2002.

[14] Richard J. Boulton and Paul B. Jackson, editors. *Theorem Proving in Higher Order Logics : 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[15] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Science*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[16] Ana Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1) :22–42, 2001.

[17] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Boulton and Jackson [14], pages 121–135.

[18] Robert S. Boyer and J Strother Moore. Proving theorems about lisp functions. *Journal of the ACM*, 22(1) :129–144, 1975.

[19] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[20] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[21] Venanzio Capretta. Certifying the fast Fourier transform with Coq. In Boulton and Jackson [14], pages 154–168.

[22] Olga Caprotti and Martijn Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *Journal of Symbolic Computation*, 32(1/2) :55–70, July 2001.

[23] Pierre Castéran and Davy Rouillard. Reasoning about parametrized automata. In *Proceedings, 8-th International Conference on Real-Time System*, volume 8, pages 107–119, 2000.

[24] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2000.

[25] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1) :56–68, 1940.

[26] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice Hall, 1986.

[27] Thierry Coquand. An analysis of Girard's paradox. In *Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1986.

[28] Thierry Coquand. Metamathematical investigations on a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[29] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[30] Solange Coupet-Grimal. LTL in Coq. Technical report, Contributions to the Coq System, 2002.

[31] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6) :801–813, 2003.

[32] Solange Coupet-Grimal and Line Jakubiec. Hardware verification using co-induction in coq. In *TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[33] Haskell B. Curry and Robert Feys. *Combinatory Logic I.* North- Holland, 1958.

[34] Olivier Danvy. Back to direct style. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 130–150. Springer-Verlag, 1992.

[35] Nicolaas G. de Bruijn. The mathematical language automath, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lectur Notes in Mathematics*. Springer-Verlag, 1970.

[36] Richard Dedekind. *Was sind und was sollen die Zahlen ?* Vieweg, 1988.

[37] David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve, Une étude dans le cadre du système Coq*. PhD thesis, Université de Paris VI, Pierre et Marie Curie, 2001.

[38] Development team. The *Coq* proof assistant. Documentation, system download. Contact : `http://coq.inria.fr/`.

[39] Edsger W. Dijkstra. *A discipline of Programming*. Prentice Hall, 1976.

[40] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), 2000.

[41] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4) :709–745, 2003.

[42] Jean-Christophe Filliâtre. L'outil de vérification Why. `http://why.lri.fr/`.

[43] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science : 19th Symposium on Applied Mathematics*, pages 19–31, 1967.

[44] Jean-Baptiste-Joseph Fourier. *Oeuvre de Fourier*. Gauthier-Villars, 1890. Publié par les soins de Gaston Darboux.

[45] Eduardo Gimenez. A tutorial on recursive types in Coq. Documentation of the Coq system.

[46] Eduardo Gimenez. An application of co-inductive types in Coq : Verification of the alternating bit protocol. In *Proceedings of the 1995 Workshop on Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152. Springer-Verlag, 1995.

[47] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Paris VII, 1972.

[48] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge University Press, 1989.

[49] Michael Gordon and Tony Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[50] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF : A mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[51] Arend Heyting. *Intuitionism - an Introduction*. North-Holland, 1971.

[52] David Hilbert. On the infinite. In van Heijenoort [86], pages 367–392.

[53] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[54] William A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry : Essays on combinatory logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[55] Gérard Huet. Induction principles formalized in the calculus of constructions. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 205–216. North-Holland, 1988.

[56] Gilles Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. North-Holland, 1988.

[57] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-aided reasoning : an approach*. Kluwer Academic Publishing, 2000.

[58] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 109–122. ACM, 1994.

[59] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3), 2000.

[60] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[61] Zhaohui Luo. *Computation and Reasoning – A Type Theory for Computer Science*. Oxford University Press, 1994.

[62] Zhaohui Luo and Randy Pollack. Lego proof development system : user's manual. Technical Report ECS-LFCS-92-211, LFCS (Edinburgh University), 1992.

[63] Assia Mahboubi and Loïc Pottier. Élimination des quantificateurs sur les réels en Coq. In *Journées Francophones des Langages Applicatifs, Anglet*, Jan 2002.

[64] Per Martin-Löf. *Intuitionistic type theories*. Bibliopolis, 1984.

[65] Conor McBride. Elimination with a motive. In *Types for Proofs and Programs'2000*, volume 2277, pages 197–217, 2002.

[66] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[67] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B :Formal Models and Semantics*. MIT Press and Elsevier, 1994.

[68] Jean-François Monin. *Understanding Formal Methods*. Springer-Verlag, 2002.

[69] Bengt Nordstrom, Kent Petersson, and Jan Smith. Martin-löf's type theory. In *Handbook of Logic in Computer Science, Vol. 5*. Oxford University Press, 1994.

[70] Sam Owre, Sreeranga P. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS : Combining specifications, proof checking and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, 1996.

[71] Catherine Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In *Proceedings of MPC'1995*, volume 947 of *Lecture Notes in Computer Science*, pages 351–379, 1995.

[72] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993. LIP research report 92-49.

[73] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[74] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

[75] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3) :363–397, 1989.

[76] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, 1977.

[78] Olivier Pons. Ingéniérie de preuve. In *Journées Francophones pour les Langages Applicatifs*, January 2000.

[79] Dag Prawitz. Ideas and results in proof theory. In *Proceedings of the second Scandinavian logic symposium*. North-Holland, 1971.

[80] William Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. *CACM*, 8 :102–114, 1992.

[81] Dana Scott. Constructive validity. In *Proceedings of Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, 1970.

[82] Alfred Tarski. The semantic conception of truth and the foundations of semantics. *Philosophy and Phenomenological Research*, 4, 1944. Transcription available at `www.ditext.com/tarski/tarski.html`.

[83] Coq Development Team. The Coq reference manual. LogiCal Project, `http://coq.inria.fr/`.

[84] Laurent Théry. A certified version of Buchberger's algorithm. In *Automated Deduction—CADE-15*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 349–364. Springer-Verlag, 1998.

[85] Andrzej Trybulec. The Mizar-qc/6000 logic information language. *ALLC Bulletin*, 6(2) :136–140, 1978.

[86] Jean van Heijenoort, editor. *From Frege to Gödel : a source book in mathematical logic, 1879-1931*. Harvard University Press, 1981.

[87] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1) :164–180, January 1980.