UNIVERSITÉ DE PARIS I
PANTHÉON-SORBONNE

THÈSE

présentée pour obtenir le titre de Docteur (arrêté du 30 mars 1992) par :

Xavier SCHAEFER

Bases de Données Orientées Objet, Contraintes d'Intégrité et Analyse Statique

Soutenue Le 15 Octobre 1997 devant le jury composé de :

Mme	Marie-Claude GAUDEL	Président
M.	Peter BUNEMAN	Rapporteur
M.	Laurent VIEILLE	Rapporteur
Mme	Véronique BENZAKEN	
Mme	Françoise GIRE	
Mme	Colette ROLLAND	
M.	Nicolas SPYRATOS	

À Anne-Catherine

Remerciements

Je remercie chaleureusement Marie-Claude Gaudel, qui me fait le très grand honneur de présider ce jury.

Je suis très reconnaissant à Peter Buneman d'avoir accepté la tâche ingrate d'être rapporteur de cette thèse (en particulier pour ses efforts pour lire mon français).

Je remercie vivement Laurent Vieille d'avoir accepté d'être rapporteur de cette thèse. Son intérêt pour mon travail et ses encouragements m'ont été très précieux.

Je tiens à remercier particulièrement Véronique Benzaken, qui a dirigé cette thèse. Tout d'abord, ses efforts pour définir un sujet "clés en main" ont été inestimables car ils m'ont permis de démarrer ce travail rapidement. Ensuite, elle a su me témoigner une grande confiance malgré mes (toutes aussi grandes) lacunes. Elle m'a beaucoup encouragé pendant ces trois années, et m'a appris à mener à bien un travail de recherche.

Colette Rolland me fait l'honneur de participer à ce jury, je l'en remercie très chaleureusement. Je lui suis infiniment reconnaissant de m'avoir donné les moyens de faire ma recherche. Sa bienveillance désintéressée m'a été d'une aide précieuse.

Je remercie vivement Françoise Gire, qui me fait l'honneur d'être membre de ce jury. L'intérêt qu'elle a porté à mon travail pendant ces trois années m'a beaucoup aidé.

Nicolas Spyratos me fait l'honneur d'être membre de ce jury. Il m'a accueilli au sein de l'équipe bases de données du L.R.I. et je l'en remercie chaleureusement.

Je remercie chaleureusement Serena Cerrito. Sa grande rigueur a été pour moi un modèle. Son aide et sa gentillesse ont été très précieuses. Je remercie aussi Giuseppe Castagna, pour ses encouragements et la lecture critique et constructive qu'il a faite de mes travaux.

Je remercie tous les membres de l'équipe bases de données du L.R.I., ainsi que les membres du C.R.I.

Enfin, je remercie du fond du cœur ma famille, et en particulier mes parents et ma femme, Anne-Catherine, qui m'ont tant soutenu pendant ces trois années.

Table des matières

Chapitre 1

Introduction

Un nouveau type de systèmes de gestion de bases de données est apparu dans le courant des années 80 : les systèmes orientés objet. Leur objectif était de combler les lacunes des systèmes relationnels [AB87, AB91, SDA94, ABGO93, ABGO95, AHV95]. Les défauts des systèmes relationnels sont de deux ordres :

Premièrement, la simplicité des structures de données (les relations) rendent leur utilisation difficile dans certains domaines, en particulier pour ce qu'il est convenu d'appeler les "nouveaux domaines d'application" : conception assistée par ordinateur, multimédia, "design", etc. Ces domaines utilisent en effet des données structurées de façon complexe et il est difficile de représenter ces données avec les seules relations. Par le biais du processus de normalisation, l'utilisation de ces relations a pour effet de disséminer la vision globale des données de l'application, que l'on ne peut retrouver que par l'application d'opérateurs de jointure.

D'autre part, les systèmes de gestion de bases de données relationnels imposent une séparation nette entre, d'un côté, le stockage des données, et de l'autre, le traitement de ces données. C'est le système de gestion de bases de données qui s'occupe du stockage de ces données. Les traitements sont effectués via un langage de programmation extérieur comme COBOL ou C, auquel il faut ajouter des instructions d'accès à la base (SQL imbriqué). Cette séparation pose des problèmes divers. En particulier, les systèmes relationnels sont faits pour manipuler des relations (c'est-à-dire des ensembles de n-uplets), alors que les langages de programmation travaillent n-uplet par n-uplet.

Afin de pallier ces problèmes, une partie importante de la recherche dans le domaine des bases de données a été consacrée à la définition de nouveaux systèmes incorporant à la fois un stockage efficace des données (persistance), un système de type riche permettant de représenter des données complexes, et enfin un langage de manipulation de données puissant et intégré au reste du système. Ces nouveaux systèmes sont le fruit de la fusion des bases de données et des langages de programmation à objets.

L'intégration de langages de programmation dans les systèmes de gestion de bases de données a ouvert de nouvelle voies pour l'amélioration du fonctionnement de ces systèmes, alors qu'avant, le caractère hétérogène des différents éléments assemblés pour obtenir un système complet rendait les optimisations globales impossibles. En particulier, un gain considérable peut être obtenu en ce qui concerne *la gestion de l'intégrité*.

La gestion de l'intégrité figure parmi les nombreuses tâches habituellement assignées aux systèmes de gestion de bases de données. D'une façon générale, *l'intégrité* d'une base de données désigne la bonne adéquation entre cette base et le monde réel qu'elle est censée représenter. Le critère d'adéquation est fourni sous la forme d'un ensemble de *contraintes d'intégrité*. Une contrainte d'intégrité est simplement une condition qui doit être satisfaite par la base de données pour que celle-ci représente bien le monde réel. Les systèmes de gestion de bases de données offrent des langages, la plupart du temps inspirés de la logique du premier ordre, qui permettent d'exprimer toute une panoplie de contraintes.

La gestion de l'intégrité est un aspect particulièrement critique eu égard à l'efficacité des systèmes de gestion de bases de données. En effet, une base de données n'a pas pour vocation d'être figée, mais évolue en parallèle avec le monde qu'elle représente. Cette évolution s'opère par le biais de *mises-à-jour*. Il faut donc, après chaque modification de la base de données, vérifier que celle-ci satisfait toujours bien l'ensemble des contraintes d'intégrité. Ceci nécessite l'exécution d'algorithmes de vérification de contraintes qui, même optimisés, restent coûteux. En outre, dans le cas où une contrainte n'est pas satisfaite après une modification, il faut annuler cette modification et restaurer l'état initial de la base de données. La prise en compte de l'intégrité induit donc un coût important.

Cependant, ce coût peut très souvent être diminué, voire totalement annulé. En effet, les contraintes n'imposent en général de conditions que sur certains aspects très spécifiques de la base de données. Aussi, il est peu probable qu'une modification opérée sur la base vienne remettre en cause toutes les contraintes à la fois. Il est aussi possible que certaines modifications, bien qu'elles touchent aux aspects "critiques" de certaines contraintes, respectent malgré tout ces contraintes. Ceci peut être le résultat de précautions particulières prises par le programmeur dans l'écriture de la modification, ou tout simplement parce que, naturellement, la modification va dans le sens de la contrainte. Dans ce cas, la vérification à l'exécution des contraintes peut être entièrement évitée. Il reste cependant à prouver formellement que la modification en question *préserve* effectivement telle ou telle contrainte. C'est le problème que nous nous proposons de résoudre dans cette thèse.

La grande variété des problèmes posés par les bases de données en a toujours fait un domaine d'application privilégié pour de nombreuses techniques développées dans d'autres disciplines. Entre autres, la Logique, la Théorie de la Complexité, et la Théorie des Langages sont constamment utilisées pour formaliser et résoudre des problèmes spécifiques aux bases de données. C'est dans cette perspective là que nous nous plaçons. Nous présentons dans ce travail une *application* des techniques de preuve de programmes au contexte des bases de données. Plus précisément, nous utilisons des techniques d'analyse statique de programmes qui ont été développées depuis trente ans, afin de prouver formellement qu'une modification de la base de données préserve un contrainte d'intégrité (si tel est le cas...).

Après avoir fait quelques rappels de logique dans le chapitre 2, nous faisons, dans le chapitre 3, le point sur un certain nombre de techniques d'analyse statique de programmes : la Logique de Hoare, le Calcul de Dijkstra et l'Interprétation Abstraite. Cette partie a pour finalité de donner au lecteur le bagage théorique minimum pour comprendre la suite de notre travail.

Ensuite, au chapitre 4, nous décrivons le cadre dans lequel nous nous plaçons et auquel nous appliquons les techniques mentionnées précédemment. Il s'agit d'un système de gestion de bases de données inspiré des systèmes orientés-objet. Le modèle de données sur lequel nous nous appuyons offre donc les concepts de type, de classe, d'objet, de méthode, d'héritage, etc., ainsi que la possibilité de définir des contraintes d'intégrité dans un langage inspiré de la logique du premier ordre. Nous définissons aussi un langage de programmation pour bases de données de type impératif. Ce langage permet de définir des *méthodes*, qui sont des programmes qui modifient la base de données. En outre, nous décrivons dans ce chapitre une technique de décomposition des formules et des instructions qui est une étape préliminaire à l'analyse statique. Cette décomposition a simplement pour but de réécrire les contraintes et les méthodes sous une forme plus élémentaire, bien qu'équivalente. Cette étape est essentielle pour bien élucider la sémantique des méthodes.

Au chapitre 5, nous présentons le concept central sur lequel toute notre approche est fondée : celui de *transformateur de prédicats*. Le concept de transformateur de prédicats a pour origine les travaux de [Dij76] sur la sémantique logique des langages de programmation. Un transformateur de prédicats est une fonction qui prend en entrée un programme (une méthode) et une formule, et renvoie une autre formule. Nous présentons deux transformateurs de prédicats différents :

- un transformateur de prédicats *avant*, noté $\mathcal{POST}$, qui prend en entrée une formule F, appelée la *pré-condition*, et une méthode m. Cette fonction calcule une autre formule, notée $\mathcal{POST}(m, F)$, appelée la *post-condition*. La fonction $\mathcal{POST}$ est construite pour capturer, de façon logique, la sémantique de m. Formellement, ceci se traduit par le fait que, si F est satisfaite par les données avant l'exécution de m, alors $\mathcal{POST}(m, F)$ sera nécessairement satisfaite après.

- un transformateur de prédicats arrière, noté $\mathcal{PRE}$. $\mathcal{PRE}$ prend en entrée une *post-condition* F et une méthode m, et renvoie une pré-condition suffisante sur les données en entrée de m pour que F soit satisfaite par les données en sortie. Cette pré-condition est notée $\mathcal{PRE}(m, F)$.

La majeure partie du chapitre 5 a pour but la définition de $\mathcal{POST}$ et de $\mathcal{PRE}$. Nous prouvons aussi que ces définitions sont correctes, c'est-à-dire que les transformateurs de prédicats sont un moyen fiable de capturer la sémantique des programmes. Les résultats essentiels de ce chapitre sont cependant les critères de préservation d'une contrainte C par une méthode m. En utilisant le transformateur de prédicats avant, nous prouvons que :

$$m \text{ préserve } C \text{ si } \mathcal{POST}(m, C) \Rightarrow C.$$

En utilisant le transformateur de prédicats arrière, nous prouvons un autre critère :

$$m \text{ préserve } C \text{ si } C \Rightarrow \mathcal{PRE}(m, C).$$

Ces implications peuvent être prouvées avec des techniques de démonstration automatique de théorèmes. À ce point, nous disposons donc d'une méthode fiable pour prouver qu'une méthode préserve une contrainte. Le critère mentionné précédemment peut déjà, en théorie, nous servir pour éviter de vérifier systématiquement toutes les contraintes lors de l'exécution d'une méthode. En pratique, nous n'avons fait qu'un tiers du travail.

Les transformateurs de prédicats ont pour avantage d'avoir une définition purement syntaxique, et donc aisée à implanter directement. Les transformateurs de prédicats prennent les formules en entrée comme des chaînes de caractères dépourvues de sens, et produisent d'autres formules sans aucune considération sémantique. Ce type de fonctionnement a comme défaut le caractère explosif des calculs effectués. Une mise en œuvre *naïve* de notre approche ne fournit donc que des résultats très pauvres. C'est pour cette raison que nous redéfinissons les transformateurs de prédicats au chapitre 6. Nous nous concentrons en fait sur la redéfinition de $\mathcal{POST}$. La redéfinition de $\mathcal{PRE}$ est faite dans le même esprit. Cette nouvelle définition des transformateurs de prédicats a comme but une meilleure prise en compte de la sémantique des formules en entrée.

Nous présentons ensuite l'implantation qui a été faite de notre approche. Soulignons le fait que cette implantation n'est pas une simple illustration, mais nous informe véritablement sur la faisabilité de notre approche. Le simple fait d'avoir à utiliser un formalisme du premier ordre est en soi décourageant, car les démonstrateurs automatiques de théorèmes sont en général incapables de prouver des théorèmes non triviaux. Cependant, les résultats que nous obtenons, sans être miraculeux, établissent de façon certaine que notre approche est un moyen effectif d'optimiser la gestion de l'intégrité. En outre, le cadre des bases de données orientées objet semble être le cadre d'application idéal des techniques d'analyse statique que nous présentons dans ce travail. En effet, dans ce cadre, les traitements (les méthodes) qui sont opérés sont de taille réduite et très modulaire. Leur sémantique est donc plus aisée à capturer. Enfin, au chapitre 7, nous donnerons les conclusions et perspectives de ce travail. Ce travail à en outre fait l'objet des publications suivantes : [BDS94, BDS95, BS96, BS97].

Chapitre 2

Rappels de logique

Ce chapitre a pour but de rappeler brièvement quelques notions de logique du premier ordre. Il n'a pas pour ambition d'être une introduction à la logique, ni d'enseigner aux profanes les bases de cette discipline. De nombreux ouvrages spécialisés, comme par exemple [CL93], [vD83], ou encore [Lal90], sont beaucoup plus appropriés.

La logique du premier ordre est un langage permettant de faire des affirmations sur des domaines quelconques. Les affirmations en question, qu'on appelle *formules*, sont construites selon une syntaxe particulière que nous décrirons en section 2.1. Ensuite, en section 2.3, nous donnerons un moyen systématique d'évaluer la justesse d'une affirmation par rapport à un domaine, en lui attribuant une *valeur de vérité* (`vrai` ou `faux`). Un fait important est que la véracité de certaines affirmations est indépendante du domaine particulier auxquelles elles s'appliquent : on dit qu'elles sont *universellement valides*. En section 2.4, nous décrirons un moyen automatique de prouver qu'une formule est universellement valide: la méthode des tableaux.

2.1 Syntaxe des formules de logique du premier ordre

Comme tout langage, la logique du premier ordre utilise un *alphabet*, c'est-à-dire un ensemble de symboles qui, collés bout à bout, permettent de construire des formules, qui sont en fait les phrases de ce langage.

Définition 1 Un langage du premier ordre est défini par:

– Un ensemble de symboles de variables v_1, v_2, ...

– Un ensemble de symboles de constantes c_1, c_2, ...

– Un ensemble de symboles de fonctions: f_1, f_2, ...

– Un ensemble de symboles de prédicats: p_1, p_2, ...

– Les connecteurs logiques: $\neg$ $\wedge$, $\vee$, $\Rightarrow$, ...

– Les quantificateurs: $\forall$, $\exists$.

– Des symboles de ponctuation “,”, “(”, ...

Pour simplifier, nous ne considérerons qu'un seul langage. L'ensemble des définitions de ce chapitre seront bien sûr relatives à ce langage. Nous attribuons en outre à chaque symbole de fonction et de prédicat une *arité*, qui désigne le nombre d'arguments auxquels ce symbole doit s'appliquer. Dans un premier temps, ces symboles, assemblés selon des règles précises, vont nous permettre de décrire les objets des affirmations. À cette fin, nous définissons les termes.

Définition 2 Les termes sont définis comme suit:

– Toute constante est un terme.

– Toute variable est un terme.

– Si f est un symbole de fonction d'arité n, et t_1,...,t_n sont des termes, alors $f(t_1, ..., t_n)$ est un terme.

Nous utiliserons, dans certaines parties de ce travail, la notion de *profondeur* d'un terme.

Définition 3 La profondeur d'un terme t, écrite $\mathrm{profondeur}(t)$, est définie comme suit.

– si t est une variable ou une constante, alors $\mathrm{profondeur}(t) = 0$.

– si $t \equiv f(t_1, ..., t_n)$, alors $\mathrm{profondeur}(t) = \max(\mathrm{profondeur}(t_1), ..., \mathrm{profondeur}(t_n)) + 1$.

Nous sommes maintenant en mesure de définir les phrases de notre langage:

Définition 4 Les formules sont définies comme suit:

– Si p est un symbole de prédicat d'arité n, et t_1,...,t_n sont des termes, alors $p(t_1, ..., t_n)$ est une formule, dite formule *atomique* ou *atome*.

– Si F et G sont des formules, alors $\neg F$, $F \wedge G$, $F \vee G$, ...sont des formules.

– Si F est une formule, et x est une variable, alors $(\forall x : F)$ et $(\exists x : F)$ sont des formules.

2.2 Variables libres, variables liées, substitutions

Les variables des formules peuvent apparaître sous deux formes non exclusives: elles représentent soit un seul objet arbitraire, ou elles varient sur un ensemble d'objets. Le premier type de variable est dit *libre*, le second est *lié*.

Définition 5 Soit F une formule. L'ensemble $\mathcal{VL}(F)$ des *variables libres* de F est défini comme suit.

– Si $F \equiv p(t_1, ..., t_n)$ alors $\mathcal{VL}(F) = \mathcal{V}(F)$, où $\mathcal{V}(F)$ dénote l'ensemble des variables de F.

– Si $F \equiv F_1 \wedge F_2$ ou $F \equiv F_1 \vee F_2$ alors $\mathcal{VL}(F) = \mathcal{VL}(F_1) \cup \mathcal{VL}(F_2)$.

– Si $F \equiv \neg G$ alors $\mathcal{VL}(F) = \mathcal{VL}(G)$.

– Si $F \equiv \forall x : G$ ou $F \equiv \exists x : G$ alors $\mathcal{VL}(F) = \mathcal{VL}(G) \setminus \{x\}$.

L'ensemble des variables liées d'une formule peut être défini d'une façon similaire. Plus simplement, nous pouvons dire qu'une variable est liée dans une formule à partir du moment où elle y est quantifiée. Une variable peut évidement être libre et liée en même temps. Une formule qui n'a pas de variables libres est *close*. Si F est une formule avec comme variables libres x_1,...,x_n, alors nous pourrons écrire F de la façon suivante:

$$F(x_1, ..., x_n)$$

Cette notation à pour avantage de bien faire ressortir l'ensemble des variables libres de F. En outre, à une variable libre d'une formule, nous pouvons substituer un terme. Une substitution simultanée des termes t_1, ..., t_n, respectivement aux variables x_1,...,x_n dans F pourra être notée de deux façons différentes:

- $F[t_1/x_1, ..., t_n/x_n]$
- $F(t_1, ..., t_n)$

L'expression $[t_1/x_1, ..., t_n/x_n]$ représente la substitution en question.

2.3 Sémantique des formules de logique du premier ordre

Nous devons maintenant donner un sens à ces formules, c'est-à-dire déterminer, pour un domaine particulier, leur véracité. Cela se fait par la notion d'*interprétation*.

Définition 6 Une interprétation $\mathcal{I}$ est définie par:

– Un ensemble non vide D, appelé le domaine de l'interprétation.

– Une fonction qui attribue à chaque symbole de constante c un élément de D, appelé l'interprétation de c dans $\mathcal{I}$.

– Une fonction qui attribue à chaque symbole de fonction f d'arité n, une fonction de D^n dans D, appelée l'interprétation de f dans $\mathcal{I}$.

– Une fonction qui attribue à chaque symbole de prédicat p d'arité n un sous-ensemble de D^n, appelé l'interprétation de p dans $\mathcal{I}$.

Par convention, nous noterons $\mathcal{I}(s)$ l'interprétation d'un symbole s dans $\mathcal{I}$.

Les variables nécessitent un traitement particulier. En effet, par définition, elles n'ont pas pour but de représenter un élément fixe du domaine. Il peut cependant être utile d'attacher une variable à un élément donné.

Définition 7 Une assignation dans $\mathcal{I}$ est une fonction α qui attribue à chaque variable d'un ensemble un élément du domaine de $\mathcal{I}$.

Une assignation “complète” donc en quelque sorte une interprétation. Elle n’a pas à être définie pour toutes les variables de notre langage, mais seulement pour un ensemble fini. Nous noterons $\mathcal{I}_\alpha$ l’interprétation $\mathcal{I}$ munie de l’assignation α.

Définition 8 Soit $\mathcal{I}$ une interprétation, t un terme tel que $t \equiv f(t_1, ..., t_n)$, et α une assignation pour les variables de t. L’interprétation de t dans $\mathcal{I}$ muni de α, est défini par:

$$\mathcal{I}_\alpha(t) = \mathcal{I}_\alpha(f)(\mathcal{I}_\alpha(t_1), ..., \mathcal{I}_\alpha(t_n))$$

Nous sommes maintenant en mesure de définir la valeur de vérité d’une formule dans une interprétation.

Définition 9 Soit $\mathcal{I}$ une interprétation, F une formule, et α une assignation des variables libres de F dans I. L’interprétation de F dans $\mathcal{I}_\alpha$ est une fonction qui associe à F une valeur de vérité “`vrai`” ou “`faux`” de la façon suivante:

- Si $F \equiv p(t_1, ..., t_n)$ alors $\mathcal{I}_\alpha(F) = \texttt{vrai}$ si et seulement si $(\mathcal{I}_\alpha(t_1), ..., \mathcal{I}_\alpha(t_n)) \in \mathcal{I}_\alpha(p)$.
- Si $F \equiv F_1 \wedge F_2, \mathcal{I}_\alpha(F) = \texttt{vrai}$ si et seulement si $\mathcal{I}_\alpha(F_1) = \texttt{vrai}$ et $\mathcal{I}_\alpha(F_2) = \texttt{vrai}$.
- Si $F \equiv F_1 \vee F_2, \mathcal{I}_\alpha(F) = \texttt{vrai}$ si et seulement si $\mathcal{I}_\alpha(F_1) = \texttt{vrai}$ ou $\mathcal{I}_\alpha(F_2) = \texttt{vrai}$.
- Si $F \equiv \neg G, \mathcal{I}_\alpha(F) = \texttt{vrai}$ si et seulement si $\mathcal{I}_\alpha(G) = \texttt{faux}$.
- Si $F \equiv \forall x : G(x)$ alors $\mathcal{I}_\alpha(F) = \texttt{vrai}$ si et seulement si pour toute assignation de x a un élément d de D, notée $x \mapsto d$, alors $\mathcal{I}_{\alpha \cup \{x \mapsto d\}}(G) = \texttt{vrai}$. (on suppose que α ne définit pas déjà une assignation pour x).
- Si $F \equiv \exists x : G(x)$ alors $\mathcal{I}_a(F) = \texttt{vrai}$ si et seulement si il existe une assignation de x a un élément d de D, notée $x \mapsto d$, telle que $\mathcal{I}_{\alpha \cup \{x \mapsto d\}}(G) = \texttt{vrai}$. (on suppose que α ne définit pas déjà une assignation pour x).

Il est clair qu’on a soit $\mathcal{I}_\alpha(F) = \texttt{vrai}$, soit $\mathcal{I}_\alpha(F) = \texttt{faux}$. Par la suite, nous noterons le fait que $\mathcal{I}_\alpha(F) = \texttt{vrai}$ de la façon suivante:

$$\mathcal{I}_\alpha \models F$$

Nous dirons aussi que l’interprétation $\mathcal{I}$ munie de l’assignation α, est un *modèle* de F, ou *satisfait* F.

2.4 Une technique de démonstration automatique de théorèmes: la méthode des tableaux

Nous avons vu dans la section précédente comment évaluer la valeur de vérité d’une formule en fonction d’une interprétation particulière. Un fait remarquable est que la valeur de vérité de certaines formules est indépendante de l’interprétation particulière considérée. C’est le cas par exemple des formules de la forme: $F \wedge \neg F$, qui sont toujours fausses. Nous donnons dans cette section un moyen systématique de prouver qu'une formule est toujours vraie (on dira aussi “valide”). Cette méthode s'appelle la méthode des tableaux, et procède en fait par réfutation : afin de prouver qu'une formule est toujours vraie, cette méthode prouve en fait que sa négation est toujours fausse. Par manque de place, nous ne présentons cette méthode que très succinctement, sous la forme d'une recette de cuisine à appliquer “bêtement”. Une exposition complète de cette méthode peut être trouvée dans [Fit90].

Un tableau est en fait un arbre étiqueté. Les étiquettes sont des formules. Pour suivre clairement le déroulement d'une preuve, nous numéroterons chaque nœud de l'arbre. Une preuve commence par une branche avec un seul nœud, qui est étiqueté par la formule à réfuter. Un preuve consiste en fait à faire “grandir” cet arbre en appliquent des *règles d'expansion de branches*. Ces règles, qui portent respectivement les noms (très évocateurs) de α, β, γ et δ, sont simples:

α : Si une formule $F_1 \wedge F_2$ apparaît sur une branche, ajouter F_1 et F_2 à l'extrémité de la branche sur deux nœuds différents.

β : Si une formule $F_1 \vee F_2$ apparaît sur une branche, ajouter deux autres branches à l'extrémité de cette branche, l'une comportant F_1, et l'autre F_2.

γ : Si une formule $\forall x : F(x)$ apparaît sur une branche, ajouter $F(u)$ à l'extrémité de la branche. u est une nouvelle variable n'apparaissant pas dans l'arbre d'origine.

δ : Si une formule $\exists x : F(x)$ apparaît sur une branche, ajouter $F(f(u_1, ..., u_n))$ à l'extrémité de la branche. f est un nouveau symbole de fonction, et u_1,...,u_n sont les variables libres de $\exists x : F(x)$.

Ces règles sont représentées graphiquement en figure 2.1. En outre, nous numéroterons les formules et étiquetterons les arcs de l'arbre pour bien indiquer la manière dont les nœuds sont obtenus (à partir de quelle formule et par application de quelle règle). Cet étiquetage est illustré en figure 2.1.

$$\begin{array}{cccc}
\begin{array}{c} \dots \\ \downarrow \\ i\ \boxed{F_1 \wedge F_2} \\ \downarrow \\ \dots \\ \downarrow \\ \boxed{\cdots} \\ \downarrow \alpha_1(i) \\ \boxed{F_1} \\ \downarrow \alpha_2(i) \\ \boxed{F_2} \end{array}
&
\begin{array}{c} \dots \\ \downarrow \\ i\ \boxed{F_1 \vee F_2} \\ \downarrow \\ \dots \\ \downarrow \\ \boxed{\cdots} \\ \beta_1(i) \swarrow \quad \searrow \beta_2(i) \\ \boxed{F_1} \qquad\quad \boxed{F_2} \end{array}
&
\begin{array}{c} \dots \\ \downarrow \\ i\ \boxed{\forall x : F(x)} \\ \downarrow \\ \dots \\ \downarrow \\ \boxed{\cdots} \\ \downarrow \gamma(i) \\ \boxed{F(u)} \end{array}
&
\begin{array}{c} \dots \\ \downarrow \\ i\ \boxed{\exists x : F(x)} \\ \downarrow \\ \dots \\ \downarrow \\ \boxed{\cdots} \\ \downarrow \gamma(i) \\ \boxed{F(f(u_1, ..., u_n))} \end{array}
\end{array}$$

FIG. 2.1 - *Règles α, β, γ et δ.*

Moyennant certaines modifications (changer $A \Rightarrow B$ en $\neg A \vee B$, $\neg(A \wedge B)$ en $\neg A \vee \neg B$, etc.), ces règles peuvent toujours être appliquées. Nous pouvons aussi définir d'une manière similaire des règles d'expansion de tableaux pour les disjonction et conjonctions multiples: $F_1 \vee ... \vee F_n$ et $F_1 \wedge ... \wedge F_n$.

La règle γ est en fait une règle de généralisation, et la règle δ correspond à une sorte de Skolemisation. De façon générale, on ne doit appliquer chaque règle qu'au plus une fois par formule, sauf pour la règle γ. Celle-ci peut être appliquée à la même formule un nombre indéfini de fois (ceci est lié à l'indécidabilité de la logique du premier ordre). En fait, la profondeur de recherche d'une preuve peut être définie comme le nombre de fois que la règle γ est appliquée.

Définition 10 Un tableau est *fermé* si il existe une substitution u telle que chaque branche comporte deux littéraux L_1 et L_2 pour lesquels $L_1u \equiv \neg L_2u$.

Il reste à montrer comment cette méthode peut être utilisée pour montrer qu'une formule est toujours fausse, ou insatisfaisable.

Théorème 1 Soit F une formule et T un tableau avec F comme racine. Si T est fermé, alors F est insatisfaisable.

Ce théorème établit que la méthode des tableaux est *correcte*. [Fit90] en donne une preuve détaillée. Nous pouvons en outre montrer que toute formule insatisfaisable est la racine d'un tableau fermé, c'est-à-dire que la méthode des tableaux est complète.

Exemple 1 Voyons maintenant un exemple d'application de la méthode des tableaux. Considérons la formule suivante:

$$(p(\mathsf{0}) \wedge (\forall x : p(x) \Rightarrow p(s(x)))) \Rightarrow p(s(s(\mathsf{0})))$$

où $\mathsf{0}$ est une constante, p un symbole de prédicat, et s un symbole de fonction, censé représenter la fonction successeur. Cette formule est toujours vraie. Elle est prouvée en figure 2.2, en démontrant en fait que sa négation est insatisfaisable. Ce tableau est fermé pour la substitution: $[\mathsf{0}/u_1, s(\mathsf{0})/u_2]$. Pour cette substitution, **7** est contradictoire avec **2**, **9** est contradictoire avec **8**, et **10** est contradictoire avec **4**. Notons au passage que nous avons du appliquer la règle gamma deux fois sur la même formule (la formule **3**, qui aboutit aux formules **5** et **6**). Intuitivement, la formule **3** spécifie un axiome dont on se sert deux fois pour obtenir la preuve finale: une première fois pour obtenir $p(s(\mathbf{0}))$ à partir de $p(\mathbf{0})$, et une deuxième fois pour obtenir $p(s(s(\mathbf{0})))$ à partir de $p(s(\mathbf{0}))$. Ce dernier littéral est contradictoire avec $\neg p(s(s(\mathbf{0})))$, et la preuve est obtenue.

Il existe de nombreuses versions différentes de la méthode des tableaux. En ce qui concerne notre approche, elle utilise la méthode des tableaux avec égalité. La prise en compte de l'égalité se fait par le biais d'une nouvelle règle d'expansion de branche. Si deux formules $t_1 = t_2$ et $F(t_1)$, d'indices respectifs i et j apparaissent dans une branche, alors nous pouvons ajouter à l'extrémité de cette branche la formule $F(t_2)$. Cette règle d'expansion de branches est aussi appelée *règle de substitution*. Celle-ci sera notée **égalité**(i,j) dans les tableaux.

Enfin, il est important de préciser que, la logique du premier ordre étant indécidable, il est difficile de parler du coût d'une preuve. Celui-ci peut être arbitrairement grand, et même être infini (sinon on pourrait définir une procédure de décision qui, passé un certain temps de calcul, conclurait qu'une formule est satisfaisable). La seule certitude dans ce domaine est que la démonstration automatique est, en général, *extrêmement* coûteuse, et qu'elle ne peut donc être appliquée qu'à des formules de taille relativement réduite.

$$\mathbf{1}\ \boxed{p(0) \wedge (\forall x : p(x) \Rightarrow p(s(x))) \wedge \neg p(s(s(0)))}$$

$$\downarrow \alpha_1(\mathbf{1})$$

$$\mathbf{2}\ \boxed{p(0)}$$

$$\downarrow \alpha_2(\mathbf{1})$$

$$\mathbf{3}\ \boxed{\forall x : p(x) \Rightarrow p(s(x))}$$

$$\downarrow \alpha_3(\mathbf{1})$$

$$\mathbf{4}\ \boxed{\neg p(s(s(0)))}$$

$$\downarrow \gamma(\mathbf{3})$$

$$\mathbf{5}\ \boxed{p(u_1) \Rightarrow p(s(u_1))}$$

$$\downarrow \gamma(\mathbf{3})$$

$$\mathbf{6}\ \boxed{p(u_2) \Rightarrow p(s(u_2))}$$

$$\beta_1(\mathbf{5}) \swarrow \qquad \searrow \beta_2(\mathbf{5})$$

$$\mathbf{7}\ \boxed{\neg p(u_1)} \qquad \mathbf{8}\ \boxed{p(s(u_1))}$$

$$\beta_1(\mathbf{6}) \swarrow \qquad \searrow \beta_2(\mathbf{6})$$

$$\mathbf{9}\ \boxed{\neg p(u_2)} \qquad \mathbf{10}\ \boxed{p(s(u_2))}$$

FIG. 2.2 - *Exemple de tableau.*

Chapitre 3

État de l'art

3.1 Méthodes formelles pour la vérification automatique de programmes

De nombreux outils formels permettent aujourd'hui de vérifier automatiquement la correction des programmes. Avant d'exposer l'application de ces outils à notre problème, nous donnons dans cette section une courte synthèse des différentes méthodes d'analyse de programmes que nous utiliserons. Notre approche pour prouver qu'une mise à jour respecte une contrainte d'intégrité est inspirée de la logique de Hoare [Hoa69], du calcul de Dijkstra[Dij76], et de l'interprétation abstraite[CC77]. Nous ne tenterons pas de faire une synthèse détaillée de tous les travaux qui ont été publiés dans ces trois domaines. Étant donné leur nombre, la tâche serait presque impossible. D'autre part, notre contribution ne se situe pas dans ces domaines mais dans l'application de ces domaines aux bases de données.

3.1.1 La logique de Hoare et le calcul de Dijkstra

Nous présentons dans cette section les bases de l'analyse logique des programmes telles qu'elles ont été établies par Hoare et Dijkstra. Hoare propose dans [Hoa69] une méthode *axiomatique* dans le but de prouver formellement la correction de programmes. Cette méthode est elle-même inspirée de la sémantique déductive des programmes proposée par Floyd [Flo67]. La méthode de Hoare permet non seulement de faire de la vérification formelle de programmes, mais elle facilite aussi la conception de nouveaux langages de programmation. C'est en effet une manière élégante et concise de spécifier leur sémantique.

Un exemple: le langage `while`

Afin d'illustrer les concepts principaux de la logique de Hoare, nous allons nous placer dans le contexte d'un petit langage "jouet", appelé le langage `while`. Ses instructions sont définies en figure 3.1. Dans un souci de rigueur, ses expressions sont définies en figure 3.2.

Un programme du langage `while` permet de faire évoluer un *environnement*. Un environnement e assigne à chaque variable x utilisée dans le programme une valeur entière $e(x)$. Voici un exemple d'environnement:

$$e = \{x_0 \mapsto 5, x_1 \mapsto 1, x_3 \mapsto 2\}$$

instruction	::=	*variable* `:=` *expression*
	::=	*instruction* `;` *instruction*
	::=	`if` *expression* `then` *instruction* `else` *instruction*
	::=	`while` *expression* `do` *instruction*

FIG. 3.1 - *Les instructions du langage* `while`

expression	::=	*variable*
	::=	*constante*
	::=	*expression opération expression*
opération	::=	$+ \mid - \mid * \mid / \mid = \mid \neq \mid > \mid < \mid \geq \mid \leq$
variable	::=	$x_0 \mid x_1 \mid x_2$ ...
constante	::=	0 | 1 | 2 ... | `error`

FIG. 3.2 - *Les expressions du langage* `while`

On étend naturellement cet environnement à l'ensemble des expressions du langage. Ainsi, les constantes (1, 2,...) et les opérateurs ($+$,$-$,$*$,$/$) sont interprétés de manière classique. La valeur `error` dénote une erreur lors d'une affectation (une division par 0, par exemple). Par exemple, on a:

$$e(x_0 + x_1) = e(x_0) + e(x_1) = 5 + 1 = 6$$

Chaque instruction prend un environnement en entrée et permet de modifier cet environnement. Étant donné le classicisme du langage `while`, nous n'indiquons pas sa sémantique formelle. Pour tout environnement e, nous noterons:

$$[\![I]\!]e$$

l'environnement résultant de l'exécution de *I* sur l'environnement e.

Les triplets de Hoare

Afin de pouvoir décrire en général le comportement d'un programme, indépendamment de son comportement vis-à-vis d'un environnement particulier, Hoare utilise des formules du type:

$$P\{I\}Q$$

P et *Q* sont des formules de logique et *I* est un programme écrit dans le langage de programmation, en l'occurrence le langage `while`. La signification intuitive de ce triplet est que si *P* est satisfaite par l'environnement initial, avant l'exécution de *I*, alors *Q* est satisfaite par l'environnement final, après l'exécution de *I*. Formellement:

Définition 11 Soient *P* et *Q* deux formules de logique et *I* un programme. Alors $P\{I\}Q$ est vrai si et seulement si pour tout environnement e tel que $e \models P$ on a $[\![I]\!]e \models Q$.
P s'appelle la *pré-condition* du triplet et *Q* la *post-condition*.

Le langage dans lequel P et Q sont écrits s'appelle la *logique sous-jacente* ou le langage d'assertion. Il s'agit souvent de la logique du premier ordre ou du langage de l'arithmétique de Peano. Lorsqu'on fait de l'analyse statique, on s'intéresse plutôt aux pré et post-conditions qu'à des ensembles d'environnements. Ceci permet de représenter *finiment* un ensemble potentiellement infini d'environnements [1].

Un système de preuve pour `while`

Afin de prouver une assertion de la forme $P\{I\}Q$, nous avons le choix entre deux solutions:

- exécuter l'instruction I sur tout environnement e satisfaisant P, et vérifier que l'environnement résultant, noté $[\![I]\!]e$, satisfait Q.
- Utiliser un système formel nous permettant d'obtenir une preuve de $P\{I\}Q$.

Si la première solution semble conceptuellement plus simple, elle ne peut par contre être mise en œuvre dans des délais raisonnables. La seconde solution est moins évidente, car l'obtention d'une preuve d'un théorème dans un système formel n'est jamais une chose aisée. Elle suppose l'établissement de nombreux lemmes... et une bonne dose d'intuition. C'est cependant la seule solution susceptible de fournir une preuve avant la fin du monde. Un système de preuve pour le langage `while` est donné figure 3.3. Ce système est tiré de [Hoa69].

Axiome d'affectation:	$P[t/x]\{x{:=}t\}P$
Règle de composition:	$\dfrac{P\{I_1\}R \quad R\{I_2\}Q}{P\{I_1;I_2\}Q}$
Règle du `if`:	$\dfrac{(P \wedge F)\{I_1\}Q \quad (P \wedge \neg F)\{I_2\}Q}{P\{\texttt{if}\ F\ \texttt{then}\ I_1\ \texttt{else}\ I_2\}Q}$
Règle du `while`:	$\dfrac{(P \wedge F)\{I\}P}{P\{\texttt{while}\ F\ \texttt{do}\ I\ \texttt{od}\}(P \wedge \neg F)}$
Règle de conséquence:	$\dfrac{P \Rightarrow P' \quad P'\{I\}Q' \quad Q' \Rightarrow Q}{P\{I\}Q}$

FIG. 3.3 - *Système de preuve pour le langage* `while`.

Comme d'habitude, $P[t/x]$ dénote la formule obtenue en substituant toutes les occurrences libres de x dans P par le terme t. Ce système formel est très simple mais permet d'obtenir des preuves rigoureuses de programmes. Considérons par exemple le programme `divise` de la figure 3.4.

1. Les systèmes informatiques ne permettent jamais de stocker effectivement des ensembles infinis d'environnements. Mais même dans le cas où le nombre d'environnements est fini, celui-ci est en général très grand. Dans tous les cas, il est donc préférable d'utiliser, pour représenter ces environnements, des formules de logique.

```
a:=0;
b:=x;
while b ≥ y
do b:=b − y;
   a:=a + 1;
od
```

FIG. 3.4 - *Programme* `divise`.

Ce programme est censé calculer la division entière de x par y. On peut effectivement prouver qu'à la fin du programme, a est la division entière de x par y et que b est le reste de cette division. Il est important de bien comprendre la manière de fonctionner de la logique de Hoare. À cette fin, nous donnons dans la suite la preuve détaillée que le programme `divise` fait bien ce qu'il doit faire. Cet exemple mettra en valeur les qualités et les défauts de cette approche.

Formellement, nous voulons prouver:

$$(x \geq 0 \ \wedge \ y \geq 0)\{\texttt{divise}\}(a * y + b = x \ \wedge \ y > b \geq 0) \tag{3.1}$$

Cette preuve passe par l'établissement d'un bon nombre de lemmes. Pour commencer, avec l'axiome d'affectation nous avons:

$$(0 * y + x = x \ \wedge x \geq 0)\{a{:=}0\}(a * y + x = x \ \wedge \ x \geq 0) \tag{3.2}$$

Pour obtenir la formule de gauche (la pré-condition), nous avons simplement remplacé a par 0 dans la formule de droite (la post-condition). Avec le même axiome, nous obtenons:

$$(a * y + x = x \ \wedge x \geq 0)\{b{:=}x\}(a * y + b = x \ \wedge \ b \geq 0) \tag{3.3}$$

Notons dors-et-déjà que, si nous établissons ces lemmes, c'est qu'ils seront utilisés par la suite. Nous pouvons cependant nous demander si nous ne risquons pas d'établir des lemmes inutiles, dont l'absence n'aurait aucune incidence sur l'obtention de la preuve finale. Laissons pour l'instant ce problème de côté... La partie droite du lemme 3.2 étant la même que la partie gauche du lemme 3.3, nous pouvons appliquer la règle de composition, ce qui nous donne le lemme suivant:

$$(0 * y + x = x \ \wedge x \geq 0)\{a{:=}0;\ b := x\}(a * y + b = x \ \wedge \ b \geq 0) \tag{3.4}$$

Le lemme 3.5 n'utilise pas le système formel de Hoare mais est un théorème simple de l'arithmétique:

$$(x \geq 0 \wedge y \geq 0) \ \Rightarrow \ (0 * y + x = x \wedge x \geq 0) \tag{3.5}$$

Avec les lemmes 3.4 et 3.5 nous obtenons par la règle de conséquence:

$$(x \geq 0 \wedge y \geq 0)\{a{:=}0;\, b{:=}x\}(a * y + b = x \ \wedge \ b \geq 0) \tag{3.6}$$

Par l'utilisation de l'axiome d'affectation, nous obtenons:

$$((a + 1) * y + b - y = x \ \wedge b - y \geq 0)\{b{:=}b - y\}((a + 1) * y + b = x \ \wedge \ b \geq 0) \tag{3.7}$$

Arrivé à ce point là, nous sommes en droit de nous demander où va la preuve. Les lemmes semblent être établis au hasard, tantôt à l'intérieur du système de Hoare, tantôt en faisant des "emprunts" à d'autres systèmes formels (l'arithmétique de Peano, en l'occurrence). Poursuivons...
Encore par l'axiome d'affectation, nous avons:

$$((a+1)*y+b=x \ \wedge b \geq 0)\{a{:=}a+1\}(a*y+b=x \ \wedge \ b \geq 0) \tag{3.8}$$

La post-condition du lemme 3.7 se retrouvant comme pré-condition du lemme 3.8, nous appliquons la règle de composition:

$$\begin{aligned}&((a+1)*y+b-y=x \ \wedge b-y \geq 0)\\ &\quad \{b{:=}b-y;a{:=}a+1\}(a*y+b=x \ \wedge \ b \geq 0)\end{aligned} \tag{3.9}$$

Un autre théorème de l'arithmétique que nous utiliserons est:

$$(a*y+b=x \wedge b \geq 0) \Rightarrow ((a+1)*y+b-y=x \wedge b-y \geq 0) \tag{3.10}$$

Avec les lemmes 3.9 et 3.10, nous obtenons par la règle de conséquence:

$$\begin{aligned}&(a*y+b=x \wedge b \geq 0)\\ &\quad \{b{:=}b-y;a{:=}a+1\}(a*y+b=x \ \wedge \ b \geq 0)\end{aligned} \tag{3.11}$$

Avec le lemme 3.11 et la règle du `while` nous déduisons:

$$\begin{aligned}&(a*y+b=x \wedge b \geq 0)\\ &\quad \{\texttt{while}\ b \geq y\ \texttt{do}\ b{:=}b-y;a{:=}a+1\ \texttt{od}\}(a*y+b=x \ \wedge \ b \geq 0 \wedge b < y)\end{aligned} \tag{3.12}$$

Nous n'avons plus qu'à appliquer la règle de composition aux lemmes 3.6 et 3.12 pour obtenir le résultat recherché.

Ce petit exemple nous suggère un certain nombre de remarques concernant la logique de Hoare. Tout d'abord, pour peu que le système proposé soit correct (ce qui est le cas), nous avons obtenu une preuve tout à fait rigoureuse du programme `divise`. Le raisonnement est assez difficile à suivre car très détaillé et sans fil conducteur apparent. Il correspond cependant au concept idéal de preuve, qui établit une chose complexe avec une série de truismes.

Nous pouvons aussi constater que le déroulement d'une preuve n'est pas déterminé à l'avance. Une même preuve peut être obtenue de différentes manières. Ce système formel ne suggère donc pas, dans son état actuel, un moyen automatique, c'est-à-dire un algorithme, pour faire de la vérification de programmes. En outre, les preuves de programme utilisent des lemmes extérieurs à la logique de Hoare, qu'on ne sait pas toujours très bien démontrer. En particulier, en ce qui concerne l'arithmétique de Peano, selon le théorème d'incomplétude de Gödel, aucun système de preuve ne peut démontrer toutes les formules vraies dans $\mathbb{N}$. De façon générale, la logique de Hoare est toujours un sur-ensemble d'une logique déjà existante (c'est ce que nous avons appelé la logique sous-jacente). Si elle profite des bonnes propriétés de cette logique, elle hérite aussi de tous ses défauts, et parfois en pire !

L'impression de non-déterminisme que donne la logique de Hoare peut être partiellement résolue. En effet, la preuve donnée précédemment suivait une démarche *ascendante*, où l'on prouvait des lemmes sans trop savoir à quoi ils serviraient. Une démarche *descendante* permet de se limiter aux lemmes dont on a vraiment besoin. Il est en fait possible de ramener en partie une preuve de logique de Hoare à un calcul, c'est-à-dire un algorithme dont les étapes sont totalement déterminées. Ce calcul est connu sous le nom de *calcul de Dijkstra*[Dij76]. Cependant, il faudra toujours, dans certaines étapes d'une preuve de programme, prouver des théorèmes dans la logique sous-jacente (dans le système de logique de Hoare donné précédemment, cette étape apparaît dans les implications de la règle de conséquence). À moins que cette logique soit décidable, ces preuves ne pourront être obtenues via un algorithme. Dans le meilleur des cas, un démonstrateur automatique de théorèmes pourra pallier ce problème. Dans le pire des cas, il faudra faire les preuves “à la main”.

Le calcul de Dijkstra

Le calcul de Dijkstra[Dij76, DS90] est fondé sur la notion de *transformateur de prédicats*. Conçu pour le même but que la logique de Hoare, il offre un cadre fonctionnel à la preuve de programme. Un transformateur de prédicats *avant* (resp. *arrière*) est une fonction qui prend en entrée un programme I et une formule P (resp. Q), et rend une formule $\mathcal{POST}(I,P)$ (resp. $\mathcal{PRE}(I,Q)$). On dit que le transformateur de prédicats avant $\mathcal{POST}$ est *correct* si, lorsque P est satisfaite par les données en entrée de I, alors $\mathcal{POST}(I,P)$ est satisfaite par les données en sortie de I. $\mathcal{POST}$ peut donc générer une post-condition en fonction d'une pré-condition et d'un programme.

À l'inverse, le transformateur de prédicats arrière est correct si, lorsque $\mathcal{PRE}(I,Q)$ est satisfaite par les données en entrée de I, alors Q est satisfaite par les données en sortie de I. $\mathcal{PRE}$ peut donc générer une pré-condition en fonction d'une post-condition et d'un programme.

Le lien entre Calcul de Dijkstra et logique de Hoare est naturel. La correction des transformateurs de prédicats peut s'exprimer très simplement par des triplets de Hoare:

$$P\{I\}\mathcal{POST}(I,P)$$

$$\mathcal{PRE}(I,Q)\{I\}Q$$

Alors qu'avec les triplets de Hoare, le lien réciproque entre pré et post-conditions n'apparaissait pas clairement (sauf peut-être dans la règle d'affectation), il est bien mis en évidence par le calcul de Dijkstra. Les transformateurs de prédicats peuvent ainsi servir à mieux diriger les étapes d'une preuve en logique de Hoare. Par exemple, considérons une formule de la logique de Hoare:

$$P\{I_1;I_2\}Q$$

Afin de prouver cette formule. Nous utilisons la règle de composition:

$$\frac{P\{I_1\}R \quad R\{I_2\}Q}{P\{I_1;I_2\}Q}$$

Nous devons trouver une formule R telle que $P\{I_1\}R$ et $R\{I_2\}Q$. Le système formel n'indique pas quelle est la “bonne” formule à choisir pour mener à bien cette preuve. Le calcul de Dijkstra permet de déterminer cette formule. Elle peut être obtenue de deux façons différentes:

$$R = \mathcal{POST}(I_1,P) \text{ ou } R = \mathcal{PRE}(I_2,Q)$$

Nous ne donnerons pas la définition des transformateurs de prédicats pour le langage `while`. À titre indicatif, les transformateurs de prédicats pour les instructions composées sont les suivants:

$$\mathcal{POST}(\{I_1;I_2\}, F) = \mathcal{POST}(I_2, \mathcal{POST}(I_1, F))$$

$$\mathcal{PRE}(\{I_1;I_2\}, Q) = \mathcal{PRE}(I_1, \mathcal{PRE}(I_2, Q))$$

Nous pouvons aussi prouver le triplet de Hoare $P\{I_1;I_2\}Q$ directement à l'aide du calcul de Dijkstra. Dans ce cas nous devons prouver l'une des deux formules suivantes:

$$\mathcal{POST}(\{I_1;I_2\}, F) \Rightarrow Q$$

$$P \Rightarrow \mathcal{PRE}(\{I_1;I_2\}, Q)$$

Le calcul de Dijkstra offre donc un formalisme moins déductif que la logique de Hoare pour faire des preuves de programme. Il est donc plus adapté à une implantation. Nous insistons cependant sur le fait que ce calcul ne supprime pas tous les aspects déductifs d'une preuve de programme. Par exemple, il faut toujours prouver les implications dans chacune des formules précédentes. C'est pour cette raison que le coût d'une preuve de programme ne peut pas être donné. Nous pouvons évaluer le coût des étapes purement calculatoires de la preuve, mais, étant donné que les logiques utilisées sont toujours indécidables, aucun coût ne peut être donné concernant les étapes déductives.

Correction et complétude

Le moins que nous puissions demander d'un système de preuve de programme est que toute propriété déduite dans ce système soit vraie. Cette exigence s'appelle la *correction* du système.

Nous pouvons aussi nous demander si le système est capable de démontrer tous les faits vrais. Dans ce cas, il est *complet*. L'idéal est de disposer d'un système correct et complet. Un tel système est le témoin idéal du comportement des programmes: il dit toute la vérité et rien que la vérité. La complétude a principalement un intérêt théorique, voire philosophique. Elle ignore en effet les contraintes de temps et d'espace liées à tout ordinateur. Elle permet par contre d'évaluer la puissance théorique d'un système de preuve.

À vrai dire, toute logique de Hoare est par nature incomplète. Un exemple trivial permet de se convaincre de ceci. Prenons par exemple un programme I qui ne termine pas. Le triplet de Hoare suivant est donc vrai.

$$\texttt{vrai}\{I\}\texttt{faux}$$

C'est-à-dire que partant de n'importe quel état de départ, on n'aboutit jamais à un état final. Cette formule est vraie mais non démontrable, car la terminaison du programme I n'est pas décidable. Nous pouvons voir dans cet exemple que, même en restreignant drastiquement la logique utilisée (qui se limite ici à `vrai` et `faux`) le système est incomplet. En fait, la logique de Hoare se limite souvent à l'étude des programmes qui terminent toujours. Les analyses qui font cette restriction fournissent des preuves de correction *partielle* de programmes. Pour ce qui nous concerne (c'est-à-dire : prouver qu'une mise à jour ne peut jamais violer une contrainte d'intégrité) cette restriction est sans conséquences. En effet, une mise à jour qui ne termine pas n'atteint jamais un état final, qui est le seul état qui doit satisfaire les contraintes d'intégrité. Concrètement, cette mise à jour sera tôt ou tard interrompue et tous ses effets annulés. Elle ne pourra donc violer aucune contrainte d'intégrité.

Nous avons donc vu qu'au sens habituel, aucune logique de Hoare n'est complète. Toutefois, dans l'intention d'évaluer malgré tout la puissance des logiques de Hoare, une autre notion de complétude a été définie par Cook [Coo78]. Son but est de déterminer si les modifications d'environnements réalisés par les programmes peuvent être *définies* par des formules de la logique sous-jacente. Par exemple, considérons une pré-condition P et un programme I. Nous pouvons définir un ensemble:

$$post(I,P) = \{ \llbracket I \rrbracket e \mid e \models P \}$$

C'est-à-dire l'ensemble des environnements résultant de l'exécution de I sur chaque environnement e satisfaisant P. Si il existe une formule (ou post-condition) Q qui *définit* $post(I,P)$, c'est-à-dire telle que:

$$\{ e \mid e \models Q\} = post(I,P)$$

alors on dit que la logique sous-jacente est *expressive* par rapport au langage de programmation. Toute modification faite par un programme peut être représentée *exactement* par un triplet de Hoare.

Si la logique sous-jacente d'un système de preuve est expressive par rapport au langage de programmation, la complétude de ce système découle ensuite du fait que la logique sous-jacente est complète. Par exemple, si nous pouvons totalement exprimer l'effet d'un programme avec des formules de logique du premier ordre, nous pourrons prouver toute propriété vraie car la logique du premier ordre est complète. Par contre, si nous avons besoin du langage de l'arithmétique de Peano, nous ne pourrons pas bénéficier d'un système de preuve complet. Ce problème est contourné — d'une façon plus ou moins convaincante — en supposant que la validité de toute formule peut être obtenue à l'aide d'un oracle. La complétude au sens de Cook permet en fait d'ignorer les problèmes dus à la logique sous-jacente (en particulier son incomplétude). Elle se concentre au contraire sur les problèmes dus au langage de programmation.

La propriété de complétude peut se transposer pour le calcul de Dijkstra. Elle consiste à évaluer la *puissance* des transformateurs de prédicats. Jusqu'à présent, nous nous sommes intéressés à la correction des transformateurs de prédicats. Il faut aussi s'assurer que l'information rendue par ces deux fonctions est la *meilleure possible*. Dans cette volonté d'exprimer exactement les modifications faites par un programme, Dijkstra a défini les transformateurs arrière: *weakest liberal pre-condition* ($\mathcal{WLP}$) et avant: *strongest liberal post-condition* ($\mathcal{SLP}$). Pour tout programme I, toutes formules P et Q, et tout environnement e, nous avons:

$$\begin{array}{lll} \llbracket I \rrbracket e \models \mathcal{SLP}(I,P) & \text{si et seulement si} & e \models P \\ \llbracket I \rrbracket e \models Q & \text{si et seulement si} & e \models \mathcal{WLP}(I,Q) \end{array}$$

Munis d'un de ces deux transformateurs de prédicats, nous pouvons prouver toute propriété vraie. Ils suggèrent donc une méthode de preuve complète (au sens de Cook).

3.1.2 L'Interprétation Abstraite

Si la complétude au sens de Cook peut être établie pour des langages simples, certains langages complexes sont malheureusement et irrémédiablement réfractaires à toute analyse complète. Clarke[Cla79] a montré que pour une classe particulière de langages, autorisant entre autres la transmission de fonctions en paramètres, aucune logique de Hoare ne peut être correcte et complète à la fois. En outre, nous avons vu que la complétude au sens de Cook ne nous indique pas un moyen effectif de prouver toutes les formules vraies.

Parallèlement à la logique de Hoare, une autre discipline s'est intéressée à la compréhension de la sémantique des langages de programmation: l'interprétation abstraite. L'interprétation abstraite est une technique qui vise à déterminer statiquement certains aspects du comportement des programmes. En tant qu'interprétation, elle *simule* en quelque sorte l'exécution des programmes. Mais son but, contrairement à la logique de Hoare, n'est pas de déterminer complètement les effets des programmes, mais de se focaliser sur un aspect particulier qui permettrait éventuellement de mettre en œuvre une optimisation donnée. Elle est donc volontairement abstraite.

La logique de Hoare et le calcul de Dijkstra ont été conçus pour permettre aux programmeurs de faire eux-mêmes les preuves de correction de leurs programmes (...à la main). À l'opposé, un des buts primordiaux de l'interprétation abstraite est d'aboutir à des techniques de vérification *totalement automatiques*.

Nous avons vu précédemment qu'un programme était essentiellement une fonction qui prenait en entrée un environnement (initial) et produisait un autre environnement (final). L'interprétation abstraite fait exactement la même chose, mais considère un environnement *abstrait*, représentant n'importe quel environnement initial, et produit un autre environnement abstrait, représentant n'importe quel environnement final. Nous devons évidement redéfinir la sémantique du langage de programmation pour pouvoir traiter des environnements abstraits à la place des environnements réels.

Une nouvelle notion d'environnement pour le langage `while`

Nous donnons ici un exemple complet d'interprétation abstraite. Nous nous intéressons par exemple à la parité de chaque variable d'un programme `while`. Nous pouvons, à l'aide de l'interprétation abstraite, obtenir *statiquement* une information *partielle* concernant la parité de chaque variable.

La parité d'une variable peut être inconnue, paire, impaire, ou non définie. La distinction entre une parité inconnue et non définie est la suivante: si la variable x est égale à 3 ou à 4, sa parité est inconnue. Par contre, si x est égal à $1/0$, sa parité est non définie. Une parité non définie signale en fait une erreur (ici, une division par 0).

Nous définissons naturellement un *environnement abstrait* comme une application qui indique, pour chaque variable d'un programme, sa parité. Elle assigne en fait à chaque variable une valeur de l'ensemble:

$$D = \{\bot, \texttt{pair}, \texttt{impair}, \top\}$$

où $\bot$ et $\top$ dénotent respectivement une parité non définie et une parité inconnue. Voici un exemple d'environnement abstrait:

$$e = \{x \mapsto \texttt{pair}, y \mapsto \texttt{impair}, a \mapsto \bot, b \mapsto \top\}$$

Notez qu'un environnement abstrait permet de représenter plusieurs environnements réels à la fois. Par exemple, l'environnement précédent permet de représenter, entre autres, les environnements suivants:

$$e_1 = \{x \mapsto 2, y \mapsto 3, a \mapsto \texttt{error}, b \mapsto 6\}$$

$$e_2 = \{x \mapsto 4, y \mapsto 1, a \mapsto \texttt{error}, b \mapsto 7\}$$

Par contre, il ne permet pas de représenter l'environnement suivant:

$$e_3 = \{x \mapsto 1, y \mapsto 3, a \mapsto \texttt{error}, b \mapsto 6\}$$

Car e décrit x comme étant pair. Il y a donc une relation entre un environnement abstrait et un environnement réel. Cette relation peut être vue comme une relation d'instanciation. Elle est en fait assez proche du lien entre une formule et un de ses modèles. Ce lien doit être établi rigoureusement. Cette étape est particulièrement importante car les interprétations abstraites manipulent souvent des environnements abstraits très spécifiques dont la sémantique vis-à-vis des environnements réels n'est pas claire. Nous commençons par définir la sémantique des valeurs de D par rapport aux valeurs réelles. Pour cela, nous introduisons une fonction γ, qui donne en fait l'interprétation des valeurs de D dans le domaine réel $\mathbb{N}$.

$$\gamma(\texttt{pair}) = \{0, 2, 4, ...\}$$

$$\gamma(\texttt{impair}) = \{1, 3, 5, ...\}$$

$$\gamma(\top) = \mathbb{N}$$

$$\gamma(\bot) = \emptyset$$

Nous pouvons naturellement étendre γ pour donner l'interprétation d'environnements.

$$\gamma(\{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}) = \{\ \{x_1 \mapsto w_1, ..., x_n \mapsto w_n\} \mid w_i \in \gamma(v_i)\}$$

Par exemple:

$$\gamma(\{x \mapsto \texttt{pair}\}) = \{\{x \mapsto 0\}, \{x \mapsto 2\}, \{x \mapsto 4\}, ...\}$$

Les domaines conçus pour spécifier de l'information avec plus ou moins de précision sont naturellement structurés comme des treillis. Comme d'habitude, nous noterons par $<$ la relation d'ordre sur D. La structure de treillis du domaine D est illustrée figure 3.5. Les éléments supérieurs du treillis représentent une information moins précise.

$$\begin{tikzcd} & \top & \\ \texttt{pair} \arrow[ur] & & \texttt{impair} \arrow[ul] \\ & \bot \arrow[ul] \arrow[ur] & \end{tikzcd}$$

FIG. 3.5 - *Le domaine D*

Cette structure de treillis se retrouve naturellement au niveau du domaine réel. En effet, à chaque valeur de D correspond un sous-ensemble de $\mathbb{N}$. Or, l'ensemble des parties de $\mathbb{N}$ est naturellement ordonné par inclusion. On peut étendre l'ordre sur D à l'ensemble des environnements abstraits:

$$(\{x_1 \mapsto v_1, ..., x_n \mapsto v_n\} < \{x_1 \mapsto v'_1, ..., x_n \mapsto v'_n\}) \Leftrightarrow (v_1 < v'_1 \ \wedge ... \wedge \ v_n < v'_n)$$

On a par exemple:

$$\{x_1 \mapsto \texttt{impair}\} < \{x_1 \mapsto \top\}$$

Enfin, nous pouvons définir une opération $\sqcup$ pour combiner deux environnements abstraits, qui est en fait la plus petite borne supérieure de ces deux environnements (*least upper bound* en anglais), qui existe toujours, du fait de la structure de treillis. Par exemple:

$$\{x_1 \mapsto \texttt{pair}\} \sqcup \{x_1 \mapsto \texttt{impair}\} = \{x_1 \mapsto \top\}$$

Ces développements théoriques peuvent sembler sans intérêt pour la résolution de notre problème. Ils permettent cependant de situer l'interprétation abstraite comme une approche très unificatrice. Ils facilitent ensuite la conception de méthodes de preuve de programme propres et surtout *sémantiquement fondées*.

Sémantique abstraite du langage `while`

Il faut maintenant définir intégralement une sémantique abstraite des instructions et expressions du langage `while`. Cela se fait naturellement en considérant la parité des expressions plutôt que leur valeur réelle. Pour définir rigoureusement cette sémantique définissons:

$$e[x \mapsto v] \ \equiv \ \lambda u.\texttt{if}\ u = x\ \texttt{then}\ v\ \texttt{else}\ e(u)$$

où e est un environnement abstrait, x est une variable et $v \in D$. $e[x \mapsto v]$ est identique à e sauf en x, où $e[x \mapsto v]x = v$. Nous utilisons pour la sémantique abstraite une notation similaire à la sémantique réelle. Soit I une instruction et e un environnement abstrait. La sémantique abstraite de I sur e est notée $\ll I \gg e$. Cette expression dénote un environnement abstrait. Soit *expr* une expression. La sémantique abstraite de *expr* sur e est notée $\ll expr \gg e$. Cette expression dénote une valeur du domaine D. La sémantique abstraite du langage `while` est donnée en figure 3.6. Il nous manque la sémantique abstraite des opérateurs arithmétiques.

$$\begin{array}{|l|}\hline
\ll var{:=}expr \gg e = e[var \mapsto \ll expr \gg e] \\
\ll instr_1;instr_2 \gg e = \ll instr_2 \gg \ll instr_1 \gg e \\
\ll \texttt{if}\ expr\ \texttt{then}\ instr_1\ else\ instr_2 \gg e = \ll instr_1 \gg e \sqcup \ll instr_2 \gg e \\
\ll \texttt{while}\ expr\ \texttt{do}\ instr \gg e = e\ \sqcup\ (\ll \texttt{while}\ expr\ \texttt{do}\ instr \gg \ll instr \gg e) \\
\ll var \gg e = e(var) \\
\ll const \gg e = \texttt{parité}(const) \\ \hline
\end{array}$$

FIG. 3.6 - *Sémantique abstraite des instructions de* `while`.

Celle-ci peut être définie en fonction de la sémantique abstraite de leurs sous-expressions. On donne en figure 3.7 une partie de la sémantique de l'addition. La sémantique des autres opérations peut être aisément définie en extension. Pour des raisons de place nous ne l'indiquerons pas ici.

$\ll expr_1 \gg e$	$\ll expr_2 \gg e$	$\ll expr_1 + expr_2 \gg e$
pair	pair	pair
pair	impair	impair
pair	$\top$	$\top$
pair	$\bot$	$\bot$
impair	impair	pair
...	...	...

FIG. 3.7 - *Sémantique abstraite de l'addition:* $expr_1 + \ expr_2$.

Un exemple: analyse du programme `divise`

Nous souhaitons prouver que, pour le programme `divise` donné en figure 3.4 de la page 20, si au début du programme x et y sont pairs, alors à la fin du programme b sera pair. Nous considérerons donc comme environnement abstrait initial:

$$e_0 = \{x \mapsto \texttt{pair}, y \mapsto \texttt{pair}, a \mapsto \top, b \mapsto \top\}$$

La première étape consiste à calculer l'environnement abstrait e_1:

$$\begin{aligned} \ll a := 0 \gg e_0 &= e_0[a \mapsto \ll 0 \gg e_0] \\ &= e_0[a \mapsto \texttt{pair}] \\ &= \{x \mapsto \texttt{pair}, y \mapsto \texttt{pair}, a \mapsto \texttt{pair}, b \mapsto \top\} \\ &= e_1 \end{aligned}$$

Ensuite, de la même manière:

$$\begin{aligned} \ll b := x \gg e_1 &= e_1[b \mapsto (\ll x \gg e_1)] \\ &= e_1[b \mapsto \texttt{pair}] \\ &= \{x \mapsto \texttt{pair}, y \mapsto \texttt{pair}, a \mapsto \texttt{pair}, b \mapsto \texttt{pair}\} \\ &= e_2 \end{aligned}$$

Nous arrivons à un point délicat: la boucle. L'équation donnée en figure 3.6 ne semble pas permettre de calculer la sémantique de la boucle. La solution de cette équation se calcule en fait par un mécanisme de point fixe, classique en sémantique dénotationnelle (voir par exemple [Sch86]). Grossièrement, la sémantique de la boucle s'obtient en itérant indéfiniment le corps de la boucle jusqu'à l'obtention d'un point fixe, pour peu qu'il existe. Pour ce qui nous concerne ici, ceci peut être fait en appliquant la fonction suivante:

$$\lambda e.e \sqcup \ \ll b := b - x; a := a + 1 \gg e$$

en partant de l'environnement de départ e_2, jusqu'à l'obtention d'un point fixe. Nous avons:

$$\begin{aligned} e_2 \sqcup \ \ll b := b - x; a := a + 1 \gg e_2 &= \{x \mapsto \texttt{pair}, y \mapsto \texttt{pair}, a \mapsto \top, b \mapsto \texttt{pair}\} \\ &= e_3 \end{aligned}$$

Nous itérons encore une fois:

$$\begin{aligned} e_3 \sqcup \ll b := b - x; a := a + 1 \gg e_3 &= \{x \mapsto \texttt{pair}, y \mapsto \texttt{pair}, a \mapsto \top, b \mapsto \texttt{pair}\} \\ &= e_4 \end{aligned}$$

Le point fixe est atteint. C'est l'environnement abstrait final pour le programme `while` en partant de l'environnement abstrait e_0. Nous pouvons donc conclure que si x et y sont pairs avant l'exécution de `divise`, alors b est pair à la fin de l'exécution.

Il est clair qu'ici, le treillis des valeurs abstraites étant fini, un point fixe est toujours atteint en un nombre fini d'étapes. Dans le cas de treillis infinis, des techniques d'approximation de point fixe sont utilisées: opérateurs d'élargissement et de rétrécissement. Nous ne décrirons pas ces techniques car elles ne sont pas à proprement parler utilisées dans notre travail.

Correction de l'interprétation abstraite

De même qu'en logique de Hoare, nous devions être sûrs que tous les faits déduits étaient vrais, il nous faut un critère formel pour déterminer qu'une interprétation abstraite produit une information fiable sur le comportement des programmes. Ceci nous amène à la définition suivante.

Définition 12 L'interprétation abstraite $\ll p \gg$ d'un programme p est *correcte* si et seulement si, pour tout environnement réel e et tout environnement abstrait e_a, tels que $e \in \gamma(e_a)$, on a :

$$[\![p]\!]e \in \gamma(\ll p \gg e_a)$$

La correction de l'interprétation abstraite nous dit que, pour peu que l'environnement réel initial soit une instance de l'environnement abstrait initial, alors, l'environnement réel final sera aussi une instance de l'environnement abstrait final. La réciproque n'est cependant pas vraie. C'est en ce sens que l'interprétation abstraite est une approximation de la sémantique des programmes. Elle considère des possibilités qui n'ont pas lieu d'être en réalité. Par contre, la correction nous garantit qu'elle considère au moins toutes les possibilités réelles. Dans l'impossibilité de nous fournir une information exacte sur le comportement des programmes, elle pèche par excès de prudence. L'interprétation abstraite renonce donc d'emblée à l'obtention d'une information totale, contrairement à la logique de Hoare. Par contre, elle offre toujours un mécanisme clair et aisé à implanter. Elle fournit donc un moyen totalement automatique de prouver certaines propriétés de programmes. De nombreuses références existent en interprétation abstraite. Le lecteur courageux pourra consulter le travail d'origine [CC77], ainsi que [CC76, CC79, Myc81, AH87] . Un synthèse récente est donnée par [JN95]. En ce qui concerne la logique de Hoare on pourra consulter, en plus des références déjà données, l'introduction de [GG94] et la synthèse de [Apt81].

3.2 Les applications des méthodes de preuve de programme au problème de l'intégrité en bases de données

Nous faisons dans cette section une étude comparative des différentes méthodes d'analyse de programmes qui ont été proposées pour résoudre le problème de l'intégrité. Plutôt que de faire une étude "en parallèle" de toutes les approches qui ont été proposées, nous les présenterons selon un ordre chronologique.

3.2.1 Les pionniers: Gardarin et Melkanoff

Gardarin et Melkanoff, dans [GM79], sont les premiers à suggérer une méthode pour prouver qu'une transactions ne peut pas violer une contrainte d'intégrité. Cette méthode est fondée sur la logique de Hoare.

Leur cadre de travail est le modèle relationnel. Les contraintes d'intégrité sont naturellement des formules de logique du premier ordre. Le langage de mise à jour est un langage impératif inspiré du langage ALGOL 60. En outre, la syntaxe de leur langage autorise des tests du premier ordre[2] (contrairement aux tests booléens, c'est-à-dire sans quantificateurs, qui sont utilisés habituellement), des requêtes, et des modifications de relations (insertions et retrait). Le choix d'intégrer un formalisme du premier ordre au langage de programmation n'est pas anodin et facilite en fait non seulement la preuve de programmes, mais aussi l'écriture de programmes corrects. Cet aspect de leur langage de manipulation de données a été repris dans notre approche. De prime abord, la possibilité de faire des requêtes booléennes peut sembler superflue eu égard à l'intégrité. Notons toutefois qu'à ce jour, elle est offerte par la plupart des langages de programmation pour bases de données. Nous justifierons plus précisément sa nécessité lorsque nous décrirons certains exemples de transactions.

L'approche de [GM79] repose sur l'utilisation d'un *vérificateur automatique de consistance* des transactions. Celui-ci serait lancé à la compilation et prouverait que les transactions *préservent* l'ensemble des contraintes d'intégrité. L'hypothèse, assez forte, de Gardarin et Melkanoff, est que chaque transaction *doit* être correcte. Aussi, le programmeur doit prendre les précautions appropriées pour qu'elles le soient. Un programme dont la correction n'aurait pas été prouvée ne pourrait pas être exécuté. Étant donné la complexité intrinsèque de la preuve de programme en général, ce mode de fonctionnement est quelque peu irréaliste. Comme nous le montrons plus loin, il est probable que la plupart des transactions ne puissent pas être détectées comme préservant l'intégrité, pour une simple raison de coût de la preuve. Nous pouvons toutefois espérer obtenir un gain significatif en prouvant la correction de transactions courtes. Aussi, nous pensons qu'un système tel qu'il est décrit par Gardarin et Melkanoff doit aussi permettre l'exécution de transactions dont la correction n'a pas été prouvée. Dans ce cas l'intégrité de la base devra être testée à l'exécution.

Un défaut majeur du langage de programmation utilisé dans [GM79] est l'emploi, à la place des quantificateurs existentiels, de termes de Skolem. Nous rappelons que les fonctions de Skolem sont d'habitude interprétées comme des *fonctions de choix*, qui rendent un élément d'un domaine satisfaisant une condition particulière. Ils peuvent en quelque sorte *simuler* le fonctionnement des quantificateurs existentiels. Selon Gardarin et Melkanoff, La forme de Skolem est un "format" qui facilite grandement la preuve de programme. En fait, les formes de Skolem doivent être utilisées avec beaucoup de précaution et les résultats les concernant ne s'appliquent que dans un cadre très précis. Une formule n'est pas en général équivalente à sa forme de Skolem. Par contre une formule et sa forme de Skolem sont *équisatisfaisables*, ce qui veut dire qu'une formule est satisfaisable si et seulement si sa forme de Skolem l'est aussi. Ceci explique pourquoi la Skolemisation est utilisée dans les méthodes de preuves qui procèdent par réfutation (résolution, méthode des tableaux). La forme de Skolem ne peut donc en aucun cas être prise comme un "format" commode pour

2. Les test du premier ordre sont en fait des requêtes booléennes.

écrire des formules. Loin d'être révélateur d'un certain puritanisme logique, cette remarque prend toute son importance sur un exemple particulier. Prenons par exemple une formule existentielle quelconque:$\exists x P(x)$, où x est une variable et P est un prédicat. Sa négation est: $\forall x \neg P(x)$. Sa forme de Skolem est: $P(a)$, où a est une constante de Skolem. La négation de sa forme de Skolem est: $\neg P(a)$ (Cette dernière formule, pour peu que a soit interprété comme il convient, peut être "vue" comme $\exists x \neg P(x)$). Enfin, la forme de Skolem de sa négation est: $\neg P(x)$ (qui peut être "vu" comme $\forall x \neg P(x)$). Si une formule et sa forme de Skolem étaient équivalentes on aurait obtenu que $\exists x \neg P(x)$ et $\forall x \neg P(x)$ sont équivalentes, ce qui est absurde. Cet exemple n'est pas anodin mais remet en cause la validité de l'approche de [GM79]. Reprenons par exemple la règle du `if` donnée page 19, qui est reprise textuellement par [GM79]:

$$\frac{(P \wedge F)\{I_1\}Q \quad (P \wedge \neg F)\{I_2\}Q}{P\{\texttt{if } F \texttt{ then } I_1 \texttt{ else } I_2\}Q}$$

Cette règle n'est plus valide si nous considérons que F est la forme de Skolem d'une formule de premier ordre. En effet, nous venons de voir que la forme de Skolem de la négation d'une formule n'est pas équivalente à la négation de la forme de Skolem de cette formule. Nous montrons plus loin comment traiter les quantificateurs proprement, sans avoir à changer le "format" des formules.

Hormis ces problèmes logiques, Gardarin et Melkanoff reprennent assez fidèlement les axiomes donnés par Hoare[Hoa69]. Pour prouver qu'une transaction T ne peut jamais violer une contrainte d'intégrité C, il faut prouver le triplet de Hoare:

$$C\{T\}C$$

L'approche de [GM79] n'est pas suffisamment formalisée pour suggérer une méthode automatique de preuve. Au contraire, les exemples fournis laissent penser que chaque preuve est établie de manière *ad-hoc* (c'est-à-dire, à la main). En outre, l'utilisation d'agrégats complique inutilement leur approche. En effet, les propriétés des agrégats sont souvent difficiles à prouver et nécessitent d'utiliser de l'induction.

3.2.2 La preuve de programme avec la logique temporelle

Casanova et Bernstein proposent dans [CB80] un système formel pour l'étude des langages de programmation pour bases de données. Leur cadre est assez proche de celui de [GM79], si ce n'est une plus grande attention portée aux opérateurs d'agrégation (somme, max, min). Casanova et Bernstein soulignent avec beaucoup de justesse que la complexité d'un système de preuve dépend de la richesse du langage de programmation. Ils préconisent une purification des langages de programmation afin de ne retenir que les constructions dont la simplicité et la clarté favoriserait la preuve de correction. Aussi, il est surprenant de les voir compliquer leur langage de programmation à souhait, en ajoutant nombre de caractéristiques souvent secondaires. [CB80] passent en fait beaucoup de temps à décrire leur modèle de données... et peu de temps à présenter leur système de preuve. La logique sous-jacente à leur système formel foisonne de possibilités séduisantes mais dont on peut se demander comment elles seront prises en compte par la suite. Elle comprend d'emblée l'arithmétique (de Presburger ou de Peano, au choix), les ensembles, les agrégats... Bien que [CB80] présentent leur formalisme comme étant du premier ordre, ils autorisent des fonctions et des prédicats sur des ensembles d'éléments. Ils vont même jusqu'à autoriser la définition d'ensembles par compréhension (ou intention). Pour illustrer la complexité de leur modèle, nous donnons à titre indicatif un des nombreux axiomes que [CB80] mentionnent:

$$\forall X \forall \bar{x} (\neg X(\bar{x}) \Rightarrow \mathrm{COUNT}^n(X \cup \{\bar{x}\}) = \mathrm{COUNT}^n(X) + 1)$$

ce qui veut dire en français: *pour toute relation X, et tout n-uplet $\bar{x}$ qui n'est pas encore dans X, la relation qui comprend les n-uplets de X ainsi que $\bar{x}$ a un n-uplet de plus que X*. Cet axiome semble bien être du second ordre (tous les éléments du domaine ne sont pas des ensembles). Sinon, le langage de programmation utilisé dans [CB80] est assez classique mais très complet et autorise notamment les requêtes avec agrégats.

Casanova et Bernstein décrivent ensuite un système formel censé permettre la vérification automatique de transactions. Ce système formel est encore une fois très proche de celui donné par [Hoa69], avec quelques ajouts de règles qui ressemblent à la déduction naturelle. C'est un système formel "à la Hilbert", c'est-à-dire un catalogue de déductions valides que l'on peut faire à partir d'une formule. Il est donc inutilisable sur le plan pratique.

Pour résumer, Casanova et Berstein présentent un modèle très riche mais pas véritablement de méthode pour prouver automatiquement qu'une transaction est correcte. Nous devons ici souligner qu'il ne suffit pas de mettre en évidence des liens entre la preuve de programme et la logique du premier ordre pour décrire une méthode *effective* pour faire de la vérification de programmes. Un catalogue axiomatique décrivant le modèle de données ne peut *jamais* être utilisé pour faire de la démonstration automatique. Ce phénomène a déjà été observé en démonstration automatique de théorèmes: lorsque le domaine satisfait des axiomes supplémentaires, comme ceux de l'égalité (ou bien d'autre comme les axiomes décrivant les relations d'ordre, les entiers...), il ne suffit pas d'ajouter à une formule à prouver tous les axiomes qui sont satisfaits par le domaine. Bien au contraire, tous les démonstrateurs automatiques existants (résolution et méthode des tableaux entre autres) *intègrent* sous forme de procédures particulières la sémantique de ces axiomes. Ceci permet de mettre en œuvre effectivement (et non seulement en théorie) la démonstration automatique. C'est ainsi que la résolution devient, lorsqu'on considère l'égalité, la *paramodulation*, et que des règles d'expansion de tableaux particulières sont ajoutées à la méthode des tableaux standard, pour une prise en compte efficace de l'égalité.

L'argument de l'efficacité des méthodes de preuve semble avoir été ignoré par la plupart des approches à ce jour. En fait, parler d'efficacité en ce qui concerne la preuve de programmes est une expression pudique. Il s'agit en fait de savoir si une méthode pourra être mise en œuvre, ou non. Les concepteurs de méthode de preuve de programme doivent se soucier de cet aspect là des choses dès le départ.

3.2.3 Un système expert pour la preuve de transactions

Sheard et Stemple présentent dans [SS89] une approche très originale pour prouver, formellement, automatiquement, et à la compilation, qu'une transaction ne pourra jamais violer une contrainte d'intégrité. Leur méthode est en fait inspirée des systèmes experts, et plus particulièrement de la logique de Boyer-Moore. Celle-ci est décrite en détail dans [BM79]. Il est à noter que, bien que portant le nom de logique, la logique de Boyer-Moore est située par ses inventeurs comme étant plus proche de l'intelligence artificielle que de la logique proprement dite.

Le cadre dans lequel [SS89] se placent s'appelle ADABTPL (Abstract Database Type Programming Language). La définition d'une application dans ce langage consiste en:

– Un ensemble de types de données abstraits qui sont construits en utilisant des constructeurs (tuple, liste, ensemble...) et des types de base classiques. Des contraintes peuvent être données dans la définition d'un type, afin de limiter les instances possibles de ce type.
– Le schéma de la base, construit à partir des types précédents. Des contraintes peuvent là aussi être données, afin de définir les instances acceptables de ce schéma.
– Un ensemble de transactions, c'est-à-dire de programmes qui vont modifier la base de données.

Les contraintes peuvent être classées en quatre grandes catégories, selon leur portée:

– Les contraintes de domaine, qui limitent les valeurs qu'un attribut particulier des n-uplets d'une relation peut prendre, indépendamment des autres attributs.
– Les contraintes sur chaque tuple d'une relation, qui définissent les conditions que chaque n-uplet d'une relation doit satisfaire, indépendamment des autres n-uplets.
– Les contraintes sur l'ensemble des n-uplets d'une relation.
– Les contraintes sur l'ensemble des relations de la base.

Les capacités expressives du langage de contraintes sont vastes : contraintes arithmétiques, possibilité de manipuler des ensembles, quantification universelle et existentielle, agrégats. Par contre, le langage de transactions est relativement limité. Il permet des insertions ou des retraits simples de n-uplets, ou des insertions, retraits ou modification d'un ensemble de n-uplets selon un critère.

Sheard et Stemple commencent tout d'abord par formuler des propriétés de base des constructions disponibles dans leur langage. Ces propriétés sont données sous forme d'axiomes qui sont écrits selon le formalisme original de Boyer-Moore. Celui-ci utilise une notation "à la LISP" des formules. Les théorèmes sont exprimés sous la forme d'expressions fonctionnelles.

Leur méthode semble assez complexe, mais s'intègre parfaitement bien dans le cadre de Boyer et Moore. Il faut tout d'abord axiomatiser les constructions de base utilisées dans le langage: ensembles, listes, relations... ce qui est assez difficile, étant donné la richesse des modèles de données utilisés en bases de données. Rien ne garantit que les axiomes sont suffisants pour prouver toutes les assertions vraies, c'est-à-dire qu'ils sont complets. Plus grave encore, la consistance des axiomes n'est pas établie, et ne peut pas en général l'être pour toute théorie suffisamment expressive (c'est-à-dire qui inclut la théorie des ensembles, par exemple). Ceci est très gênant, car à partir d'un ensemble d'axiomes inconsistants, on peut déduire n'importe quoi. Ce n'est pas trop grave si toute la théorie repose sur des axiomes classiques, dont on peut raisonnablement supposer qu'ils sont consistants, sans pouvoir jamais le prouver (c'est le cas, par exemple, des axiomes de Peano, ou ceux de Zermelo-Fraenkel). Par contre, le risque d'incohérence augmente considérablement au fur et à mesure qu'on incorpore des axiomes plus spécifiques pour formaliser les modèles de données complexes. Le concepteur de la base devra donc faire particulièrement attention aux axiomes qui sont introduits. L'idéal serait d'établir des preuves de consistance *relative* des axiomes, vis-à-vis d'un nombre très restreint d'axiomes supposés consistants. Cette tâche est cependant très ardue. Selon Sheard et Stemple, la fiabilité de leur méthode tient à l'espérance qu'aucune propriété "surprenante" ne pourra être déduite. C'est une façon pudique de dire que leur approche n'est pas formellement fondée (ou du moins qu'elle n'est fondée que sur la justesse des axiomes de départ, qui à son tour, ne peut être établie). Une propriété "surprenante" serait par exemple une contradiction, qui pourrait à son tour servir à prouver n'importe quelle propriété.

À partir de ces axiomes, des théorèmes sont déduits et stockés dans une base. La construction d'une preuve consiste à "dérouler" l'expression fonctionnelle de chaque terme dont la preuve est constituée. La puissance du démonstrateur est augmentée par l'utilisation de *méta-lemmes*, qui expriment des propriétés générales sur des fonctions et des relations, indépendamment de leur nature. A partir d'un méta-lemme, on peut générer quantités de lemmes en *instanciant* chaque paramètre par des fonctions ou relations particulières.

Par rapport à un démonstrateur automatique de théorèmes en logique du premier ordre, le fonctionnement de leur démonstrateur de théorèmes semble assez obscur. Il génère des théorèmes et les stocke dans une base jusqu'à ce qu'"assez" de théorèmes aient été prouvés sur un domaine particulier. Ceci est fait afin d'éviter l'explosion de la base de théorèmes. La définition de la quantité "suffisante" est floue. Elle ne permettra pas nécessairement de prouver le théorème final. La quantité de théorèmes générée, même pour une petite application jouet, est assez importante, ce qui ne permet pas de suivre véritablement les étapes d'obtention d'une preuve (les preuves fournies par des démonstrateurs automatiques sont toujours difficiles à suivre). La théorie sur les ensembles et les n-uplets comporte 55 définitions de fonctions et 209 lemmes, l'exemple d'application entraîne l'ajout de 55 nouvelles fonctions et de 139 nouveaux lemmes.

En conclusion, nous insistons sur le fait que cette approche est la seule à avoir été implantée jusqu'à présent, et qu'elle fournit des résultats positifs. C'est en fait un travail considérable et surtout complet. Il a nécessité la maîtrise d'un outil théorique complexe, peu employé dans le domaine des bases de données. Surtout, Sheard et Stemple ont évité la situation d'apesanteur dans laquelle la plupart des autres travaux se retrouvent. Alors que chez [GM79, CB80, Qia90], "tout est possible" (mais rien n'est implanté, ni validé par l'expérimentation !), [SS89] délimitent bien le champ d'application de leur méthode. La vérification automatique est pour la première fois décrite comme un processus très difficile et complexe. Le langage de transactions est d'emblée assez limité et le modèle de données est simple. C'est justement cela qui permet à leur méthode d'être opérationnelle.

3.2.4 Un système d'axiomes pour les transactions

Xiaolei Qian propose dans [Qia90] un système formel fondé sur la logique de Hoare à des fins de preuve, d'optimisation et de synthèse de programmes. Ce travail présente en quelque sorte une version "propre" des systèmes de [GM79, CB80].

Tout d'abord, son modèle de données et son langage de transactions sont totalement dépourvus de fantaisie, ce qui est louable. Les constructions utilisées sont simples et classiques. Ceci permet à [Qia90] de décrire en peu de pages (ce qui n'est pas négligeable) un système de preuve élégant. Qian, à l'instar de [GM79, CB80], dit qu'un bon équilibre doit exister entre la puissance d'expression du langage de transactions, et la possibilité de faire de la preuve automatique. Le fait nouveau est qu'elle met en application ses conseils. Qian ne mentionne pas l'utilisation d'agrégats, ou même de l'arithmétique, ce qui, par rapport à [GM79, CB80] (et contrairement à ce que l'on pourrait penser), est un progrès. Le langage proposé par Qian est donc classique: insertion-effacement-modification d'un n-uplet dans une relation, instruction conditionnelle, boucle base de données (itération sur tous les éléments d'un ensemble). Ce langage est simple, mais pose déjà des problèmes délicats. En effet, la boucle base de données n'est pas déterministe. Elle nécessite un choix d'ordre entre les éléments d'un ensemble qui, par définition, n'est pas ordonné. Ce choix peut radicalement changer le résultat des calculs. Qian spécifie un ensemble de conditions afin de définir les transactions qui sont "bien formées". Il n'est pas surprenant, après ce que l'on vient de voir, que chacune de ces conditions concerne l'utilisation des boucles. Parmi ces conditions, la plus importante semble être qu'à l'intérieur d'une boucle, on ne peut pas accéder à une relation en lecture et en écriture. Cette restriction semble suggérer que les résultats d'une itération ne peuvent pas être utilisés dans une autre itération. En effet, Qian considère que le corps d'une boucle est exécuté en parallèle sur tous les éléments d'un ensemble.

Comme [GM79], Qian utilise des triplets de Hoare. Elle établit la complétude de son système en montrant que toute modification relationnelle peut être définie en logique du premier ordre (c'est-à-dire, la logique du premier ordre est *expressive* par rapport au langage de programmation). La complétude découle du fait que la logique du premier ordre est complète.

[Qia90] se concentre en fait plutôt sur des aspects théoriques de la preuve de programme, et élude les problèmes de mise en œuvre de sa méthode. Ceci est regrettable, d'autant plus qu'encore une fois, la preuve de complétude n'est pas très convaincante et ressemble plus à une preuve de correction. En fait, la preuve de complétude serait, nous le pensons, très difficile à établir. La plupart des instructions sont sans problèmes, et sont d'ailleurs reprises de [Hoa69]. La seule instruction qui pose véritablement problème est la boucle "bases de données", dont la sémantique est, dans le meilleur des cas, très complexe, et dans le pire des cas, pas claire. Elle repose en effet sur un modèle d'exécution parallèle afin d'éviter les effets de bord et le non déterminisme. Trouver "la" bonne formule pour définir une boucle "bases de données" est vraiment hasardeux. De nombreuses hypothèses informelles sont postulées sur la construction des programmes, sans aucune contrepartie formelle au niveau du système d'axiomes. Il eut en fait mieux valu définir une sémantique partielle (mais correcte) de la boucle "bases de données", plutôt que de chercher à la décrire précisément, ce qui complique inutilement l'approche de [Qia90].

Le travail de [Qia90] n'a pas abouti à une implantation. De son propre aveu, Qian[Qia93] renonce à utiliser son système d'axiomes et prend la direction opposée qu'est la *synthèse de transactions*. Pour Qian, les méthodes d'analyse de programmes disponibles ne permettent pas de faire de la vérification automatique de transactions: "The generation of unnecessary constraint checking and enforcement code should be avoided. It should be noticed that such optimization is out of reach for conventional program optimization techniques, because conventional program optimizers do not have the ability to reason about integrity constraint." De "tout est possible" dans [Qia90], nous avons abouti à "rien n'est possible" dans [Qia93]. La vérité se situe en fait entre ces deux extrêmes.

3.2.5 La vérification optimisée des contraintes par les pré-conditions

Lawley, dans [Law95], propose une méthode originale pour vérifier efficacement un ensemble de contraintes. Son but principal n'est donc pas de faire de la preuve de programmes. Cependant, il utilise d'une façon inédite le calcul Dijkstra, et c'est à ce titre que nous le mentionnons .

Le principe de sa méthode est de vérifier, avant l'exécution d'une transaction, une pré-condition, qui est une condition suffisante pour que cette transaction produise une base de données cohérente. Cette pré-condition est produite par un transformateur de prédicats arrière (*weakest liberal pre-condition*). L'avantage de cette méthode est qu'elle permet de vérifier *a priori* qu'une exécution particulière d'une transaction sera cohérente. Cela permet d'éviter les reprises arrière coûteuses. Nous soulignons que la pré-condition est vérifiée à l'*exécution*, sur la base de données en entrée de la transaction, et non pas à la compilation.

Le cadre utilisé par [Law95], GULOG, est déductif et orienté-objets. Une base de données est décrite à l'aide de prédicats, intentionnels et extensionnels. La plupart des concepts orientés objets: classes, héritage,...sont en fait traduits en termes logiques. Le langage de programmation utilisé diffère de tous ceux qui sont considérés dans [GM79, CB80, SS89, Qia90]. Il prend la forme de formules de logiques du premier ordre. Par exemple, l'insertion de n-uplets dans la base se fait comme suit:

$$\forall \bar{x}((\Gamma \vdash q(\bar{x}), P) \rightarrow +p(\bar{x}))$$

où $\bar{x}$ est un n-uplet variable, $q(\bar{x})$ est une requête, P est un programme Datalog et p est un symbole de relation. Le sens de cette instruction est: *pour tout n-uplet $\bar{x}$ satisfaisant $q(\bar{x})$, insérer $\bar{x}$ dans la relation p*. On considère en outre le programme Datalog P pour évaluer $q(\bar{x})$, et Γ indique le type des objets de l'instruction. Les autres instructions sont similaires. La création d'objet est la seule exception:

$$\forall \bar{x}\mathcal{I}y((\Gamma \vdash q(\bar{x}), P) \rightarrow +y : c$$

La signification des symboles est identique à l'instruction précédente. Il y a en outre une variable libre y et un nom de classe c. Le sens de cette instruction est: *pour chaque n-uplet $\bar{x}$ qui satisfait $q(\bar{x})$, créer un objet (dénoté par y), dans la classe c*. Il n'y a pas de lien particulier entre $\bar{x}$ et y. La requête $q(\bar{x})$ sert en fait à fixer le nombre d'objets de la classe c qui devront être créés. Un autre point intéressant est que ce langage permet de mettre à jour des relations intentionnelles, ce qui revient à faire de la mise à jour de vues. Par contre, il n'offre pas d'instruction conditionnelle "`if`".

Lawley décrit ensuite un transformateur de prédicats pour chaque instruction. À l'instar de [GM79], [Law95] utilise des fonctions de Skolem, ce qui est très regrettable. En plus des raisons qui ont déjà été avancées contre l'utilisation abusive de telles fonctions, nous pouvons ajouter que ces fonctions rendent impossible une véritable preuve de correction et de complétude de la méthode de Lawley. En effet, la validité des formules décrites par Lawley dépend de l'interprétation particulière donnée aux fonctions de Skolem. Or, une preuve, par nature syntaxique, ne peut faire d'hypothèses particulières sur l'interprétation des fonctions de Skolem, sauf en formalisant ces hypothèses par des axiomes particuliers, ce qui n'est pas fait dans [Law95]. Une fois encore, les preuves de complétude ne sont pas très convaincantes. De même, la méthode n'est pas mise en œuvre. Certains problèmes, et non des moindres, sont ignorés. La pertinence de la méthode, qui était censée aboutir à une vérification efficace, n'est pas établie. En effet, rien n'indique qu'il vaut mieux, en termes de coût, vérifier la pré-condition à priori, que la contrainte a posteriori. Supposons que la pré-condition soit de la forme:

$$F \wedge C$$

où C est la contrainte d'intégrité et F une formule de logique. Cette pré-condition peut être réduite à F, car on suppose que C est satisfaite avant l'exécution de la transaction. Dans ce cas, l'évaluation a priori de la pré-condition risque d'être effectivement plus efficace que la vérification a posteriori de la contrainte. Cependant, les techniques pour isoler la contrainte C dans la pré-condition, c'est-à-dire obtenir une pré-condition de la forme $F \wedge C$, ne sont pas évoquées. L'évaluation de la pré-condition risque d'être plus coûteuse que la vérification a posteriori de la contrainte.

3.2.6 Une méthode fondée sur le typage

Un autre approche a été développée dans [BD95, Pol96], dans le cadre du projet Thémis. Elle est moins ambitieuse que les méthodes proposées dans [GM79, CB80, SS89, Qia90], mais est beaucoup plus opérationnelle. Cette méthode se propose de détecter statiquement que certaines transactions ne peuvent pas violer certaines contraintes. L'originalité de cette méthode est qu'elle ignore le détail précis des modifications qui sont faites sur la base, mais se fonde exclusivement sur l'information de typage des objets modifiés.

Le cadre dans lequel [BD95, Pol96] se placent est celui d'un langage de programmation pour bases de données proche d'O_2. Outre le modèle de données orienté objet, et le langage de mise à jour impératif (proche de O_2C), ils intègrent un langage d'expression de contraintes semi-déclaratif fondé sur la logique du premier-ordre. Une caractéristique importante de ce langage est qu'il autorise l'utilisation de méthodes au sein des contraintes. Cette idée peut sembler intéressante dans un cadre orienté objet mais pose certains problèmes. En effet, les méthodes ne sont pas décrites de façon déclarative, mais avec un langage impératif. Le programmeur a donc la possibilité de concentrer la sémantique d'une contrainte dans la définition des méthodes, plutôt que de l'exprimer de façon propre et déclarative avec la logique du premier ordre. Ce problème gêne la bonne compréhension des contraintes. En outre, le programmeur peut écrire des contraintes qui ne terminent pas, ou qui font des erreurs d'exécution. Or, la logique du premier ordre n'est définie que sur des fonctions *totales*. Concrètement, une contrainte ne pourra être évaluée que si tous les appels de méthodes terminent et sont dépourvus d'erreurs d'exécution, ce qui est indécidable. Le langage de contraintes proposé dans [BD95, Pol96] peut être en fait totalement détourné. Il peut en effet servir à spécifier la sémantique des traitements (c'est-à-dire des méthodes), autant que pour imposer des contraintes sur les données. Nous reprenons un exemple de [Pol96]:

$$\forall r \in \texttt{Les\_Reseaux\_Routiers},\ r \rightarrow \texttt{nb\_troncons}() \geq 1$$

`Les_Reseaux_Routiers` est une racine de persistance structurée comme un ensemble. r dénote n'importe quel objet appartenant à `Les_Reseaux_Routiers`. La méthode `nb_troncons` est appliquée à l'objet r. Cette contrainte ne concerne pas véritablement les données mais plutôt la définition de la méthode `nb_troncons`. Elle pourrait même être vérifiée une bonne fois pour toutes, à la compilation, en appliquant à `nb_troncons` des techniques de preuve de programme. La sémantique des contraintes est parfois loin d'être claire, tant elle est masquée par l'utilisation des méthodes:

$$\forall r \in \texttt{Les\_Reseaux\_Routiers},\ r \rightarrow \texttt{matrice}() \rightarrow \texttt{fermeture}() \rightarrow \texttt{connexe}()$$

Force est de constater qu'une connaissance profonde de l'application est nécessaire pour comprendre cette contrainte. Toute la sémantique de cette contrainte est en fait cachée par les différentes méthodes auxquelles elle fait appel. Ceci empêche aussi de contrôler la consistance des contraintes et d'éviter les redondances.

Comme nous l'avons indiqué, la méthode de [BD95, Pol96] est fondée sur le typage et part d'un principe simple: une transaction qui met à jour des objets d'un type τ ne pourra pas violer une contrainte que si celle-ci fait référence à des objets du type τ. Leur approche procède par analyse syntaxique des transactions et des contraintes pour déterminer les types des objets mis à jour et le type des objets référencés par les contraintes. Cette technique est affinée dans de nombreuses directions:

– Pour les objets structurés comme des n-uplets, on peut s'intéresser, en plus de la simple information de typage, aux attributs qui sont mis à jour ou référencés.

– Le sous-typage et l'héritage sont pris en compte.

– Les méthodes référencées dans les contraintes sont analysées d'une façon détaillée.

Cette méthode est volontairement abstraite car elle ignore la façon dont les objets sont modifiés. Elle produit cependant des résultats positifs et est très peu coûteuse à mettre en œuvre. À ce titre, elle doit être considérée comme une phase préliminaire de toute autre méthode plus détaillée. L'approche de [BD95, Pol96] a été implantée comme un pré-processeur de O_2, nommé O_2-integrity.

3.3 Conclusion

Nous avons présenté dans ce chapitre trois techniques pour faire de la preuve de programme: la logique de Hoare, le calcul de Dijkstra, et l'interprétation abstraite. La logique de Hoare offre un système déductif pour prouver des propriétés de programmes, alors que le calcul de Dijkstra, fondé sur la notion de transformateur de prédicats, permet d'obtenir la sémantique logique des programmes de façon plus directe, en générant des pré ou post-conditions. Le cadre de l'interprétation abstraite permet de prouver certaines propriétés de programmes, sans avoir à capturer la totalité de la sémantique de ces programmes.

Nous avons ensuite présenté les différentes approches disponibles à ce jour pour prouver qu'une transaction préserve une contrainte d'intégrité. Ces approches, hormis celle de [SS89], n'ont pas permis d'aboutir à un système de gestion de contraintes d'intégrité efficace. Certains travaux ne sont pas suffisamment formalisés ([GM79, CB80]). D'autres considèrent un modèle de données et un langage trop complexes qui les empêchent de développer une théorie claire et implantable ([GM79, CB80, Law95]). En général, les auteurs se focalisent sur les aspects théoriques de leur approche, en particulier la complétude. Les aspects plus pratiques sont ignorés. Le but de cette thèse sera donc d'apporter une solution essentiellement *pratique* au problème de la preuve de transactions, tout en établissant notre méthode sur des techniques formelles propres, comme celles qui ont été proposées dans la première partie de ce chapitre.

Le problème que nous traitons dans cette thèse n'est qu'un des nombreux problèmes posés par l'intégrité. De nombreux travaux concernent en particulier la vérification optimisée des contraintes à l'exécution :[Nic79, LT85, HI85, BM86, KSS87, HCN84, WSK83, BDM88, BD95]. Il est clair que lorsque nous ne pourrons pas prouver qu'une transaction préserve une contrainte, nous devrons vérifier cette contrainte systématiquement, après chaque exécution de cette transaction. Notre système de gestion de contraintes d'intégrité devra bien évidement intégrer les techniques d'optimisation développées dans ce domaine.

Avant d'exposer notre technique de preuve de programme, nous présentons dans le prochain chapitre le cadre d'application de notre approche. Il s'agit d'un modèle de bases de données orienté objet simplifié, muni d'un langage de programmation impératif, et offrant la possibilité d'exprimer des contraintes d'intégrité.

Chapitre 4

Cadre de travail

Nous présentons dans ce chapitre le contexte auquel nous appliquons les techniques de preuve automatique de programme. Comme nous l'avons vu dans le chapitre précédent, l'efficacité de ces méthodes dépend de la complexité du modèle de données et du langage de programmation utilisés. C'est donc volontairement que nous nous plaçons dans un cadre simple et sans fioritures. Ce parti pris permet aussi de faire une présentation claire et concise de notre approche. Le système de gestion de bases de données que nous considérons a les caractéristiques suivantes :

- Le modèle de données est orienté objet. Il est inspiré du modèle de données de O_2. À ce titre, il intègre les notions d'objet, de méthode, de type, de classe et d'héritage.
- Il permet l'expression de *contraintes d'intégrité* de façon déclarative.
- Le langage de programmation, proche du langage O_2C, est de type impératif et permet d'exprimer des *méthodes*. Ces méthodes peuvent être surchargées.

Le but n'est pas de présenter en toute rigueur un nouveau modèle de données, mais de rappeler un certain nombre de caractéristiques essentielles des modèles de données déjà existants. Notre approche est avant tout liée au style de programmation utilisé : le style impératif. Cependant, elle est particulièrement adaptée aux contexte des bases de données orientées objet dans lequel les traitements opérés (les méthodes) sont de taille réduite et très modulaires.

Dans le prochain chapitre, nous décrivons une technique d'analyse statique des méthodes afin de prouver que celles-ci préservent des contraintes d'intégrité. Avant toutes choses, nous donnons à la fin de ce chapitre une technique de décomposition des contraintes et des méthodes qui permet de simplifier cette analyse statique.

4.1 Le modèle de données

Le modèle de données que nous présentons dans cette section est une version simplifiée du modèle de données orienté-objet, tel qu'il est décrit par exemple dans [AHV95]. Ce modèle intègre les concepts des langages orientés objet tels que les notions d'objet, de méthode, de type, de classe, d'héritage, de surcharge et de liaison tardive. Nous nous contenterons de faire des rappels sur ces différentes notions.

Dans notre modèle, les données sont stockées dans des *objets*. Intuitivement, un objet associe des données avec un ensemble d'opérations qui peuvent utiliser ou modifier ces données : les méthodes. Un objet est identifié par un *identificateur*, et a un certain nombre d'*attributs*, qui sont soit des références à d'autres objets, soit des valeurs atomiques (entiers...). Les objets sont *typés*. Un type permet de décrire la structure des données associées à un objet. Une classe permet de décrire un certain nombre de caractéristiques communes à un ensemble d'objets, en particulier leur type, et les méthodes que l'on peut leur appliquer. Les données d'un objet ne peuvent être manipulées que par les méthodes associées à cet objet. On parle alors d'*encapsulation*. L'application du traitement spécifié par une méthode aux données d'un objet s'appelle l'"envoi de message". L'objet concerné s'appelle le destinataire de l'envoi.

L'*héritage* est un mécanisme qui permet de décrire une classe à partir d'une autre classe déjà existante. La première classe s'appelle la *sous-classe* et la seconde la *super-classe*. Les objets de la sous-classe "héritent" de tous les attributs et des méthodes définis dans la super-classe. On peut en outre définir des attributs et des méthodes nouvelles dans cette nouvelle classe. Il est aussi possible de *redéfinir* une méthode dans une sous-classe. À un nom de méthode correspond alors plusieurs implantations. On dit alors que la méthode est *surchargée*. Le choix du code à exécuter dépend alors de la classe d'appartenance de l'objet destinataire de l'appel de méthode. Ce choix est déterminé à l'exécution. On parle alors de *résolution tardive*.

Un point essentiel des bases de données est la *persistance* des données. Le stockage des objets se fait par le biais de *racines de persistance*, qui sont en fait les points d'entrée dans la base de données. Dans notre contexte, une racine de persistance est une variable du système, stockée en permanence, faisant référence à un ensemble d'objets. La persistance se fait par atteignabilité : la base de données (c'est-à-dire, l'ensemble des objets stocké à la fin d'un programme) est définie comme l'ensemble des objets auxquels on peut accéder à partir des racines de persistance. Les autres objets sont détruits avec un mécanisme de "ramasse-miettes".

Une contrainte d'intégrité est une condition, construite dans un langage proche de la logique du premier-ordre, qui doit être satisfaite par les objets de la base de données.

4.1.1 Types et sous-typage

Dans notre modèle, les objets sont structurés comme des n-uplets en première forme normale (chaque attribut est indécomposable), ou comme des ensembles dont les éléments sont indécomposables. Les racines de persistance sont structurées comme des ensembles. Les notions de type et de classe sont interdépendantes : les types sont définis à partir de noms de classes et les classes à partir de types. Commençons par définir les types. Nous supposerons l'existence d'un ensemble de *noms de classes* (qui seront décrites plus loin).

Définition 13 Les types sont définis comme suit :

- `integer`, `string`, `boolean` sont des types atomiques.
- un nom de classe est un type atomique.
- Soit τ un type atomique. $\{\tau\}$ est un type ensemble.
- Soient a_1,...,a_n des attributs distincts, et τ_1,...,τ_n des types atomiques. $[a_1 : \tau_1, ..., a_n : \tau_n]$ est un type n-uplet.

Par exemple, si `Personne` et `Enfants` sont des noms de classe, alors :

```
Type-Personne=[epoux:Personne,ami:Personne,argent:integer]
Type-Parent=[epoux:Personne,ami:Personne,argent:integer,enfants:Enfants]
Type-Employe=[epoux:Personne,ami:Personne,argent:integer,patron:Personne]
```

sont tous trois des types n-uplet.

On peut définir une relation d'ordre partiel en types (notée $\leq$) appelée *relation de sous-typage*. Le sous-typage est structurel et inféré selon les règles de Cardelli [Car84].

Définition 14 La relation de sous-typage $\leq$ est définie comme suit :

– $\leq$ est réflexive.

– Soient a_1,...,a_m des attributs, et τ_1,...,τ_m, τ'_1,...,τ'_n des types, tels que $n \leq m$ et $\forall i \in [1, n], \tau_i \leq \tau'_i$. Alors :

$$[a_1 : \tau_1, ..., a_m : \tau_m] \leq [a_1 : \tau'_1, ..., a_n : \tau'_n]$$

– Soient deux types τ et τ' tels que $\tau \leq \tau'$. Alors $\{\tau\} \leq \{\tau'\}$.

Prenons par exemple les trois types définis précédemment. Nous avons :

$$\texttt{Type-Parent} \leq \texttt{Type-Personne}$$
$$\texttt{Type-Employe} \leq \texttt{Type-Personne}$$

4.1.2 Classes

Une classe définit les caractéristiques communes à un ensemble d'objets : structure de données (c'est-à-dire leur type) et ensemble de méthodes. En outre, à l'instar de la relation de sous-typage, on peut définir une relation dite d'*héritage* entre classes. Lorsqu'une classe κ hérite d'une classe κ', elle hérite de tous ses attributs et de toutes ses méthodes. Elle peut en outre définir d'autre attributs, d'autres méthodes, ou redéfinir les méthodes déjà définies en classe κ' (méthodes surchargées).

Définition 15 Une classe est définie par un nom, un type, et un ensemble de méthodes. Soit κ et κ' des noms de classe, τ un type, et m_1,...,m_n des noms de méthodes. Une classe permet de définir un ensemble d'objets qui ont les mêmes caractéristiques. La déclaration de κ se fait comme suit :

`class` $\kappa : \tau$ `inherits` κ'
signature de m_1*;*
...
signature de m_n*;*
`end;`

où "*signature de* m_i" indique le nom de la méthode m_i ainsi que le type de ses paramètres.

Lorsqu'une classe hérite d'une autre, alors le type de la sous-classe est un sous-type de la super-classe. Nous donnons en Figure 4.1 un exemple de schéma censé modéliser des personnes qui peuvent être des parents ou des employés. Les objets de la classe `Personne` ont un époux, un meilleur ami, et une somme d'argent, respectivement représentés par les attributs `epoux`, `ami` et `argent`. On peut appliquer à ces objets les méthodes `separe`, `marie`, `debit` et `meilleur-ami`. Nous décrirons plus tard l'implantation de ces méthodes.

Les objets de la classe `Parent` ont les mêmes attributs que les objets de la classe `Personne`, avec en outre un ensemble d'enfants (attribut `enfants`). Les méthodes `separe`, `marie` et `debit` sont redéfinies dans la classe `Parent`. Ces trois méthodes sont donc surchargées.

Les objets de la classe `Employe` ont les mêmes attributs que les objets de la classe `Personne`, avec en outre un patron (attribut `patron`). La méthode `separe` est redéfinie dans cette classe.

```
class Personne : [ epoux:Personne, ami:Personne, argent:integer]
method separe();
method marie(q:Personne);
method debit(somme:integer);
method meilleur-ami(q:Personne);
end;

class Parent : [ enfants:Enfants ]
inherits Personne
method separe();
method marie(q:Personne);
method debit(somme:integer);
end;

class Employe : [ patron: Personne ]
inherits Personne
method separe();
end;

class Enfants : { Personne }
end;
```

FIG. 4.1 - *Le schéma de la base des personnes.*

4.1.3 Objets et persistance

Les classes ne décrivent que des structures de données vides, qui constituent le schéma de la base de données. Pour construire une base de données particulière, il nous faut maintenant "instancier" ces classes, c'est-à-dire définir un ensemble d'objets dont les données sont structurées conformément à leur classe d'appartenance, et auxquels on peut appliquer les méthodes définies dans cette même classe. Les objets sont référencés par un *identificateur d'objet*. Nous supposerons donc l'existence d'un ensemble $\mathcal{O} = \{\mathtt{o1}, \mathtt{o2}, ...\}$ d'identificateurs d'objets.

Définition 16 L'ensemble $\mathbf{instances}(\tau)$ des instances d'un type τ est défini comme suit :

- Si $\tau = \mathbf{integer}$ alors $\mathbf{instances}(\tau) = \mathbb{N}$
- Si $\tau = c$, où c est un nom de classe, alors $\mathbf{instances}(\tau)$ dénote l'ensemble des identificateurs d'objets de c.
- Si $\tau = \{\tau'\}$, alors $\mathbf{instances}(\tau)$ dénote tous les ensembles d'éléments de $\mathbf{instances}(\tau')$ (formellement : $2^{\mathbf{instances}(\tau')}$).
- Si $\tau = [a_1 : \tau_1, ..., a_n : \tau_n]$, alors $\mathbf{instances}(\tau)$ dénote tous les n-uplets $[a_1 : v_1, ..., a_n : v_n]$ où $v_i \in \mathbf{instance}(\tau_i)$, pour tout i entre 1 et n.

En outre, une valeur spéciale `nil` permet de représenter une valeur non définie, ou inexistante. Nous pouvons maintenant définir précisément ce qu'est un objet.

Définition 17 Soit c une classe de type τ. Un objet de la classe c est un couple *(identificateur,valeur)*, où *identificateur* est un identificateur d'objet appartenant à $\mathcal{O}$, et *valeur* est une instance du type τ (ou τ est le type associé à la classe c). Pour un identificateur d'objet donné, il ne peut y avoir qu'une seule valeur associée.

Définition 18 Une racine de persistance est une variable du système, de type ensemble. Sa valeur est stockée en permanence sur disque, ainsi que tous les objets qu'elle référence directement ou indirectement (persistance par atteignabilité). Elle se déclare comme suit :

$$\mathtt{root}\ r : \{\tau\}$$

La base de données est définie comme l'ensemble des objets auxquels on peut accéder à partir des racines de persistance.

Nous pouvons par exemple déclarer trois racines de persistance, afin de stocker respectivement des objets des classes `Personne`, `Parent` et `Employe`.

```
root LesPersonnes : { Personne };
root LesParents : { Parent };
root LesEmployes : { Employe };
```

Remarques Ce modèle "expurgé" oublie de nombreuses notions : valeurs structurées, listes, multi-ensembles ...etc. Ces oublis sont volontaires et vont nous permettre de présenter notre approche "sans parasites". Nous traiterons ces notions ultérieurement.

Le modèle orienté objet habituel autorise des types plus riches que les types que nous avons présentés précédemment. Ces types évolués permettent de décrire des données structurées complexes. Par exemple, on peut construire un type : `[a:integer,b:{integer}]` qui décrit un n-uplet dont le second attribut est structuré comme un ensemble d'entiers. Dans notre modèle, et pour des raisons de simplicité, ceci n'est pas autorisé. Nous pouvons toutefois "simuler" ce type de valeurs en décomposant leur structure. On doit tout d'abord définir une classe intermédiaire :

```
classe Entiers: { integer }
```

Le type `[a:integer,b:{integer}]` se traduit dans notre modèle par : `[a:integer,b:Entiers]`.

Notre approche peut prendre en compte ces valeurs structurées, mais pour simplifier sa présentation, nous nous limiterons à des valeurs simples. Les valeurs complexes satisfont des conditions (elles sont non partageables et non modifiables), qu'il serait inopportun de considérer en premier lieu.

4.1.4 Contraintes d'intégrité

Une contrainte d'intégrité est une condition qu'une base de données doit respecter. Cette condition est construite dans un langage très proche de la logique de premier-ordre. Ceci est particulièrement intéressant à de nombreux égards :

– la logique du premier ordre est un langage expressif,

– sa sémantique est bien définie,

– de nombreux résultats théoriques et pratiques la concernant peuvent être exploités.

Les définitions suivantes ne sont qu'une adaptation à notre cadre des définition habituelles en logique.

Définition 19 On considère les ensembles de symboles suivants :

– les constantes : 0, 1,..., `nil`.

– les variables : x, y, z...

– les attributs (ou fonctions unaires) : `epoux`, `ami`, `argent`...

– les prédicats : $=$, $\neq$, $\in$, $>$,...

– les connecteurs logiques : $\wedge$, $\vee$, $\neg$...

– les quantificateurs : $\forall$, $\exists$.

Les attributs sont considérés comme des symboles de fonctions unaires. L'union de ces ensembles constitue un *langage*. Nous conviendrons que les termes et formules définies dans la suite le sont relativement à ce langage, sans avoir à le préciser. Traditionnellement, l'application d'une fonction, par exemple `epoux`, à un élément, par exemple `o1`, est notée `epoux(o1)`. Nous emploierons la notation "pointée" : `o1.epoux`, plus utilisée dans le domaine de la programmation.

Définition 20 Les termes des contraintes d'intégrité sont définis comme suit :

– Les constantes et les variables sont des termes.

– Si t est un terme, et a un attribut, alors $t.a$ est un terme.

Nous masquons le fait que les termes sont *typés*, afin de ne pas alourdir les notations.

Définition 21 L'ensemble des formules est défini comme suit :

– Si t_1 et t_2 sont des termes, alors $t_1 = t_2$, $t_1 \neq t_2$, $t_1 \in t_2$,... sont des formules (atomiques).

– Si F_1 et F_2 sont des formules, alors $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$,... sont des formules.

– Si x est une variable, t un terme, et F une formule, alors $(\forall x \in t : F)$, $(\exists x \in t : F)$ et $(\exists x \in t)$ sont des formules.

Là aussi, nous masquons le fait que les contraintes sont (bien) typées. Munis de toutes ces définitions, nous aboutissons à :

Définition 22 Une contrainte est une formule close (i.e. sans variables libres).

Exemple 2 Nous pouvons définir un certain nombre de contraintes sur le schéma de la figure 4.1 :

1. Au moins une personne est célibataire.
2. Une personne ne peut pas être mariée à elle-même.
3. Une personne ne peut être mariée à un de ses enfants.
4. Une personne qui n'est pas mariée doit avoir un patron.
5. Une personne doit avoir de l'argent.
6. Une personne qui n'est pas mariée ne peut pas avoir d'enfants (!).
7. Au moins une personne n'a pas de meilleur ami.

Ces conditions sont traduites en contraintes d'intégrité données figure 4.2

La sémantique des contraintes d'intégrité est en tout point identique à celle des formules de logique du premier ordre. Qu'elle soit relationnelle, orientée-objet ou déductive, une base de données définit toujours une interprétation d'un langage du premier ordre.

– Ce langage est défini d'une part par les constantes, fonctions et relations standard que nous avons vues : 0, 1,... ,= ,... mais aussi par le schéma de la base de données, qui permet de définir les attributs : `epoux`, `argent`,...

– Le domaine d'interprétation est l'union disjointe de l'ensemble des entiers (`integer`), des chaînes de caractères `string`,..., et de l'ensemble $\mathcal{O}$ de tous les identifiants d'objets.

– Les éléments standard du langage sont interprétés de manière habituelle. Une base de données ne diffère d'une autre que par son interprétation des attributs, et du symbole $\in$.

C1: $\exists\ x \in$ LesPersonnes: x.epoux=nil;

C2: $\forall\ x \in$ LesPersonnes: x.epoux$\neq x$;

C3: $\forall\ x \in$ LesPersonnes: x.epoux $\notin x$.enfants;

C4: $\forall\ x \in$ LesPersonnes: x.epoux=nil $\Rightarrow x$.patron$\neq$nil;

C5: $\forall\ x \in$ LesPersonnes: x.argent > 0;

C6: $\forall\ x \in$ LesPersonnes: x.epoux=nil $\Rightarrow \neg(\exists\ y \in x$.enfants);

C7: $\exists\ x \in$ LesPersonnes: x.ami=nil;

FIG. 4.2 - *Contraintes d'intégrité*

Nous pouvons donc dire qu'une base de données B *satisfait* une contrainte C, qu'elle est un modèle de cette contrainte (noté $B \models C$), ou encore qu'elle est *cohérente* vis-à-vis de cette contrainte. Dans ce domaine, le vocabulaire est riche et n'a pas toujours le même sens qu'en logique. Par exemple, en logique, une interprétation ne peut pas être incohérente. Par contre une théorie peut l'être. Soulignons que le symbole $\in$ est un symbole de relation binaire quelconque. L'emploi de ce symbole n'est pas anodin, car il permet de "simuler" l'appartenance ensembliste. Cependant, il n'est pas assujetti à respecter les axiomes de la théorie des ensembles (heureusement ! De même, les entiers `integer` ne satisfont pas les axiomes de Peano). L'appellation d'objet "structuré comme un ensemble" est donc abusive, mais bien pratique. Ajoutons que, dans notre cadre, les ensembles sont tous finis, et qu'un objet de type ensemble peut s'appartenir à lui même, ce qui est totalement inoffensif. Cette remarque est importante dans la mesure où nous utilisons dans la suite des techniques de démonstration automatique. En pratique, nous n'aurons pas besoin d'un démonstrateur de théorèmes spécifique à la théorie des ensembles, mais nous pourrons nous contenter d'un démonstrateur en logique du premier ordre général.

4.1.5 Implantation des méthodes

Une base de données orientée-objet peut être modifiée par des *méthodes*. Les méthodes permettent de décrire, à l'intérieur même du schéma, les aspects comportementaux des objets. Nos méthodes sont définies de façon standard par un *nom*, une *signature*, et une *implantation*.

Nous avons déjà mentionné la signature des méthodes. Il nous reste à définir leur *implantation*. Nous soulignons le fait qu'une méthode peut être définie dans plusieurs classes. Elle est alors *surchargée*. Lors d'un appel de méthode, l'implantation à exécuter est déterminée par la classe d'appartenance de l'objet destinataire. Ceci est fait à l'exécution : on parle alors de résolution tardive.

À l'instar de la plupart des langages de programmation pour bases de données commerciaux, le langage de programmation que nous présentons est de type impératif. Encore une fois, nous nous limitons dans un premier temps à un nombre restreint d'instructions. On suppose l'existence d'un ensemble de paramètres $\mathcal{P} = \{p_1, p_2, ...\}$. Comme les contraintes d'intégrité, les méthodes sont construites à partir des termes et des formules des définitions 20 et 21. En outre, le terme `self` dénote le destinataire de l'appel de méthode, c'est-à-dire l'objet auquel on applique cet appel de méthode.

Définition 23 L'ensemble des instructions est défini comme suit :

- Soient t_1 et t_2 des termes, et a un attribut, "$t_1.a$:=t_2" est une instruction (affectation).
- Soient t, t_1,... ,t_n des termes et m un nom de méthode, "t`<-`$m(t_1, ..., t_2)$" est une instruction (appel de méthode).
- Soient i_1 et i_2 deux instructions, "i_1;i_2" est une instruction (séquencement).
- Soit i une instruction, "`{`i`}`" est une instruction (bloc d'instructions).
- Soient i une instruction et F une formule, "`if` F `then` i" est une instruction (conditionnelle).
- Soient x une variable, F une formule et i une instruction, "`forall` x `where` F `do` i" est une instruction (boucle).

À nouveau, nous masquons le fait que le langage est fortement et statiquement typé. Le but n'étant pas de définir en toute rigueur un nouveau langage de programmation pour bases de données, nous de donnons pas de sémantique formelle pour le langage décrit précédemment. En effet, une connaissance intuitive de la signification de chaque instruction suffit amplement pour exposer notre approche, sans avoir besoin de considérer une multitude de détails fastidieux.

Nous soulignons le fait que nous avons défini les appels de méthodes non comme des termes, mais comme des instructions. En effet, dans ce travail, nous ne nous intéressons qu'aux méthodes qui ne renvoient pas de résultat. Nous nous sommes avant tout interressés aux effets de bord des méthodes sur la base de données.

Définition 24 Une méthode est définie comme suit :

`method` *m* (*paramètres*) `in class` κ *corps*.

où "*m*" est le nom de la méthode, "*paramètres*" est une liste de paramètres (avec indication de leur type), κ est la classe d'appartenance du receveur, et "*corps*" est une instruction (en général un bloc d'instructions).

Exemple 3 Nous donnons en figure 4.3 et 4.4 des exemples de méthodes. Commençons par les méthodes de la figure 4.3. Ces méthodes ont été écrites de façon à ce qu'elles ne violent pas au moins une des contraintes `C1`,...,`C7` données en figure 4.2, page 46.

- La méthode `separe` de la classe `Personne` met à `nil` l'attribut `epoux` d'un objet `self`. Il est clair que cette méthode ne peut pas violer la contrainte `C1`, qui dit qu'il existe au moins une personne qui n'est pas mariée.

- La méthode `mariage` de la classe `Personne` marie deux personnes `self` et q, en vérifiant toutefois que ces deux personnes sont différentes. Elle ne viole pas la contrainte `C2` qui dit qu'une personne ne peut pas être mariée avec elle-même.
- La méthode `mariage` dans la classe `Parent` marie deux personnes `self` et q, en vérifiant toutefois que `self` n'est pas un enfant de q et réciproquement. Cette méthode ne peut pas violer la contrainte `C3` qui dit qu'une personne ne peut pas être mariée à l'un de ses enfants.
- La méthode `separe` de la classe `Employe` met à `nil` l'attribut `epoux` d'une personne `self`, après avoir vérifié que `self` a un patron. Elle ne peut pas violer la contrainte `C4` qui dit qu'une personne qui n'est pas mariée doit avoir un patron.
- La méthode `debit` dans la classe `Personne` soustrait `somme` de l'attribut `argent` d'une personne `self`. Toutefois, cette opération n'est faite que si il y assez d'argent sur le compte. Cette méthode ne viole pas la contrainte `C5`, qui dit que toute personne doit posséder de l'argent.
- La méthode `debit` dans la classe `Parent` soustrait `somme` de l'attribut `argent` d'une personne `self` et fait de même pour chacun de ses enfants. Cette méthode prend le même type de précautions que la méthode `débit` dans la classe `Personne` et ne viole donc pas la contrainte `C5`.

Les méthodes précédentes sont simples. Elles opèrent des modifications tout en prenant certaines précautions vis-à-vis des contraintes d'intégrité. Pour l'instant, le langage utilisé est relativement restreint et n'exploite pas véritablement les possibilités offertes par les langages de programmation pour bases de données.

En plus des instructions classiques, ces langages permettent de consulter la base de données par l'intermédiaire de requêtes. Un aspect intéressant des langages de requêtes est qu'ils sont fondés sur la logique du premier ordre. Une requête est de la forme : $\{x|P(x)\}$, et dénote l'ensemble des objets de la base qui satisfont $P(x)$. Les requêtes peuvent aussi servir à tester un état de la base de données. De telles requêtes sont appelées des *requêtes booléennes*. Elles sont en fait similaires aux contraintes d'intégrité, mais sont utilisées différemment. Nous avons vu précédemment que certaines méthodes devaient prendre des précautions particulières avant de modifier la base de données. Par exemple, la méthode `mariage` de la classe `Personne` teste que les deux époux `self` et q sont différents. Dans certains cas, une simple expression booléenne (sans quantificateurs) ne suffit pas pour faire un tel test. Il faut souvent faire appel à une requête booléenne. Considérons par exemple la méthode suivante :

```
method meilleur-ami(q: Personne) in class Personne {
                        self.ami:=q;}
```

Cette méthode affecte q à l'attribut `ami` d'un objet `self`. Elle peut manifestement violer la contrainte `C7` qui dit qu'au moins une personne n'a pas de meilleur ami. Une violation aura lieu à coup sûr si `self` est la seule personne à ne pas avoir de meilleur ami. Avant de faire cette modification, il faut donc tester si il existe une autre personne, différente de `self`, qui n'a pas de meilleur ami. Ceci est fait à l'aide de la requête booléenne suivante :

$$\texttt{exists } u \in \texttt{LesPersonnes: } u \neq \texttt{self} \wedge u\texttt{.ami} = \texttt{nil}$$

```
method separe() in class Personne {
    self.epoux:=nil;}

method mariage(q:Personne) in class Personne {
  if (self≠q){
    self.epoux:=q;
    q.epoux:=self;}}

method mariage(q:Personne) in class Parent {
    if (self ≠ q ∧ ¬(q ∈ self.enfants) ∧ ¬(self ∈ q.enfants)){
    self.epoux:=q;
    q.epoux:=self;}}

method separe() in class Employe {
  if (self.patron≠nil)
    self.epoux:=nil;}

method debit(somme:integer) in class Personne {
  if (self.argent-somme>0)
    self.argent:=self.argent-somme;}

method debit(somme:integer) in class Parent {
  if (self.argent-somme>0)
    self.argent:=self.argent-somme;
  forall u where u ∈ u.enfants do
    if (u.argent-somme>0)
      u.argent:=u.argent-somme;}
```

FIG. 4.3 - *Méthodes simples.*

La version corrigée de la méthode `meilleur-ami`, qui intègre ce test, est donnée en figure 4.4.

La méthode `separe` de la classe `Parent` utilise aussi une requête booléenne. Elle met à `nil` l'attribut `epoux` d'une personne `self`. Toutefois, elle vérifie préalablement que `self` n'a pas d'enfants. Elle ne peut donc pas violer la contrainte `C6` qui dit qu'une personne qui n'est pas mariée ne peut pas avoir d'enfants.

```
method meilleur-ami(q: Personne) in class Personne{
    if (exists u ∈ LesPersonnes: u≠self ∧ u.ami=nil)
      self.ami:=q;}

method separe() in class Parent {
  if (¬(exists u ∈ self.children))
    self.epoux:=nil;}
```

FIG. 4.4 - *Méthodes avec requêtes booléennes.*

On peut s'interroger sur l'opportunité d'imbriquer de tels tests dans les méthodes. Ce style de programmation, appelé *programmation défensive*, a des avantages mais aussi des inconvénients.

Parmi les avantages, les modifications qui sont opérées par des méthodes qui préservent les contraintes ne risquent pas d'être annulées (puisqu'elles préservent forcément l'intégrité) et les contraintes n'ont plus à être vérifiées à l'exécution.

Par contre, les tests qui sont imbriqués par le programmeur dans les méthodes ont un coût. Celui-ci est faible dans le méthode `mariage` de la classe `Personne`. Il peut par contre être élevé si le programmeur utilise des requêtes booléennes. Évaluer la requête booléenne peut être aussi coûteux que tester la contrainte d'intégrité après l'exécution de la méthode. C'est le cas par exemple pour la méthode `meilleur-ami`. En outre, la programmation défensive peut être dangereuse car la stabilité des méthodes vis-à-vis des contraintes n'est plus assurée. Si les contraintes changent, les méthodes doivent aussi être modifiées afin de mettre en œuvre les précautions appropriées.

En dehors du débat sur la discipline de programmation qu'il est bon d'adopter, nous soulignons que, dans les systèmes réels, la programmation défensive est chose courante. À ce titre, il est bon d'avoir un mécanisme d'analyse statique afin de la détecter et d'éviter de vérifier les contraintes d'intégrité à l'exécution, ce qui serait redondant avec les précautions déjà prises par le programmeur à l'intérieur des méthodes. Enfin, si la programmation défensive n'est pas autorisée (ou du moins déconseillée), il faut aussi pouvoir la détecter pour prévenir le programmeur du danger.

4.2 Décomposition des contraintes et des instructions

La méthode de preuve de programme que nous présentons dans le prochain chapitre fournit des informations très détaillées sur l'effet potentiel qu'une méthode peut avoir sur une contrainte. Toutefois, elle ne peut être appliquée qu'à des contraintes et des méthodes qui sont sous une certaine forme, que nous appellerons *forme décomposée*. Nous montrons dans cette section comment ramener automatiquement n'importe quelle contrainte ou méthode à sa forme décomposée.

Soulignons qu'en démonstration automatique de théorèmes, une simplification préliminaire des formules est souvent nécessaire. Par exemple, toutes les techniques inspirées de la résolution requièrent la mise sous forme normale, puis sous forme de Skolem, des formules manipulées.

Les formes décomposées des formules et des méthodes sont définies en partant du même principe : limiter l'utilisation de termes complexes. Commençons par les formules.

4.2.1 Forme décomposée des formules

Considérons une formule atomique simple quelconque. Par exemple, celle qui apparaît dans la contrainte `C3` de la figure 4.2, page 46 :

$$x.\mathtt{epoux} \notin x.\mathtt{enfants}$$

Supposons que cette formule est satisfaite pour un objet x quelconque. Il est clair que cette formule peut devenir fausse après une modification de la base de données dans un des trois cas (non exclusifs) suivants :

- L'attribut "`epoux`" de x est modifié.
- L'attribut "`enfants`" de x est modifié.
- Une insertion est faite dans $x.\mathtt{enfants}$.

De façon générale, afin d'étudier l'effet d'une modification sur une formule, il faut bien faire ressortir les différents cas pour lesquels cette formule risque d'être remise en cause. Ceci permettra, en outre, de déterminer facilement la façon dont elle est remise en cause, et de trouver éventuellement une formule qui est satisfaite *après* la modification. Ceci sera le but du prochain chapitre. Pour l'instant, nous allons nous contenter de définir une autre formule qui est équivalente à la formule de départ et qui, d'une certaine façon, fasse mieux ressortir ces différents cas. Commençons par une définition :

Définition 25 Soit t_1,...,t_n des termes, g une fonction n-aire, et F une formule qui comprend une occurrence du terme $g(t_1, ..., t_n)$ telle qu'aucune variable de ce terme n'est liée dans F. (c'est à dire que F peut s'écrire $F(g(t_1, ..., t_n)))$. Une décomposition de F est une formule :

$$\exists u : g(t_1, ..., t_n) = u \wedge F(u)$$

où u est une nouvelle variable.

Il est clair que, pour peu que le symbole "=" soit interprété comme l'égalité, une formule est équivalente à n'importe laquelle de ses décompositions.

Exemple 4 Les décompositions de la formule : $x.\mathtt{epoux} \notin x.\mathtt{enfants}$ sont les suivantes :

- $\exists\ u$: $x.\mathtt{epoux}{=}u \wedge u \notin x.\mathtt{enfants}$
- $\exists\ u$: $x.\mathtt{enfants}{=}u \wedge x.\mathtt{epoux} \notin u$

Nous voyons bien dans cet exemple que le but de la décomposition est de limiter le nombre de termes complexes dans chaque atome. Par termes complexes, nous entendons des termes de profondeur[1] supérieure ou égale à 1.

Définition 26 Une formule F est totalement décomposée si elle ne contient que des atomes de la forme :

- $R(x_1, x_2)$, où R est un symbole de relation binaire (=,≠,>, ...).
- $R(f(x_1, x_2), x_3)$, où R est soit "=", soit "≠", et f est un symbole de fonction (binaire ou unaire; dans dernier ce cas x_2 n'est pas pris en compte).

x_1, x_2 et x_3 dénotent des variables ou des constantes.

L'avantage de la décomposition est le suivant :

> *les formules atomiques d'une formule totalement décomposée faisant référence à un attribut "a" seront toutes de la forme :* $x_1.a = x_2$ *ou* $x_1.a \neq x_2$.

Cette propriété tient à la définition même d'une formule totalement décomposée. Ce format facilite considérablement l'étude de l'effet d'une instruction (qui, par définition, ne fait que modifier les attributs des objets) sur une formule.

On peut ensuite facilement prouver que, pour toute formule F, il existe une formule F', totalement décomposée, qui est équivalente à F. Ceci se fait en extrayant un par un les termes des formules atomiques, jusqu'à ce que la formule soit totalement décomposée. Prenons par exemple une formule non totalement décomposée :

$$F \equiv [f(t_1, t_2) = y]$$

En outre, supposons que les termes de F ont une profondeur au plus de n. Une décomposition de F est par exemple :

$$F' \equiv [\exists x_1 : \exists x_2 : (t_1 = x_1) \wedge (t_2 = x_2) \wedge f(x_1, x_2) = y]$$

Cette formule contient des termes d'au plus une profondeur de $n - 1$. Nous avons donc la garantie que la décomposition termine et qu'elle aboutira tôt ou tard à une formule totalement décomposée, en décomposant successivement les formules obtenues. Nous pouvons en outre utiliser quelques transformations triviales pour simplifier le processus : $w = u.a$ peut se réécrire $u.a = w$, etc.

Exemple 5 Décomposons la contrainte `C3` :

$$\forall\ x \in \texttt{LesPersonnes:}\ x\texttt{.epoux} \notin x\texttt{.enfants}$$

L'extraction du terme x`.epoux` donne :

$$\forall\ x \in \texttt{LesPersonnes:}\ \exists\ u_1\texttt{:}\ x\texttt{.epoux=}u_1 \wedge u_1 \notin x\texttt{.enfants}$$

1. La *profondeur* d'un terme est définie dans la définition 3, page 10

L'extraction du terme x.enfants donne :

$$\forall\ x\ \in\ \texttt{LesPersonnes:}\ \exists\ u_1\texttt{:}\ \exists\ u_2\texttt{:}\ x\texttt{.enfants=}u_2\ \wedge\ x\texttt{.epoux=}u_1\ \wedge\ u_1\notin u_2$$

Soulignons que cette décomposition peut être faite de différentes façons mais que les résultats sont équivalents. Par contre, il y a des manières plus ou moins efficaces de l'opérer. Par exemple, prenons la formule suivante :

$$x\texttt{.epoux}\notin a\ \wedge\ x\texttt{.epoux}\notin b$$

Les deux décompositions suivantes sont valides :

- $\exists\ u_1\texttt{:}\ \exists\ u_2\texttt{:}\ x\texttt{.epoux=}u_1\ \wedge\ x\texttt{.epoux=}u_2\ \wedge\ u_1\notin a\ \wedge\ u_2\notin b$
- $\exists\ u\texttt{:}\ x\texttt{.epoux=}u\ \wedge\ u\notin a\ \wedge\ u\notin b$

Nous choisirons évidement la formule la plus petite. Le principe mis en œuvre ici est simplement de ne pas utiliser deux fois le même terme (x.epoux, ici) pour décomposer une formule.

Pour terminer, à chaque fois que nous rencontrerons les formules : “$\forall x \in S : P(x)$” et “$\exists x \in S : P(x)$”, nous les remplacerons respectivement par : “$\forall x : x \in S \Rightarrow P(x)$” et “$\exists x : x \in S \wedge P(x)$”. Les deux notations sont bien évidement équivalentes.

4.2.2 Décomposition des instructions

Par abus, nous identifions une méthode à l'ensemble des instructions qu'elle exécute (son corps). La décomposition des instructions ressemble assez à celles des formules. En fait, la seule instruction concernée par une éventuelle décomposition est l'instruction d'affectation. Considérons une instruction d'affectation quelconque :

$$p\texttt{.epoux.argent:=}p\texttt{.epoux.argent}+1\texttt{;}$$

qui ajoute 1 à la somme d'argent de l'époux de p. Cette instruction complexe regroupe en son sein des opérations de nature différente :

- Elle détermine dans un premier temps les objets sur lesquels elle va faire la modification. À cette fin, elle “navigue” le long des chemins définis par les termes. Soit u un objet tel que u=p.epoux, v un entier tel que v=u.argent, et w un autre entier tel que $w = v + 1$. L'instruction de départ peut se réécrire u.argent:=w.
- Ensuite, l'instruction est exécutée.

Pour opérer la décomposition, nous avons introduit des nouveaux symboles : u, v et w, respectant certaines conditions. Le langage de méthodes tel qu'il est défini en définition 23, page 47, ne permet pas de faire ceci. Pour cette raison, nous devons introduire une nouvelle instruction, dont le seul but est de définir des variables qui satisfont certaines conditions. Cette instruction s'écrit :

$$\texttt{forone}\ x\ \texttt{where}\ f(x_1, ..., x_n) = x\ \texttt{do}\ \textit{instruction}$$

x est une nouvelle variable, f est une fonction n-aire totale, et x_1,...,x_n sont des variables ou des constantes différentes de x. La signification de cette instruction est que "*instruction*" est exécutée avec le seul objet x tel que $f(x_1, ..., x_n) = x$. Nous soulignons que par définition, x existe et est unique. Munis de cette nouvelle instruction, nous sommes en mesure de définir la décomposition des instructions d'affectation.

Définition 27 Soit i une instruction d'affectation telle que $i \equiv t_1.a := t_2$, où t_1 et t_2 sont des termes, et a est un attribut. Soit t un terme qui a une occurrence dans t_1 ou t_2. Une décomposition de i est une instruction de la forme :

$$\texttt{forone } x \texttt{ where } t = x \texttt{ do } t_1[x/t].a := t_2[x/t]$$

où x est une nouvelle variable, $t_i[x/t]$ dénote la substitution de t par x dans t_i, et la formule $x = t$ est elle même totalement décomposée.

Exemple 6 Reprenons l'instruction :

```
p.epoux.argent:=p.epoux.argent+1;
```

Voici un exemple de décomposition de cette instruction :

```
forone u where p.epoux=u do u.argent:=u.argent+1;
```

Par contre l'instruction suivante n'en est pas une décomposition :

```
forone u where p.epoux.argent=u do p.epoux.argent:=u+1;
```

car la formule p.epoux.argent$=u$ n'est pas totalement décomposée.

Définition 28 Soient v_1 et v_2 des variables ou des constantes, et a un attribut. Une méthode est totalement décomposée si toutes ses affectations sont de la forme : $v_1.a$:=v_2, et si toutes les formules auxquelles elle fait référence sont totalement décomposées.

Exemple 7 Une décomposition totale de l'instruction :

```
p.epoux.argent:=p.epoux.argent+1;
```

pourrait par exemple être :

```
forone u where p.epoux=u do
 forone v where u.argent=v do
  forone w where v+1=w do
   u.argent:=w;
```

Notez que la substitution de v à u.argent n'a été faite qu'en partie droite de l'affectation, conformément à la définition de la décomposition. En effet, on aurait sinon abouti à : v:=u, qui est une affectation entre variables qui ne modifie pas la base de données, contrairement à l'instruction initiale.

Nous pouvons nous demander si cette transformation préserve bien la sémantique des instructions. Nous nous garderons bien d'aborder cette question sous cet angle, car nous n'avons pas défini de sémantique pour notre langage. En fait, le but de l'instruction "`forone`" est justement de définir une sémantique opérationnelle claire et concise pour les instructions d'affectation complexes. L'utilisation d'une telle instruction est assez habituelle lorsqu'il faut définir précisément la sémantique opérationnelle d'un langage (voir par exemple [AH87]). Elle prend souvent la forme suivante :

`let` *x*=*expression* `in` *instruction*

Cette instruction définit la variable *x* comme étant l'abréviation de "*expression*" dans "*instruction*".

Ajoutons que la forme décomposée d'une instruction exige que, non seulement les affectations soient élémentaires, mais qu'en outre toutes les formules qui apparaissent dans l'instruction, en particulier dans les "`if`" et les "`forone`" le soient aussi. Pour conclure, soulignons que le but de la décomposition et de l'instruction `forone` est avant tout de prendre en compte les aspects navigationnels des langages de programmation orientés objet.

4.3 Conclusion

Nous avons présenté dans ce chapitre le cadre d'application de notre approche. Il s'agit d'un modèle de système de gestion de bases de données inspiré du système O_2, qui permet d'exprimer un schéma de bases de données comportant des classes décrivant la structure des objets, les méthodes associées, ainsi que des contraintes d'intégrité, dans un langage proche de la logique du premier ordre.

Nous avons aussi décrit une manière de simplifier la structure des contraintes et des méthodes, par le biais d'une décomposition. Cette décomposition est automatique et a pour but de faciliter une analyse approfondie de l'effet potentiel qu'une méthode peut avoir sur une contrainte, ce qui sera fait au prochain chapitre.

Nous définissons dans le prochain chapitre une approche afin de prouver qu'une méthode préserve une contrainte. Cette approche est à la fois fondée (c'est-à-dire fiable), et peut, a priori, s'appliquer à n'importe quelles contraintes et méthodes : elle est générale. Ceci ne veut pas dire qu'elle réussit toujours à prouver qu'une méthode préserve une contrainte. Dans certains cas, notre approche conclut à tort qu'une méthode peut potentiellement violer une contrainte. Elle pêche par excès de prudence. Par contre, elle est toujours fiable : elle ne parviendra jamais à prouver qu'une méthode préserve un contrainte si ce n'est pas effectivement le cas. Notre approche est donc valide, générale, mais pas complète.

Pour terminer, nous soulignons que notre approche est en grande partie indépendante du modèle de données que nous avons présenté. Par contre, ce modèle de données est le cadre d'application idéal de cette approche. Les traitements qui sont opérés sont en effet de taille réduite et très modulaires. Ceci facilite l'analyse statique que nous décrivons.

Chapitre 5

Transformateurs de prédicats : définition et application

Nous présentons dans ce chapitre le "noyau" de notre méthode. Nous commençons par définir un transformateur de prédicats afin de capturer la sémantique des méthodes. Contrairement aux transformateurs de prédicats du calcul de Dijkstra du chapitre 3, le transformateur de prédicats que nous définissons ne prétend pas fournir une information *totale* sur le comportement potentiel d'un programme. Le but n'est pas de détecter à coup sûr qu'une méthode ne violera jamais une contrainte (ce qui n'est pas possible), mais de fournir une technique de détection qui, bien que partielle, soit *véritablement opérationnelle*. Nous favoriserons donc volontairement la validité et la simplicité du transformateur de prédicats, au détriment de sa complétude, ceci afin d'éviter d'aboutir à une approche trop lourde, que la complexité rendrait inutilisable. Le transformateur de prédicats que nous présentons en premier lieu fonctionne en *avant*, c'est-à-dire qu'il analyse les instructions selon leur ordre d'écriture dans les méthodes (la première, la seconde...). Il est aussi possible de définir une transformateur de prédicats *arrière*, qui analyse d'abord la dernière instruction, puis l'avant dernière... Nous présentons un tel transformateur de prédicats à la fin de ce chapitre. Dans un premier temps, nous nous concentrerons sur la définition et l'étude du transformateur de prédicats *avant*.

Il y a différentes façons de présenter le transformateur de prédicats. La plus classique consiste à le définir "en bloc", puis à montrer que cette définition est correcte. Cette présentation masque le fait que le transformateur de prédicats n'est pas un don du ciel, mais que c'est une *construction*, conçue d'emblée pour capturer la sémantique des méthodes. Ce n'est pas lui qui régit le comportement des méthodes : il n'en est que le reflet fidèle. Montrer complètement le processus de sa construction n'a pas qu'un but pédagogique. Bien au contraire, ce processus permet de prendre pleinement conscience de la généralité de cette approche, et de définir des transformateurs de prédicats pour des langages différents.

Nous ne ferons pas pour autant l'économie d'une présentation classique du transformateur de prédicats, et notamment de sa preuve de correction, qui seule fait foi. Ce chapitre ne sera donc pas exempt de redondances : la preuve du théorème de correction du transformateur de prédicats reprend en fait une à une les étapes de sa construction. Le lecteur déjà convaincu de sa correction à la fin de la section 5.2 pourra donc sauter cette preuve.

Nous verrons, en section 5.4, l'application assez naturelle du transformateur de prédicat à notre problème.

Le transformateur de prédicats permet de ramener la preuve qu'une méthode préserve une contrainte d'intégrité à la démonstration d'un théorème en logique du premier ordre.

Ensuite, nous montrerons comment étendre la transformateur de prédicats afin de prendre en compte différents aspects des modèles orientés-objet : les valeurs structurées, les ensembles, l'héritage, etc.

Enfin, nous définirons une autre approche pour prouver qu'une méthode préserve une contrainte, fondée sur un transformateur de prédicats *arrière*. Celui-ci peut en outre être utilisé pour corriger (...en théorie) le code d'une méthode afin que celle-ci préserve les contraintes.

Avant toutes choses, afin de justifier l'analyse que nous présentons dans ce chapitre, nous exposons en section 5.1 deux approches différentes. La première est naïve et non fondée. La seconde est trop spécifique et ne peut être que rarement mise en œuvre.

5.1 Fausses pistes

En préliminaire à ce chapitre, nous répondons dans cette section aux questions suivantes :

– Une approche naïve peut-elle suffire pour prouver qu'une méthode préserve une contrainte ?

– En restreignant la classe des méthodes et contraintes considérées, ne peut-on pas obtenir une preuve à moindre coût ?

À cette fin, nous définirons deux approches pour prouver qu'une méthode préserve une contrainte.

– La première approche est une analyse naïve qui simule grossièrement les méthodes. Nous verrons que ce type d'analyse est non fondé.

– La seconde approche arrive, en restreignant drastiquement la classe des contraintes et méthodes considérées, à fournir des résultats à moindre frais. Son cruel défaut est son absence totale de généralité.

5.1.1 Une approche naïve

La première impression que l'on a, lorsqu'on essaie de prouver qu'une méthode préserve une contrainte, c'est qu'une analyse grossière pourrait certainement suffire à déterminer les propriétés de cette méthode. Le premier réflexe est donc de faire une *quasi-exécution* du programme, et de vérifier que les données résultantes de cette exécution satisfont la contrainte. Par *quasi-exécution*, nous entendons une analyse naïve du programme, où les paramètres sont confondus avec les objets qu'ils dénotent et les instructions sont exécutées sur ces objets. Le principe sous-jacent d'une telle approche est le suivant :

> *Il suffit, pour prouver qu'une méthode préserve une contrainte, de simuler une exécution de cette méthode et de vérifier que les objets manipulés par cette méthode satisfont toujours la contrainte.*

Ce principe est erroné. Voyons les conséquences de ce principe sur deux exemples. Prenons par exemple la méthode et la contrainte ci-dessous :

```
method separe() in class Personne{
    self.epoux:=nil;}

C1: ∃ x ∈ LesPersonnes: x.epoux=nil;
```

Si nous exécutons effectivement le code de cette méthode, alors la base de données résultante contiendra un objet dénoté par `self`, tel que `self.epoux` = `nil`. Nous pouvons donc conclure que la contrainte `C1` sera toujours satisfaite. Le principe énoncé précédemment semble être suffisant pour conclure qu'une méthode préserve une contrainte.

Prenons maintenant une autre méthode et une autre contrainte, données ci-dessous, et appliquons le même principe. La méthode `mariage2` opère un mariage entre deux personnes `self` et q, alors que la contrainte `C8` dit que les liens du mariage doivent être réciproques.

```
method mariage2(q:Personne) in class Personne{
    self.epoux:=q;
    q.epoux:=self;}

C8: ∀ x ∈ LesPersonnes: x.epoux.epoux=x ∨ x.epoux=nil;
```

Une exécution de cette méthode aboutit à une base de données où les faits suivants sont vrais :

$$\texttt{self.epoux}=q \quad \text{et} \quad q\texttt{.epoux}=\texttt{self}$$

Nous pouvons voir ces deux faits comme définissant une interprétation et tout simplement vérifier la contrainte sur `self` et q. La contrainte étant satisfaite, nous concluons que la méthode `mariage2` ne peut pas violer la contrainte `C8`.

Le problème est que la méthode `mariage2` peut effectivement violer la contrainte `C8`, et que, partant du principe énoncé auparavant, nous concluons le contraire. Voyons un cas de violation : considérons une base de données comportant trois objets distincts o_1, o_2, o_3 tels que :

$$o_1\texttt{.epoux}=o_2 \text{ et } o_2\texttt{.epoux}=o_1$$

L'appel de la méthode o_1`<-mariage2(`o_3`)` va mener à une base de données où les faits suivants seront satisfaits :

$$o_1\texttt{.epoux}=o_3 \text{ et } o_3\texttt{.epoux}=o_1 \text{ et } o_2\texttt{.epoux}=o_1$$

Il est clair que cette base de données ne satisfait pas la contrainte `C8`, car $o_2\texttt{.epoux.epoux}\neq o_2$ et $o_2\texttt{.epoux}\neq\texttt{nil}$. Remarquons que ce cas particulier implique un objet o_2, qui ne correspond pas à un paramètre du programme. C'est la raison pour laquelle une quasi-exécution naïve de la méthode ne peut suffire pour détecter une violation potentielle. Ceci met en valeur une particularité remarquable de notre approche par rapport aux analyses conventionnelles : *les objets que l'on doit considérer ne se limitent pas aux objets* *manipulés par les programmes*. En effet, les méthodes manipulent des bases de données et une analyse se limitant aux objets manipulés par les méthodes ne permet pas de rendre compte de tous les risques de violation de l'intégrité. Il faut au contraire prendre en compte l'environnement initial sur lequel la méthode est exécutée, qui, contrairement à ce qui se passe dans les langages de programmation non persistants, n'est pas vide. Cet environnement de départ, c'est une base de données qui peut déjà contenir de nombreux objets (en l'occurrence l'objet o_2) et qui, en outre, satisfait la contrainte d'intégrité.

Nous voyons donc clairement qu'une approche naïve ne peut suffire à la preuve de programme. Il est donc nécessaire de nous placer à un autre niveau, plus abstrait, mais qui permet par contre de définir une analyse *sémantiquement fondée*.

5.1.2 Analyses ad-hoc

Les analyses ad-hoc sont fondées sur l'espérance que dans certains cas très précis, un mécanisme rudimentaire suffit à déterminer les propriétés d'une méthode. En effet, il est parfois possible de définir une analyse valide en restreignant drastiquement la classe des méthodes et des contraintes considérées. Nous donnons ci-dessous un exemple particulier d'analyse ad-hoc.

Intéressons nous particulièrement aux contraintes de la forme :

$$C \equiv \forall x \in S : \bigvee_{i \in [1..n]} R_i(x.a_i, c_i)$$

où S est un ensemble d'éléments d'une classe κ (une racine de persistance), les R_i sont des prédicats binaires appartenant à $\{=, \neq, \geq, \leq, >, <)\}$, et les c_i et les a_i sont respectivement des constantes et des attributs. Définissons maintenant une classe de méthodes :

$$\begin{array}{l} \texttt{method } m\texttt{() in class } \kappa\ \texttt{\{} \\ \quad \texttt{self.}a_{j_1}\texttt{:=}d_{j_1}\texttt{;} \\ \quad \texttt{...} \\ \quad \texttt{self.}a_{j_m}\texttt{:=}d_{j_m}\texttt{;\}} \end{array}$$

où $j_1, ..., j_m \in [1..n]$. Nous supposerons en outre que les a_{j_k} dénotent tous des attributs distincts (il n'y a pas plus d'une affectation par attribut).

Comme m ne modifie que les attributs d'un seul objet `self`, il est clair que la méthode m ne pourra violer la contrainte C que si, après l'exécution de m, la formule suivante est vraie :

$$\bigwedge_{i \in [1..n]} \neg R_i(\texttt{self}.a_i, c_i)$$

Qui est équivalente à :

$$\bigwedge_{i \in [1..n]} \neg R_i(d_i, c_i)$$

Cette condition ne porte que sur des constantes et peut donc être testée à la compilation. Voyons une application concrète de ce critère. Ajoutons à la définition de la classe `Personne` un attribut `age` de type `integer`, et définissons une contrainte qui dit : “une personne qui a moins de dix huit ans ne peut pas être à découvert”. Cette contrainte se traduit par :

$$\texttt{C9} : \forall x \in \texttt{LesPersonnes} : x.\texttt{age} > 18 \vee x.\texttt{argent} \geq 0$$

Considérons maintenant la méthode suivante :

```
method init-Personne() in class Personne {
    self.age:=0;
    self.argent:=0;}
```

Selon le critère énoncé précédemment, la méthode `init-Personne` ne peut violer la contrainte `C9` que si $(0 < 18) \wedge (0 < 0)$, condition qui est trivialement évaluée à faux. Nous avons donc prouvé que `init-Personne` préserve la contrainte `C9`.

Notez que cette technique est valide, et que, malgré sa simplicité, sa validité est prouvée avec rigueur. Il est clair que, disposant d'une telle approche et dans ce cas particulier, il serait absurde d'utiliser une analyse plus complexe. De façon générale, les analyses ad-hoc, pour peu qu'elles soient valides, seront très efficaces car taillées sur mesure pour une classe donnée de méthodes et de contraintes. Toutefois, la classe de méthodes et de contraintes à laquelle cette analyse s'applique est tellement restreinte qu'elle ne pourra être que rarement employée. En outre, si on cherche à généraliser cette approche, il est extrêmement difficile d'isoler comme nous l'avons fait de telles classes et de définir une analyse fiable. Nous ne pourrons donc pas nous passer d'une méthode générale.

En conclusion, pour détecter de façon satisfaisante les méthodes qui préservent les contraintes d'intégrité, il nous faut une approche qui soit à la fois fiable et fondée, à l'opposé de l'approche naïve, et générale, à l'opposé des méthodes ad-hoc. C'est pour satisfaire ces deux exigences que notre approche est fondée sur l'utilisation de *transformateurs de prédicats*.

5.2 Le transformateur de prédicats avant : construction

Nous montrons dans cette section les étapes de construction d'un *transformateur de prédicats avant*, $\mathcal{POST}$, pour le langage défini au chapitre 4. Nous rappelons qu'un transformateur de prédicats avant est une fonction qui prend en entrée un programme et une formule satisfaite par les données en entrée de ce programme, et fournit une autre formule, satisfaite par les données en sortie de ce programme. $\mathcal{POST}$ sera construit pas à pas, de façon à ce qu'il capture *effectivement* la sémantique du langage de méthodes, c'est-à-dire qu'il soit *correct*. Rappelons avant toutes choses la définition de la correction, qui sera employée tout au long de cette section.

Définition 29 Le transformateur de prédicats $\mathcal{POST}$ est *correct* si, pour toute instruction i, et toute formule F, si F est vraie avant l'exécution de i, alors $\mathcal{POST}(i, F)$ est vraie après l'exécution de i.

Nous nous concentrerons tout d'abord sur l'étude des méthodes *non-surchargées*. Analyser une méthode non-surchargée, c'est avant tout analyser les instructions qui la composent. Nous commettrons donc l'abus qui consiste à confondre une méthode avec les instructions de son corps. Nous commencerons par construire $\mathcal{POST}$ pour des catégories restreintes d'instructions et de formules : les affectations, et les formules atomiques. Nous supposerons sans rien perdre en généralité que les affectations et les formules sont totalement décomposées. Nous étendrons ensuite naturellement cette construction très partielle de $\mathcal{POST}$ aux formules et instructions quelconques, via une induction structurelle classique. Nous traiterons aussi le cas des boucles, des appels de méthodes, et des méthodes surchargées.

5.2.1 Affectations élémentaires et formules atomiques

L'idée est de trouver, étant donné une instruction d'affectation $i \equiv u.a{:=}v$ et une formule atomique F satisfaite avant l'exécution de cette affectation, une autre formule atomique satisfaite après cette exécution. u et v sont des constantes ou des variables, et a est un attribut (Nous voyons ici l'intérêt de la décomposition des instructions donnée au chapitre 4. En effet, tout programme peut se décomposer de telle façon que toutes ses affectations soient de cette forme). Le transformateur de prédicats $\mathcal{POST}$ doit refléter cette mise à jour. À cette fin, nous sommes amenés à considérer la forme de la pré-condition F. Si l'attribut a n'a pas d'occurrence dans F, alors l'instruction i est sans effet sur F. Par contre, si a a une occurrence dans F, alors F est soit de la forme $F \equiv (x.a = y)$, ou $F \equiv (x.a \neq y)$. Étant donné que F est totalement décomposée, ce sont les deux seuls cas où l'attribut a peut avoir une occurrence dans F. Plus précisément :

- Supposons que la formule en entrée de $\mathcal{POST}$ est : $F \equiv (x.a = y)$, où x et y sont des constantes ou des variables, et a est l'attribut déjà rencontré dans l'instruction. Si F est satisfaite par la base de données en entrée de i, alors :

  - Soit x et u sont égaux, et $u.a = v$ est vraie après l'exécution de i.
  - Soit x et u sont différents, et $u.a = v$, ainsi que $x.a = y$ sont vraies après l'exécution de i.

  Afin de refléter ces deux éventualités, nous définirons donc $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(\, i \,,\, F \,) \equiv ((x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a = y))}$$

- Supposons maintenant que la formule en entrée de $\mathcal{POST}$ est : $F \equiv (x.a \neq y)$. Si F est satisfaite par la base de données en entrée de i, alors :

  - Soit x et u sont égaux, et $u.a = v$ est vraie après l'exécution de i.
  - Soit x et u sont différents, et $u.a = v$, ainsi que $x.a \neq y$ sont vraies après l'exécution de i.

  Toujours dans le but de refléter ces deux possibilités, nous définirons donc $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(\, i \,,\, F \,) \equiv ((x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a \neq y))}$$

- Dans tous les autres cas de formules atomiques où l'attribut a n'a pas d'occurrences dans F ($F \equiv u \in w$, $F \equiv u = w$, $F \equiv u \neq w$, $F \equiv u > w$, $F \equiv u + v = w$, $F \equiv u \times v = w$, etc.), l'instruction i ne peut pas remettre en cause la valeur de vérité de la formule de départ. La formule de départ est donc naturellement préservée par la modification. Il faut en outre prendre en compte la formule “produite” par l'instruction, c'est-à-dire : $(u.a = v)$.

Nous définissons donc $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(\ i\ ,\ F\ ) \equiv (F \wedge u.a = v)}$$

5.2.2 Généralisation aux formules quelconques

Le transformateur de prédicats est défini pour les instructions élémentaires et pour les formules atomiques. La généralisation aux formules quelconques est naturelle et définit $\mathcal{POST}$ comme étant un endomorphisme de l'ensemble des formules, pour la disjonction et la conjonction. La conséquence pratique de cette remarque est que $\mathcal{POST}$ ne traite pas les formules de façon globale, mais seulement au niveau des littéraux. Commençons par la conjonction de formules.

Conjonction

Intuitivement, lorsque le transformateur de prédicats rencontre une conjonction, il calcule une post-condition pour chacun des membres de la conjonction. La post-condition pour la conjonction est en fait la conjonction des post-conditions obtenues. Nous étendons donc la définition de $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(i, F_1 \wedge F_2) \;\equiv\; \mathcal{POST}(i, F_1) \wedge \mathcal{POST}(i, F_2)}$$

Disjonction

Le cas de la disjonction est tout à fait similaire. Reprenons i, F_1 et F_2. Supposons que $(F_1 \vee F_2)$ soit vraie avant l'exécution de i. Alors deux cas se présentent :

– Soit F_1 est vraie avant l'exécution de i, et par hypothèse, $\mathcal{POST}(i, F_1)$ est vraie après l'exécution de i.

– Soit F_2 est vraie avant l'exécution de i, et par hypothèse $\mathcal{POST}(i, F_2)$ est vraie après l'exécution de i.

Donc naturellement : $\mathcal{POST}(i, F_1) \vee \mathcal{POST}(i, F_2)$ est vraie après l'exécution de i. Nous pouvons donc étendre la définition de $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(i, F_1 \vee F_2) \;\equiv\; \mathcal{POST}(i, F_1) \vee \mathcal{POST}(i, F_2)}$$

Quantification universelle

Lorsque le transformateur de prédicats rencontre une formule universellement quantifiée $(\forall x : F)$ et une instruction i, il calcule tout d'abord une post-condition $\mathcal{POST}(i, F)$ pour i et F. La variable x représentant un élément arbitraire, dans F comme dans $\mathcal{POST}(i, F)$, la post-condition vaut pour tout x. La quantification universelle peut donc être “recollée” au début de la post-condition. Nous étendons donc $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(i, \forall x : F) \;\equiv\; \forall x : \mathcal{POST}(i, F)}$$

Quantification existentielle

Le cas de la quantification existentielle est similaire. La quantification existentielle peut tout simplement être "évacuée" de la précondition et "recollée" sur la post-condition. Nous étendons donc $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(i, \exists x : F) \;\equiv\; \exists x : \mathcal{POST}(i, F)}$$

Négation

Le cas de la négation est un peu particulier. En fait, $\mathcal{POST}$ n'est pas à proprement parler défini pour la négation. Dans notre langage, tout prédicat est accompagné de sa négation ($=$ et $\neq$, $\in$ et $\notin$, $>$ et $\leq$, etc.). Le symbole de négation peut donc être intégré aux formules atomiques. Partant de ce principe, nous ne parlerons pas de littéraux, mais plus simplement d'atomes. En outre nous pouvons toujours transformer les formules de façon à ce que seules les formules atomiques soient niées, en utilisant les règles classiques suivantes :

- $\neg(F_1 \wedge F_2) \equiv \neg F_1 \vee \neg F_2$
- $\neg\forall x : P \equiv \exists x : \neg P$
- $\neg(F_1 \vee F_2) \equiv \neg F_1 \wedge \neg F_2$
- $\neg\exists x : P \equiv \forall x : \neg P$
- $\neg\neg F \equiv F$

À ce point, $\mathcal{POST}$ est correctement défini pour les formules quelconques, mais seulement pour les instructions élémentaires, c'est-à-dire l'affectation. Voyons comment étendre sa définition aux instructions composées.

5.2.3 Instructions composées

Séquence d'instructions

À partir d'une séquence d'instructions "$i_1; i_2$", où i_1 et i_2 sont des instructions, le transformateur de prédicats calcule une post-condition en "cumulant" l'effet potentiel de chaque instruction sur la pré-condition F. À partir de i_1 et de F, il produit une post-condition satisfaite avant l'exécution de i_2, à partir de laquelle il produit une autre post-condition satisfaite après l'exécution de "$i_1; i_2$". Nous étendons donc $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}((i_1;i_2), F) \equiv \mathcal{POST}(i_2, \mathcal{POST}(i_1, F))}$$

Instruction conditionnelle

La définition de $\mathcal{POST}$ pour l'instruction conditionnelle est plus complexe. Prenons une instruction conditionnelle $j \equiv$ "`if` G `then` i". Le transformateur de prédicats produit une post-condition à partir d'une pré-condition F et de j en considérant deux cas : G est soit vrai, soit faux. Le second cas est le plus simple, car j est alors sans effet et F est préservée. Dans le premier cas, $\mathcal{POST}$ doit calculer la post-condition de l'instruction i, sachant que F et G sont vraies au départ.

Plus précisément :

– Soit G est vraie avant l'exécution de j. Alors, i est exécuté et, par hypothèse, $\mathcal{POST}(i, G \wedge F)$ est vraie après l'exécution de j.

– Soit G est fausse avant l'exécution de j. Alors, aucune instruction n'est exécutée et $\neg G \wedge F$ est vraie après l'exécution de j.

Nous pouvons donc étendre la définition de $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(\texttt{ if } G \texttt{ then } i,\ F\ ) \equiv \mathcal{POST}(i, G \wedge F) \vee (\neg G \wedge F)}$$

Instruction "`forone`"

Prenons enfin une instruction $j \equiv$ "`forone` x `where` G `do` i" (x est une variable et G est une formule). Partant de j et d'une formule F, $\mathcal{POST}$ produit en fait une post-condition pour i, sachant que F est vraie et que G est satisfaite par un objet x. La précondition qui décrit l'état initial est donc : $\exists x : G \wedge F$. L'instruction i est alors exécutée. La post-condition qui décrit l'état final est donc $\mathcal{POST}(i, \exists x : G \wedge F)$, qui est équivalente à $\exists x : \mathcal{POST}(i, G \wedge F)$. Nous pouvons donc étendre la définition de $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(\texttt{ forone } x \texttt{ where } G \texttt{ do } i\ ,\ F\ ) \equiv \exists x : \mathcal{POST}(i, G \wedge F)}$$

Soulignons au passage un point intéressant : le transformateur de prédicat "absorbe" les erreurs d'exécution. Imaginons un instant qu'aucun objet x ne satisfasse la condition G. Survient alors une erreur d'exécution et la méthode est interrompue et annulée. Étant donné que cette méthode n'atteint pas d'état final, la formule qui est satisfaite par cet état final inexistant est : `faux` (`faux` n'admet pas de modèles). Pour prendre en compte l'éventualité d'une erreur d'exécution, on doit en toute rigueur définir $\mathcal{POST}$ comme suit :

$$\boxed{\mathcal{POST}(\texttt{ forone } x \texttt{ where } G \texttt{ do } i\ ,\ F\ ) \equiv \exists x : \mathcal{POST}(i, G \wedge F) \vee \texttt{faux}}$$

Qui est équivalente à notre définition de départ, étant donnée la propriété suivante de la disjonction : $F \vee \texttt{faux} \equiv F$.

5.3 Définition et correction du transformateur de prédicats avant

5.3.1 Définition

La section précédente avait pour but de montrer que $\mathcal{POST}$ n'est pas défini arbitrairement, mais qu'au contraire, il était, dans sa construction même, sémantiquement fondé. Nous récapitulons ci-dessous la définition de $\mathcal{POST}$. Nous avons en outre ajouté le cas où l'instruction est un bloc d'instructions : $\{i\}$. Nous rappelons que, pour l'instant, les méthodes que nous traitons n'ont pas de boucles et n'appellent pas d'autres méthodes. Nous traiterons le cas des boucles et des appels de méthodes plus loin, en section 5.5.

Définition 30 Soient i, i_1 et i_2 des instructions totalement décomposées, et F, F_1, F_2 et G des formules totalement décomposées. Le transformateur de prédicats avant $\mathcal{POST}$ est défini inductivement comme suit :

– **Si:** $i \equiv u.a{:=}v$ et $F \equiv x.a = y$
  Alors: $\mathcal{POST}(\ i\ ,\ F\ ) \equiv ((x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a = y))$

– **Si:** $i \equiv u.a{:=}v$ et $F \equiv x.a \neq y$
  Alors: $\mathcal{POST}(\ i\ ,\ F\ ) \equiv ((x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a \neq y))$

– **Si:** $i \equiv u.a{:=}v$ et F est un littéral différent de $(x.a = y)$ et $(x.a \neq y)$
  Alors: $\mathcal{POST}(\ i\ ,\ F\ ) \equiv (F \wedge u.a = v)$

– $\mathcal{POST}(i, F_1 \wedge F_2) \ \equiv\ \mathcal{POST}(i, F_1) \wedge \mathcal{POST}(i, F_2)$

– $\mathcal{POST}(i, F_1 \vee F_2) \ \equiv\ \mathcal{POST}(i, F_1) \vee \mathcal{POST}(i, F_2)$

– $\mathcal{POST}(i, \forall x : F) \ \equiv\ \forall x : \mathcal{POST}(i, F)$

– $\mathcal{POST}(i, \exists x : F) \ \equiv\ \exists x : \mathcal{POST}(i, F)$

– $\mathcal{POST}(\ i_1;i_2\ , F) \equiv \mathcal{POST}(i_2, \mathcal{POST}(i_1, F))$

– $\mathcal{POST}(\ \texttt{if}\ G\ \texttt{then}\ i,\ F\ ) \equiv \mathcal{POST}(i, G \wedge F) \vee (\neg G \wedge F)$

– $\mathcal{POST}(\ \texttt{forone}\ x\ \texttt{where}\ G\ \texttt{do}\ i\ ,\ F\ ) \equiv \exists x : \mathcal{POST}(i, G \wedge F)$

– $\mathcal{POST}(\ \{i\}\ ,\ F\ ) \equiv \mathcal{POST}(i, F)$

Soulignons le fait que $\mathcal{POST}$ est en fait naturellement défini *au niveau des littéraux*. Il ne modifie pas la structure supérieure de la formule en entrée. À titre d'exemple, considérons une formule définie à partir de quatre littéraux L_1, L_2, L_3 et L_4 :

$$\exists x : (L_1 \vee L_2) \wedge (L_3 \vee L_4)$$

La façon dont $\mathcal{POST}$ fonctionne, en laissant invariante la structure supérieure des formules, est illustrée en figure 5.1. Nous verrons plus loin que ce mode de fonctionnement n'est pas sans conséquence sur l'efficacité de $\mathcal{POST}$.

$$\mathcal{POST}\left(i,\ \begin{array}{cccccccc} & & & \multicolumn{2}{c}{\exists x :} & & & \\ & & & \multicolumn{2}{c}{\downarrow} & & & \\ & & & \multicolumn{2}{c}{\wedge} & & & \\ & & \swarrow & & & \searrow & & \\ & \multicolumn{2}{c}{\vee} & & & \multicolumn{2}{c}{\vee} & \\ \swarrow & & & \searrow & \swarrow & & & \searrow \\ L_1 & & & L_2 & L_3 & & & L_4 \end{array} \right) = \left(\begin{array}{cccccccc} & & & \multicolumn{2}{c}{\exists x :} & & & \\ & & & \multicolumn{2}{c}{\downarrow} & & & \\ & & & \multicolumn{2}{c}{\wedge} & & & \\ & & \swarrow & & & \searrow & & \\ & \multicolumn{2}{c}{\vee} & & & \multicolumn{2}{c}{\vee} & \\ \swarrow & & & \searrow & \swarrow & & & \searrow \\ \mathcal{POST}(i,L_1) & & & \mathcal{POST}(i,L_2) & \mathcal{POST}(i,L_3) & & & \mathcal{POST}(i,L_4) \end{array} \right)$$

FIG. 5.1 - *Exemple de décomposition d'une formule.*

5.3.2 Correction

Théorème 2 Soit i une instruction, F une formule, B une base de données, et B^+ la base de données résultante de l'exécution de i sur B.
Si pour une assignation α des variables libres de F, $B_\alpha \models F$, alors $B_\alpha^+ \models \mathcal{POST}(i, F)$.
Preuve : Par induction structurelle.

- Soient $i \equiv u.a{:=}v$ et $F \equiv x.a = y$. Alors :
  - Soit $B_\alpha \models (x = u)$, alors $B_\alpha^+ \models (x = u \wedge u.a = v)$.
  - Soit $B_\alpha \models (x \neq u)$, alors $B_\alpha^+ \models (x \neq u \wedge u.a = v \wedge x.a = y)$.

  Alors : $B_\alpha^+ \models (x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a = y)$
  Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$.
- Soient $i \equiv u.a{:=}v$ et $F \equiv x.a \neq y$. Alors :
  - Soit $B_\alpha \models (x = u)$, alors $B_\alpha^+ \models (x = u \wedge u.a = v)$.
  - Soit $B_\alpha \models (x \neq u)$, alors $B_\alpha^+ \models (x \neq u \wedge u.a = v \wedge x.a \neq y)$.

  Alors : $B_\alpha^+ \models (x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a \neq y)$
  Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$.
- Soient $i \equiv u.a{:=}v$ et F un littéral différent de $(x.a = y)$ et $(x.a \neq y)$.
  Alors : $B_\alpha^+ \models F \wedge u.a = v$
  Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$
- Soit $F \equiv F_1 \wedge F_2$.
  Par hypothèse : $B_\alpha \models F_1$ et $B_\alpha \models F_2$.
  Par hypothèse d'induction : $B_\alpha^+ \models \mathcal{POST}(i, F_1)$ et $B_\alpha^+ \models \mathcal{POST}(i, F_2)$.
  Alors : $B_\alpha^+ \models \mathcal{POST}(i, F_1) \wedge \mathcal{POST}(i, F_2)$.
  Et : $B_\alpha^+ \models \mathcal{POST}(i, F_1 \wedge F_2)$.
  Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$.
- Soit $F \equiv F_1 \vee F_2$.
  Par hypothèse : $B_\alpha \models F_1$ ou $B_\alpha \models F_2$.
  Par hypothèse d'induction : $B_\alpha^+ \models \mathcal{POST}(i, F_1)$ ou $B_\alpha^+ \models \mathcal{POST}(i, F_2)$.
  Alors : $B_\alpha^+ \models \mathcal{POST}(i, F_1) \vee \mathcal{POST}(i, F_2)$.
  Et : $B_\alpha^+ \models \mathcal{POST}(i, F_1 \vee F_2)$.
  Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$.
- Soit $F \equiv \forall x : G$.
  Alors : quelque soit x, $B_\alpha \models G$.
  Par hypothèse d'induction : quelque soit x, $\mathcal{B}_\alpha^+ \models \mathcal{POST}(i, G)$.
  Alors : $\mathcal{B}_\alpha^+ \models \forall x : \mathcal{POST}(i, G)$.
  Donc : $\mathcal{B}_\alpha^+ \models \mathcal{POST}(i, F)$.

– Soit $F \equiv \exists x : G$.
Alors : pour un x, $B_\alpha \models G$.
Par hypothèse d'induction : pour un x, $\mathcal{B}_\alpha^+ \models \mathcal{POST}(i, G)$.
Alors : $\mathcal{B}_\alpha^+ \models \exists x : \mathcal{POST}(i, G)$.
Donc : $\mathcal{B}_\alpha^+ \models \mathcal{POST}(i, F)$.

– Soient $i \equiv i_1;i_2$, et B^* la base de données résultant de l'exécution de i_1 sur B.
Par hypothèse d'induction : $B_\alpha^* \models \mathcal{POST}(i_1, F)$.
Par hypothèse d'induction : $B_\alpha^+ \models \mathcal{POST}(i_2, \mathcal{POST}(i_1, F))$.
Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$

– Soient $i \equiv$ "`if` G `then` j", et B^* la base de données résultant de l'exécution de j sur B. Alors :

  – Soit $B_\alpha \models G$, et j est exécuté.
  Alors : $B^+ = B^*$.
  Par hypothèse d'induction : $B_\alpha^* \models \mathcal{POST}(j, F)$
  Donc : $B_\alpha^+ \models \mathcal{POST}(j, F)$.

  – Soit $B_\alpha \not\models G$.
  Alors $B^+ = B$ (aucune instruction n'est exécutée).
  Donc : $B_\alpha^+ \models \neg G \wedge F$.

Alors : $B_\alpha^+ \models \mathcal{POST}(j, F) \vee (\neg G \wedge F)$.
Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$.

– Soit $i \equiv$ "`forone` x `where` G `do` j". Alors :

  – Soit il existe un x tel que $B_\alpha \models G$.
  Alors j est exécuté.
  Par hypothèse d'induction : il existe un x tel que $B_\alpha^+ \models \mathcal{POST}(j, F \wedge G)$.
  Donc : $B_\alpha^+ \models \exists x : \mathcal{POST}(j, F \wedge G)$

  – Sinon, une erreur d'exécution survient et B^+ n'existe pas.
  En d'autres termes : $B_\alpha^+ \models$ `faux`.

Alors : $B_\alpha^+ \models \exists x : \mathcal{POST}(j, F \wedge G) \vee \texttt{faux}$.
Qui se réécrit : $B_\alpha^+ \models \exists x : \mathcal{POST}(j, F \wedge G)$
Donc : $B_\alpha^+ \models \mathcal{POST}(i, F)$.

Dans tous les cas, on a bien : $B_\alpha^+ \models \mathcal{POST}(i, F)$.
C.Q.F.D.

Cette preuve respecte les étapes de construction du transformateur de prédicats. Insistons bien sur le fait que chaque étape est indissociable des autres (c'est le propre d'une définition par induction). En particulier, certaines définitions sont totalement déterminées par d'autres définitions. C'est la cas, par exemple, de la définition de $\mathcal{POST}$ lorsque la formule en entrée est $(\forall x : F)$ ou $(\exists x : F)$. En effet, si le domaine de la base de données est D, nous pouvons voir la quantification universelle comme une conjonction généralisée sur D, et la quantification existentielle comme une disjonction généralisée sur D.

- $\forall x : F(x) \;\equiv\; \bigwedge_{e \in D} F(e)$
- $\exists x : F(x) \;\equiv\; \bigvee_{e \in D} F(e)$

Muni de ces définitions, nous pouvons retrouver la définition de $\mathcal{POST}$ pour les formules universelles, en fonction de la définition donnée pour les formules conjonctives :

$$\begin{aligned}
\mathcal{POST}(i, \forall x : F(x)) &= \mathcal{POST}(i, \bigwedge_{e \in D} F(e)) \\
&= \bigwedge_{e \in D} \mathcal{POST}(i, F(e)) \\
&= \forall x : \mathcal{POST}(i, F(x))
\end{aligned}$$

Le cas de la quantification existentielle est similaire :

$$\begin{aligned}
\mathcal{POST}(i, \exists x : F(x)) &= \mathcal{POST}(i, \bigvee_{e \in D} F(e)) \\
&= \bigvee_{e \in D} \mathcal{POST}(i, F(e)) \\
&= \exists x : \mathcal{POST}(i, F(x))
\end{aligned}$$

5.3.3 Exemple de calcul

Prenons une contrainte et une méthode du chapitre 4 :

$$\texttt{C1: } \exists\ x \in \texttt{ LesPersonnes: } x\texttt{.epoux=nil;}$$

```
method separe() in class Personne {
            self.epoux:=nil;}
```

Nous donnons en Figure 5.2 le calcul effectué par le transformateur de prédicats $\mathcal{POST}$, pour la méthode `separe` de la classe `Personne`, avec comme pré-condition la contrainte `C1`. La méthode `separe` de la classe `Personne` est de taille réduite, ainsi que la contrainte `C1`. Ceci va nous permettre de traiter intégralement, et “à la main”, un cas de preuve de programme. Dans la suite, le symbole s est une abréviation pour le receveur de l'appel de méthode, c'est-à-dire `self`.

$$\begin{aligned}
\mathcal{POST}(\texttt{separe}, \texttt{C1}) \quad &\equiv \mathcal{POST}(\ s.\texttt{epoux} := \texttt{nil}\ ,\ \exists\, x : x \in \texttt{ LesPersonnes} \wedge x.\texttt{epoux} = \texttt{nil}) \\
&\equiv \exists\, x : \mathcal{POST}(\ s.\texttt{epoux} := \texttt{nil}\ ,\ x \in \texttt{ LesPersonnes} \wedge x.\texttt{epoux} = \texttt{nil}) \\
&\equiv \exists\, x : \mathcal{POST}(\ s.\texttt{epoux} := \texttt{nil}\ ,\ x \in \texttt{ LesPersonnes}) \wedge \mathcal{POST}(\ s.\texttt{epoux} := \texttt{nil}\ , x.\texttt{epoux} = \texttt{nil}) \\
&\equiv \exists x : (x \in \texttt{LesPersonnes} \wedge s.\texttt{epoux} = \texttt{nil}) \\
&\quad \wedge ((s = x \wedge s.\texttt{epoux} = \texttt{nil}) \vee (s \neq x \wedge s.\texttt{epoux} = \texttt{nil} \wedge x.\texttt{epoux} = \texttt{nil}))
\end{aligned}$$

FIG. 5.2 - *Calcul de* $\mathcal{POST}(\texttt{separe}, \texttt{C1})$

5.4 Application au problème de l'intégrité

Nous avons enfin tous les éléments nécessaires pour résoudre notre problème. Nous donnons dans cette section une condition *suffisante* pour qu'une méthode préserve une contrainte d'intégrité. Nous rappelons que nous ne nous intéressons pour l'instant qu'aux méthodes non-surchargées.

Théorème 3 Soient m une méthode et C une contrainte d'intégrité. m préserve C si

$$\mathcal{POST}(m, C) \Rightarrow C$$

Preuve : m préserve C si, lorsqu'elle est exécutée sur une base de données initiale B telle que $B \models C$, elle produit une base de données B^+ telle que $B^+ \models C$.
Par le théorème 2, on sait que, si $B \models C$, alors $B^+ \models \mathcal{POST}(m, C)$.
Comme, par hypothèse, $\mathcal{POST}(m, C) \Rightarrow C$, on a $B^+ \models C$. m ne peut donc jamais violer C.

Ce théorème ramène donc, à l'aide du transformateur de prédicats, notre problème à celui de la preuve de théorème en logique de premier ordre. Nous allons appliquer ce théorème pour prouver que la méthode `separe` de la classe `Personne` ne peut jamais violer la contrainte `C1`. D'après le théorème précédent, il faut prouver :

$$\mathcal{POST}(\texttt{separe}, \texttt{C1}) \Rightarrow \texttt{C1}$$

À cette fin, nous utiliserons la technique de démonstration automatique de théorèmes exposée dans le chapitre 2 : la méthode des tableaux avec égalité. Nous rappelons que cette méthode procède par *réfutation*. Elle prouvera donc que :

$$\mathcal{POST}(\texttt{separe}, \texttt{C1}) \wedge \neg\texttt{C1}$$

est insatisfaisable. Le tableau obtenu à partir de cette formule est donné en figure 5.3. Nous rappelons que le symbole s est une abréviation pour le receveur de l'appel de méthode, c'est-à-dire `self`.

Ce tableau est *fermé* pour la substitution $[c/u]$. En effet, pour cette substitution, chaque branche du tableau est contradictoire : **16** est contradictoire avec **8**, **19** avec **17**, **20** avec **8** et **21** avec **18**. La raison pour laquelle nous avons choisi un exemple simple apparaît clairement. Pour des raisons de place, il n'aurait pas été possible de traiter entièrement, et "à la main", un exemple plus complexe.

5.5 Boucles et appels de méthodes, méthodes surchargées

Nous montrons dans cette section comment gérer deux types d'instructions importantes : les boucles et les appels de méthodes. En outre, nous verrons le cas des méthodes *surchargées*.

5.5.1 Les boucles

Les bases de données emploient un type de boucle particulier, qui permet de sélectionner dans un premier temps un certain nombre d'objets satisfaisant une condition particulière, puis d'exécuter une instruction donnée pour chaque objet sélectionné. Cette instruction s'écrit :

`forall` *paramètre* `where` *condition* `do` *instruction*

1 $\boxed{\begin{array}{c}(\exists x : (x \in \texttt{LesPersonnes} \wedge s.\mathtt{epoux} = \mathtt{nil}) \wedge ((s = x \wedge s.\mathtt{epoux} = \mathtt{nil}) \vee (s \neq x \wedge s.\mathtt{epoux} = \mathtt{nil} \wedge x.\mathtt{epoux} = \mathtt{nil}))) \\ \wedge(\forall x : x \notin \texttt{LesPersonnes} \vee x.\mathtt{epoux} \neq \mathtt{nil})\end{array}}$

$\downarrow \alpha_1(\mathbf{1})$

2 $\boxed{\exists x : (x \in \texttt{LesPersonnes} \wedge s.\mathtt{epoux} = \mathtt{nil}) \wedge ((s = x \wedge s.\mathtt{epoux} = \mathtt{nil}) \vee (s \neq x \wedge s.\mathtt{epoux} = \mathtt{nil} \wedge x.\mathtt{epoux} = \mathtt{nil}))}$

$\downarrow \alpha_2(\mathbf{1})$

3 $\boxed{\forall x : x \notin \texttt{LesPersonnes} \vee x.\mathtt{epoux} \neq \mathtt{nil}}$

$\downarrow \delta(\mathbf{2})$

4 $\boxed{(c \in \texttt{LesPersonnes} \wedge s.\mathtt{epoux} = \mathtt{nil}) \wedge ((s = c \wedge s.\mathtt{epoux} = \mathtt{nil}) \vee (s \neq c \wedge s.\mathtt{epoux} = \mathtt{nil} \wedge c.\mathtt{epoux} = \mathtt{nil}))}$

$\downarrow \gamma(\mathbf{3})$

5 $\boxed{u \notin \texttt{LesPersonnes} \vee u.\mathtt{epoux} \neq \mathtt{nil}}$

$\downarrow \alpha_1(\mathbf{4})$

6 $\boxed{(c \in \texttt{LesPersonnes} \wedge s.\mathtt{epoux} = \mathtt{nil})}$

$\downarrow \alpha_2(\mathbf{4})$

7 $\boxed{(s = c \wedge s.\mathtt{epoux} = \mathtt{nil}) \vee (s \neq c \wedge s.\mathtt{epoux} = \mathtt{nil} \wedge c.\mathtt{epoux} = \mathtt{nil})}$

$\downarrow \alpha_1(\mathbf{6})$

8 $\boxed{c \in \texttt{LesPersonnes}}$

$\downarrow \alpha_2(\mathbf{6})$

9 $\boxed{s.\mathtt{epoux} = \mathtt{nil}}$

$\swarrow \beta_1(\mathbf{7})$ $\searrow \beta_2(\mathbf{7})$

10 $\boxed{s = c \wedge s.\mathtt{epoux} = \mathtt{nil}}$ **11** $\boxed{s \neq c \wedge s.\mathtt{epoux} = \mathtt{nil} \wedge c.\mathtt{epoux} = \mathtt{nil}}$

$\swarrow \alpha_1(\mathbf{10})$ $\searrow \alpha_1(\mathbf{11})$

12 $\boxed{s = c}$ **13** $\boxed{s \neq c}$

$\swarrow \alpha_2(\mathbf{10})$ $\searrow \alpha_2(\mathbf{11})$

14 $\boxed{s.\mathtt{epoux} = \mathtt{nil}}$ **15** $\boxed{s.\mathtt{epoux} = \mathtt{nil}}$

$\swarrow \beta_1(\mathbf{5})$ $\searrow \beta_2(\mathbf{5})$ $\downarrow \alpha_3(\mathbf{11})$

16 $\boxed{u \notin \texttt{LesPersonnes}}$ **17** $\boxed{u.\mathtt{epoux} \neq \mathtt{nil}}$ **18** $\boxed{c.\mathtt{epoux} = \mathtt{nil}}$

↓*égalité(***12**,**14***)* $\swarrow \beta_1(\mathbf{5})$ $\searrow \beta_2(\mathbf{5})$

19 $\boxed{c.\mathtt{epoux} = \mathtt{nil}}$ **20** $\boxed{u \notin \texttt{LesPersonnes}}$ **21** $\boxed{u.\mathtt{epoux} \neq \mathtt{nil}}$

FIG. 5.3 - *Tableau pour* $\mathcal{POST}(\mathtt{separe1}, \mathtt{C1}) \wedge \neg\mathtt{C1}$

La prise en compte des boucles dans la preuve de programme a toujours été un problème. Diverses solutions ont été proposées, que ce soit en logique de Hoare ou en interprétation abstraite. En logique de Hoare, cela se fait par le calcul d'un *invariant* de la boucle, c'est-à-dire d'une condition qui, si elle est satisfaite à l'entrée de la boucle, est préservée par celle-ci. En interprétation abstraite, la sémantique abstraite d'une boucle peut être obtenue à l'aide de techniques d'*approximation de point fixe*, assez sophistiquées. Il est important de noter que ces techniques ne peuvent être mises en œuvre que dans la mesure où le domaine abstrait a une structure simple.

À nouveau, nous prenons le parti d'utiliser une méthode simple, plutôt que complète. Celle-ci est en fait une adaptation assez naturelle de la sémantique de la boucle donnée en logique de Hoare. À l'opposé de ce qui a été fait jusqu'à présent, notre façon de calculer la sémantique est assez binaire : elle détermine si, oui ou non, la boucle préserve une contrainte donnée. Prenons une boucle particulière :

$$\texttt{forall}\ x\ \texttt{where}\ P(x)\ \texttt{do}\ j(x)$$

Cette instruction calcule dans un premier temps l'ensemble S des objets qui satisfont la condition $P(x)$, ou encore $\{x|P(x)\}$. Ensuite, pour chaque objet de cet ensemble, l'instruction $j(x)$ est exécutée. Un premier problème surgit : l'exécution d'une boucle n'est pas déterministe et peut produire, selon l'ordre dans lequel les différents objets sont pris, des résultats différents. De nombreuses hypothèses ont été émises, notamment dans [Qia90], concernant les précautions à prendre dans l'utilisation de ce type de boucle. Le modèle théorique d'exécution de ces boucles est fondé sur l'exécution en parallèle du corps de la boucle (c'est-à-dire, l'instruction j) sur chaque objet.

En ce qui nous concerne, nous préférons éviter de définir des hypothèses concernant l'exécution des boucles, qui seront difficiles à formaliser et à intégrer dans notre méthode. Bien que la boucle "bases de données" soit, par nature, non déterministe, nous pouvons lui définir une sémantique assez générale pour que celle-ci soit effectivement indépendante de tout ordre d'exécution. Cette sémantique est très simple et fondée sur le principe suivant :

Une boucle b préserve une contrainte C si son corps préserve C.

Cette règle est valide. Sa validité est prouvée ci-dessous :

Théorème 4 Soit i une instruction telle que $i \equiv \texttt{forall}\ x\ \texttt{where}\ P(x)\ \texttt{do}\ j(x)$. Si j préserve une contrainte C, alors i préserve aussi C.

Preuve : Soit B (resp.B^+) la base de données avant (resp. après) l'exécution de i. On définit une ensemble $S = \{x|B \models P(x)\}$. S est l'ensemble des objets de B sur lesquels le corps de la boucle va être exécuté. S étant fini, nous pouvons l'ordonner : $S = \{o_1, .., o_n\}$. Notez que cet ordre est quelconque. L'exécution de i est donc équivalente à l'exécution de la séquence d'instructions suivante :

$$\{i(o_1)\,;\,...\,;i(o_n)\}$$

Or cette instruction préserve C car, par hypothèse, c'est la composition d'instructions préservant C.

Nous avons indiqué comment prouver qu'une boucle préserve une contrainte, sans donner de sémantique à la boucle. La transformateur de prédicats pour la boucle peut être défini de la façon suivante :

Définition 31 Soit i une boucle, C une contrainte et F une formule. Nous définissons $\mathcal{POST}$ pour i et F comme suit.

- si i préserve C et $F \Rightarrow C$, alors $\mathcal{POST}(i, F) \equiv C$.
- sinon $\mathcal{POST}(i, F) \equiv$ `true`.

La contrainte C de la définition précédente sera justement celle dont nous souhaitons prouver la préservation. Notez qu'afin de prouver les implications, nous devrons faire appel à un démonstrateur automatique de théorèmes. Prenons par exemple la méthode et la contrainte de la figure 5.4. La méthode `debit` préserve la

```
method debit() in class Parent{
  if (self.argent-1>0)
    self.argent:=self.argent-1;
  forall u where u ∈ self.enfants
    if (u.argent-1>0)
      u.argent:=u.argent-1;}

C5: ∀ x in LesPersonnes: x.argent > 0;
```

FIG. 5.4 - *Exemple de méthode avec boucle.*

contrainte `C5`, et ceci peut être prouvé en prouvant que les deux instructions :

- `if (self.argent-1>0) self.argent:=self.argent-1;`
- `if (u.argent-1>0) u.argent:=u.argent-1;`

préservent `C5`. Notez que ces preuves seront en tous points identique à la preuve, que la méthode `debit` de la classe `Personne` préserve `C5`.

5.5.2 Appels de méthodes

En ce qui concerne les appels de méthodes, nous ne faisons qu'appliquer des techniques standard qui ont été définies dans les domaines de la logique de Hoare et de l'analyse statique. Nous nous contenterons donc de rappeler la façon de traiter les appels de méthodes, et notamment les appels récursifs. Les appels simples de méthodes (i.e. sans appels récursifs) sont aisés à traiter car de tels appels peuvent être vus comme des macro-instructions. Une méthode sans appels récursifs peut être traduite en une méthode sans appels du tout.

Par souci de clarté, nous présenterons notre approche d'une manière moins abstraite que ce qui est fait habituellement, en prenant un exemple "type" de méthode. Le lecteur intéressé par une description générale peut consulter [Sch86] et [Apt81].

Pour simplifier, nous considérons que toute méthode récursive peut être écrite de la façon suivante :

$\texttt{method}\ m$ (*parameters*) $\texttt{in class}\ \kappa$
{`if` *cas de base*
`then` i
`else` *appel récursif*}

où "*cas de base*" dénote le test pour arrêter la récursion, et "*appel récursif*" dénote un appel récursif à m, avec un objet qui peut être atteint à partir de `self` où tout autre paramètre de m.

Nous définissons le transformateur de prédicats de m comme suit :

$$\mathcal{POST}(m,F) \equiv \begin{cases} C & \textbf{si } F \Rightarrow C \textbf{ et } \mathcal{POST}(i,C) \Rightarrow C \\ \texttt{true} & \textbf{sinon}. \end{cases}$$

Le transformateur de prédicats ainsi défini est correct.

Théorème 5 Soient F une formule, B une base de données, m une méthode récursive, et B^+ la base obtenue après l'exécution de m. Alors :

$$B \models F \implies B^+ \models \mathcal{POST}(m,F)$$

.

Preuve :
Tout appel à m va résulter en une série d'exécutions de l'instruction i sur différents objets. Cet appel à m sera donc équivalent à l'exécution de la séquence d'instructions suivante :

$$i(parametres_1);...;i(parametres_n)$$

où $parametres_k$ dénote les paramètres de la k-ème exécution de i (nous considérons que `self` est en fait le premier paramètre, ce qui simplifie les notations).
Par hypothèse, nous avons soit :

- $F \Rightarrow C$ et $\mathcal{POST}(i,C) \Rightarrow C$. Dans ce cas, $\mathcal{POST}(m,C) \equiv C$. Supposons que $B \models F$. Comme $F \Rightarrow C$, nous avons $B \models C$. Nous définissons n bases de données comme suit :
  - $\mathcal{B}_1$ résulte de l'exécution de la séquence d'instructions "$i(parametres_1)$" sur B.
  - $\mathcal{B}_2$ résulte de l'exécution de la séquence d'instructions "$i(parametres_1);i(parametres_2)$" sur B.
  - ...
  - $\mathcal{B}_n$ résulte de l'exécution de la séquence d'instructions "$i(parametres_1);...;i(parametres_n)$" sur B.

  Nous avons $\mathcal{B}_n = B^+$, parce que la séquence d'instructions "$i(parametres_1);...;i(parametres_n)$" est équivalente à m. Et comme $\mathcal{POST}(i,C) \Rightarrow C$, nous avons : $\mathcal{B}_1 \models C$,...,$\mathcal{B}_n \models C$, et donc : $B^+ \models C$.
- Sinon, $\mathcal{POST}(m,C) \equiv \texttt{true}$, et trivialement, $B^+ \models \texttt{true}$.

5.5.3 Méthodes surchargées

Dans le cadre modèle de données orienté-objet, il importe de prendre en compte les méthodes dites *surchargées*, c'est-à-dire qui ont plusieurs implantations. Le choix de l'implantation à exécuter dépend de la classe d'appartenance de l'objet receveur, et est fait à l'exécution (il s'agit de résolution tardive).

Nous noterons $m = \{m_1, ..., m_n\}$ une méthode surchargée m, où m_1,...,m_n dénotent les différentes implantations de la méthode m. Les méthodes surchargées posent un problème dans la mesure où, à la compilation, nous ne savons pas précisément quel code va être exécuté. Ce problème est résolu très simplement de la façon suivante : prouver qu'une méthode préserve une contrainte, c'est prouver que chacune de ses implantations préserve cette contrainte.

Théorème 6 Soient $m = \{m_1, ..., m_n\}$ une méthode surchargée, et C une contrainte. m préserve C si, pour tout $i \in [1..n]$, m_i préserve C.

L'intérêt de notre mécanisme d'analyse statique est qu'il est *modulaire* : lors de la redéfinition d'une méthode, les preuves qui ont déjà été obtenues pour les autre implantations de cette méthode restent valides. Il suffit de prouver que la nouvelle implantation respecte aussi les contraintes. En outre, l'information de typage nous permettrait d'optimiser davantage la gestion de l'intégrité.

5.6 Remarques

Dans cette section nous mettons en valeur différents aspects, qualités ou défauts, du transformateur de prédicats. Une caractéristique essentielle du transformateur de prédicats est que c'est un *calcul*. Contrairement à la logique de Hoare, il ne fonctionne pas avec des règles de déduction, mais par une transformation purement syntaxique et déterminée d'une formule en entrée. Nous commencerons par étudier la façon dont le transformateur de prédicats résout l'ambiguïté entre les symboles qu'il manipule. Nous verrons ensuite les principaux défaut de $\mathcal{POST}$: sa myopie et sa tendance à se répéter. Nous ferons une courte étude du coût de notre approche, et montrerons ses liens avec l'interprétation abstraite.

5.6.1 Résolution au vol de l'aliasing

Nous illustrons dans cette section la manière dont $\mathcal{POST}$, implicitement, résout l'ambiguïté entre les différents symboles apparaissant dans les formules et les méthodes. Prenons une instruction d'affectation simple sur notre base de données du chapitre 4.

$$i \equiv\ p\texttt{.epoux:=}q$$

considérons à présent la formule F suivante :

$$F \equiv (u.\texttt{epoux} = v)$$

Comme nous l'avons vu dans la définition de $\mathcal{POST}$, pour peu que F soit satisfaite avant l'exécution de i, la formule suivante sera satisfaite après :

$$F' \equiv (p = u \wedge p.\texttt{epoux} = q) \vee (p \neq u \wedge p.\texttt{epoux} = q \wedge u.\texttt{epoux} = v)$$

On peut s'étonner de la taille de cette post-condition. En effet, nous avons vu au chapitre 3 une règle simple pour les affectations (l'axiome d'affectation, page 19), qui produit une pré-condition en fonction d'une post-condition, par application d'une simple substitution de termes :

$$P[t/x]\{x{:=}t\}P$$

La raison pour laquelle la formule F' est si compliquée est qu'elle ne peut être obtenue par simple substitution. Dans le système du chapitre 3, les variables dans les formules et les variables du programme étaient les mêmes, et une modification de la valeur d'une variable laissait invariantes les autres variables.

Au contraire, dans notre cadre, les paramètres d'un programme peuvent tous désigner le même objet, et il est bien évident qu'à chaque variable, nous n'assignons pas forcément un objet différent. Nous sommes donc dans une situation où tous les symboles désignant des objets (variables et paramètres) ne sont que des variables "muettes", sans aucune autre hypothèse. Pour évaluer l'effet d'une instruction sur une formule, il faudra donc avant tout lever l'ambiguïté entre les différentes variables. Ceci est fait naturellement en faisant des hypothèses :

– soit deux variables dénotent le même objet. On dit alors qu'ils sont des *alias*.

– soit deux variables dénotent des objets différents.

La complexité de la post-condition reflète donc la tache du transformateur de prédicats qui est double : résoudre dans un premier temps l'aliasing (c'est-à-dire déterminer l'ensemble des possibilités), puis donner la sémantique logique de l'instruction dans chaque cas.

La définition de $\mathcal{POST}$ pour l'affectation semble bien faire cela, mais seulement pour *un couple* de paramètres/variables. En fait, via la définition de $\mathcal{POST}$ pour la conjonction et la disjonction, c'est pour un ensemble quelconque de variables que cela fonctionne. Prenons par exemple la formule F suivante :

$$F \equiv (u.\texttt{epoux} = v \wedge x.\texttt{epoux} = y)$$

Nous avons

$$\begin{aligned}
&\mathcal{POST}(i, F) = \\
&\quad ((p = u \wedge p.\texttt{epoux} = q) \vee (p \neq u \wedge p.\texttt{epoux} = q \wedge u.\texttt{epoux} = v)) \wedge \\
&\quad ((p = x \wedge p.\texttt{epoux} = q) \vee (p \neq x \wedge p.\texttt{epoux} = q \wedge x.\texttt{epoux} = y))
\end{aligned}$$

L'ensemble des quatre possibilités $(p = u)$, $(p \neq u)$, $(p = x)$ et $(p \neq x)$ apparaît clairement dans la post-condition. Les différentes hypothèses à évaluer lors du calcul pourront être faites en parallèle, comme dans l'exemple précédent, ou successivement, dans le cas d'une séquence d'instructions.

5.6.2 Défauts choisis de $\mathcal{POST}$: myopie et bégaiement

La définition de $\mathcal{POST}$ est purement syntaxique, et du point de vue d'une éventuelle mise en œuvre, c'est une qualité. Par contre, $\mathcal{POST}$ a une fâcheuse tendance à ignorer totalement le contenu sémantique de la formule en entrée. Il a donc une certaine propension à se répéter et évalue des hypothèses qui sont parfois en contradiction flagrante avec le sens de la formule d'origine.

bégaiement

Pour illustrer la tendance de $\mathcal{POST}$ à répéter plusieurs fois la même chose, prenons une formule :

$$F \equiv (u \in v \wedge x \in y)$$

et toujours la même instruction :

$$i \equiv p.\mathtt{epoux} := q$$

Le calcul de $\mathcal{POST}$ pour i et F est donné ci-après :

$$\begin{aligned}
\mathcal{POST}(i, F) &= \mathcal{POST}(i, (u \in v \wedge x \in y)) \\
&= \mathcal{POST}(i, (u \in v)) \wedge \mathcal{POST}(i, (x \in y)) \\
&= ((u \in v) \wedge (p.\mathtt{epoux} = q)) \wedge ((x \in y) \wedge (p.\mathtt{epoux} = q))
\end{aligned}$$

Nous pouvons voir que le littéral $(p.\mathtt{epoux} = q)$ est dupliqué dans la formule, sans que cela soit vraiment nécessaire. Comme nous l'avons vu précédemment (en figure 5.1, page 66), $\mathcal{POST}$ agit au niveau des littéraux, et n'a pas une vision globale de son action sur la formule de départ. Certains littéraux auront donc de nombreuses occurrences redondantes dans la formule résultante.

myopie

Pour illustrer l'incapacité du transformateur de prédicats à comprendre la pré-condition, prenons par exemple la formule suivante :

$$F \equiv p.\mathtt{epoux} = q$$

Et l'instruction : $i \equiv p.\mathtt{epoux} := r$.
Le calcul de $\mathcal{POST}$ pour i et F est donné ci-après :

$$\mathcal{POST}(i, F) = ((p = p \wedge p.\mathtt{epoux} = r) \vee (p \neq p \wedge p.\mathtt{epoux} = r \wedge p.\mathtt{epoux} = q))$$

Nous remarquons que $\mathcal{POST}$ a, pour résoudre l'aliasing, considéré les hypothèses suivantes : $(p = p)$ et $(p \neq p)$. Or il est bien évident que si nous nous restreignons aux interprétations égalitaires de nos formules, $(p = p)$ est toujours vrai, et $(p \neq p)$ est toujours faux.

5.6.3 Coût de la méthode

En théorie, cette méthode peut être très coûteuse. Nous n'avons pas étudié à proprement parler la complexité (au sens théorique) du transformateur de prédicats. Nous avons au contraire préféré évaluer d'une façon expérimentale les performances de cette méthode : c'est ce qui sera fait dans le chapitre 6. Il est cependant aisé de voir qu'une application littérale du transformateur donné dans ce chapitre aboutit à un algorithme exponentiel. Par exemple, l'application de $\mathcal{POST}$, pour une instruction atomique et une formule comportant n atomes peut aboutir, dans le pire des cas, à une formule comportant $5 \times n$ atomes. Ce type de calcul peut être fait pour les autres définitions de $\mathcal{POST}$. Il faut toutefois remarquer que ceci ne nous permet pas d'évaluer précisément le coût global de notre méthode, mais seulement le coût de la première phase : le calcul de la post-condition. Nous ne pouvons évidement pas donner un coût pour la phase de démonstration automatique de théorèmes, la logique de premier ordre étant indécidable. En outre, à ce jour rien ne nous permet d'affirmer que notre méthode, appliquée à une classe particulière de formules, serait décidable. Nous avons déjà vu (au chapitre 3) qu'en matière de preuve de programmes, le fait de simplifier la logique sous-jacente d'un système de preuve n'apportait pas grand chose (Par exemple, en limitant les formules à `vrai` et `faux`, qui est la manière la plus drastique de restreindre la logique utilisée, nous pouvons toujours exprimer par une formule la terminaison d'un programme).

En l'état, il y a peu de chances que cette méthode puisse être appliquée concrètement. Cependant, nous montrons ci-dessous qu'il suffit de prendre en compte la sémantique des formules à un niveau très élémentaire pour améliorer considérablement la situation. L'exemple qui suit est une extension de l'exemple déjà donné en section 5.6.2 de ce chapitre. Considérons l'instruction et la formule suivantes :

$$i \equiv \{p.a{:=}r; p.a{:=}s\}$$

$$F \equiv p.a = q$$

Un application littérale et systématique de $\mathcal{POST}$ aboutit à :

$$\begin{array}{rcl}
\mathcal{POST}(i,F) & \equiv & \mathcal{POST}(\ \{p.a{:=}r; p.a{:=}s\}\ ,\ p.a{=}q\ ) \\
& \equiv & \mathcal{POST}(\ p.a{:=}s\ ,\ \mathcal{POST}(\ p.a{:=}r\ ,\ p.a = q\ )) \\
& \equiv & \mathcal{POST}(\ p.a{:=}s\ , (p = p \wedge p.a = r) \vee (p \neq p \wedge p.a = r \wedge p.a = q)) \\
& \equiv & ((p = p \wedge p.a = s) \\
& & \quad \wedge((p = p \wedge p.a = s) \vee (p \neq p \wedge p.a = s \wedge p.a = r))) \\
& & \vee((p \neq p \wedge p.a = s) \\
& & \quad \wedge((p = p \wedge p.a = s) \vee (p \neq p \wedge p.a = s \wedge p.a = r)) \\
& & \quad \wedge((p = p \wedge p.a = s) \vee (p \neq p \wedge p.a = s \wedge p.a = q)))
\end{array}$$

Cette formule peut être réduite en utilisant les règles suivantes :

- $(p \neq p) \Leftrightarrow \mathtt{faux}$
- $(\mathtt{faux} \wedge G) \Leftrightarrow \mathtt{faux}$
- $(\mathtt{vrai} \wedge G) \Leftrightarrow G$
- $(p = p) \Leftrightarrow \mathtt{vrai}$
- $(\mathtt{faux} \vee G) \Leftrightarrow G$
- $(G \wedge G) \Leftrightarrow G$

Par application de ces règles, nous obtenons :

$$\begin{array}{rcl}
\mathcal{POST}(i,F) & \equiv & ((\mathtt{vrai} \wedge p.a = s) \\
& & \quad \wedge((\mathtt{vrai} \wedge p.a = s) \vee (\mathtt{faux} \wedge p.a = s \vee p.a = r))) \\
& & \vee((\mathtt{faux} \wedge p.a = s)
\end{array}$$

$$
\begin{aligned}
& \wedge((\texttt{vrai} \wedge p.a = s) \vee (\texttt{faux} \wedge p.a = s \vee p.a = r)) \\
& \wedge((\texttt{vrai} \wedge p.a = s) \vee (\texttt{faux} \wedge p.a = s \vee p.a = q))) \\
\equiv \;\; & (p.a = s \wedge (p.a = s \vee \texttt{faux})) \\
& \vee(\texttt{faux} \wedge (p.a = s \vee \texttt{faux}) \wedge (p.a = s \vee \texttt{faux})) \\
\equiv \;\; & (p.a = s \wedge p.a = s) \vee \texttt{faux} \\
\equiv \;\; & p.a = s \wedge p.a = s \\
\equiv \;\; & p.a = s
\end{aligned}
$$

Il apparaît clairement au vu de cet exemple qu'une évaluation théorique du coût ne rendra pas compte des éventuelles optimisations qui peuvent être réalisées sur le transformateur de prédicats. Ce sont justement des optimisations de ce type qui vont permettre une réelle mise en œuvre de $\mathcal{POST}$.

5.6.4 Liens avec l'interprétation abstraite

Pourquoi avoir mentionné l'interprétation abstraite dans le chapitre 3, pour avoir ensuite utilisé le calcul de Dijkstra ? La raison principale est que l'ensemble des résultats du chapitre 5 ont été obtenus dans l'ignorance quasi complète des travaux en logique de Hoare et en calcul de Dijkstra, mais que par contre, la façon de procéder de l'interprétation abstraite a beaucoup influencé notre conception du transformateur de prédicats. Le but de l'interprétation abstraite est en effet de capturer partiellement la sémantique des programmes par des moyens totalement automatiques. Ceci nécessite de faire une description exhaustive de la manière de capturer de l'information, mais aussi de la façon de traiter cette information, toujours par des moyens automatiques. Alors que les travaux en logique de Hoare s'arrêtent souvent à des considérations théoriques, l'interprétation abstraite vise à l'intégration effective des méthodes formelles dans des systèmes réels.

En outre l'interprétation abstraite est un cadre bien plus général que la logique de Hoare ou le calcul de Dijkstra, car elle permet de formaliser proprement des analyses statiques spécifiques, dont le but n'est pas de capturer toute la sémantique des programmes, mais qui se focalisent sur un aspect particulier de celle-ci. Notre approche peut donc se formaliser en termes d'interprétation abstraite :

– Le treillis de valeurs abstraites considéré est celui des formules de logique du premier ordre (sur un certain langage).

– La relation d'ordre sur ce treillis est naturellement l'implication logique "$\Rightarrow$".

– La fonction de *concrétisation* γ (à ne pas confondre avec la règle γ de la méthode des tableaux) est la fonction qui, étant donné une formule, renvoie l'ensemble des modèles de cette formule (pour une assignation donnée des variables libres de cette formule).

Notre approche est donc bien une interprétation abstraite. On peut toutefois se demander si cette vision des choses n'est pas quelque peu abusive, étant donné la précision du transformateur de prédicats. L'intérêt de l'interprétation abstraite est justement de pouvoir capturer certains aspects sémantiques des programmes, en se limitant à certaines caractéristiques "abstraites", sans avoir à charrier une information complète.

Le fait est que notre approche, bien qu'elle semble très détaillée et assez peu abstraite, reste à un niveau de détail insuffisant pour prouver que certaines méthodes préservent certaines contraintes. En dehors de la complexité de notre approche, qui peut déjà nous empêcher d'arriver à une conclusion, c'est le contenu même de l'information obtenue qui peut poser problème.

Les exemples pour lesquels le manque de précision de $\mathcal{POST}$ se fait sentir sont des exemples assez complexes. Nous commencerons donc par illustrer les "oublis" de $\mathcal{POST}$ avec un exemple simple. Considérons la formule suivante :

$$F \equiv x.a = y \wedge x.a = z$$

Il est clair que, l'attribut a étant en fait une fonction unaire, une conséquence de cette formule est que $y = z$. Prenons maintenant une instruction d'affectation :

$$i \equiv p.a{:=}q$$

Le résultat de l'application de $\mathcal{POST}$ à i et F est donné ci-dessous :

$$\begin{array}{rcl}\mathcal{POST}(i,F) & \equiv & ((p = x \wedge p.a = q) \vee (p \neq x \wedge x.a = y \wedge p.a = q)) \\ & & \wedge((p = x \wedge p.a = q) \vee (p \neq x \wedge x.a = z \wedge p.a = q))\end{array}$$

Or, alors que l'instruction i ne peut pas remettre en cause la valeur de vérité de $y = z$, et que donc cette formule est toujours satisfaite, elle n'est pas une conséquence de $\mathcal{POST}(i, F)$. Le transformateur de prédicats a donc "oublié" la formule $y = z$, qui n'était pas explicitement mentionnée dans la formule initiale, mais n'en était qu'une conséquence. Ceci montre bien que le transformateur de prédicats ne capture pas totalement la sémantique des méthodes, et peut à ce titre être vu comme une interprétation abstraite. On peut ajouter que, si l'information capturée par $\mathcal{POST}$ semble très précise, cette précision n'est pas un luxe car elle est déjà insuffisante pour traiter certains cas.

Nous illustrons dans la suite un autre type d'oubli, qui est lui beaucoup plus difficile à détecter. Nous avons vu que le transformateur de prédicats résolvait les alternatives dans l'exécution d'une méthode en faisant des *hypothèses*. Dans l'exemple précédent, ces hypothèses étaient $p = x$ et $p \neq x$. Il est à noter que ces hypothèses sont faites au cours du processus d'analyse, uniquement lorsque leur besoin se fait sentir. Ces hypothèses ne concernent donc que le point de la méthode où elles sont introduites. Un problème surgit alors lorsque ces hypothèses sont contradictoires avec le fait que l'état initial de la base satisfait la contrainte. Le transformateur de prédicats va alors considérer des alternatives d'exécution de la méthode qui, ne partant pas d'un état initial cohérent, n'on pas lieu d'être considérées. Tout cela est un peu abstrait... voyons un exemple. Prenons une méthode et une contrainte :

C11 : $\forall x \in$ LesPersonnes : x.epoux.epoux $= x \vee x$.epoux $=$ nil

```
method divorce() in class Personne{
  if (self.epoux≠nil){
    self.epoux.epoux:=nil;
    self.epoux:=nil;}}
```

Après décomposition, cette contrainte et cette méthode deviennent :

$\texttt{C11} : \forall x \in \texttt{LesPersonnes} : \exists u : (x.\texttt{epoux} = u \wedge (u.\texttt{epoux} = x \vee u = \texttt{nil}))$

```
method divorce() in class Personne{
  if (self.epoux≠nil){
    forone v where self.epoux=v do v.epoux:=nil;
    self.epoux:=nil;}}
```

$\mathcal{POST}$ va, au cours de son calcul, considérer en autres l'alternative suivante :

$$\texttt{self} \neq x \text{ et } \texttt{self} \neq u \text{ et } u = v$$

Cette alternative est contradictoire avec le fait que la contrainte est satisfaite au début de l'exécution, car on a : $\texttt{self.epoux} = v$, $u = v$, et donc $\texttt{self.epoux.epoux} = x$, et la contrainte n'est pas satisfaite car $\texttt{self} \neq x$. En outre, cet ensemble d'hypothèses, décrivant un état qui ne satisfait pas la contrainte, va aboutir à un autre état qui ne satisfait pas la contrainte. Notre méthode va donc conclure à tort que la contrainte peut être violée.

Cet exemple est assez ardu et isoler le problème qui empêche d'obtenir une preuve est en général une tâche ingrate car elle nécessite d'analyser en détail les formules produites par le transformateur de prédicats, qui sont de taille importante, pour déterminer en quoi l'information qui a été collectée est insuffisante. Notons pour terminer que le problème mentionné précédemment est spécifique aux bases de données et à l'utilisation des contraintes d'intégrité. Il est lié à la difficile prise en compte de l'hypothèse de cohérence. Sa solution nécessite de reformuler entièrement le transformateur de prédicats. Celui-ci, au lieu d'émettre des hypothèses "au vol", doit dans un premier temps formuler, a priori, l'ensemble des hypothèses qui seront nécessaires au cour de l'analyse, ne retenir ensuite que celles qui sont cohérentes avec la contrainte initiale, pour enfin calculer, pour chaque hypothèse, la post-condition correspondante.

5.7 Extensions

Cette section à pour but de montrer que notre approche n'est pas confinée à un modèle de données particulier, mais qu'elle peut s'appliquer aussi à d'autres cadres. Dans un premier temps, nous resterons dans le cadre des systèmes de gestion de bases de données orientés objets et traiterons des différents aspects suivants :

– les affectations entre paramètres,

– la création d'objets,

– les valeurs structurées,

– les ensembles, les multi-ensembles, et les listes.

Étant donné le coût de la preuve de programme, sans avoir à prendre en compte les caractéristiques avancées des modèles orientés-objet, nous pouvons nous douter que ces extensions ont avant tout un intérêt théorique. Plutôt que de redéfinir en toute rigueur un transformateur de prédicats complet pour chaque extension, ce qui alourdirait considérablement la présentation faite précédemment, nous illustrerons les problèmes posés par ces ajouts et donnerons des pistes pour leur solution. Nous insisterons donc sur l'intuition, au détriment de la formalisation. Nous montrerons aussi comment adapter notre approche aux bases de données relationnelles et aborderons très succinctement le cas des bases de données déductives.

5.7.1 Affectations entre paramètres

Jusqu'à présent, nous nous sommes limités à l'analyse des instructions d'affectation de la forme :

paramètre.attribut:=*paramètre* (ou *constante*).

Ce type d'instruction permet uniquement de modifier les attributs des objets de la base de données. Nous allons maintenant montrer les problèmes posés par les assignations de la forme :

paramètre:=*paramètre* (ou *constante*).

Ce type d'instruction ne change pas l'attribut d'un objet, mais change l'assignation d'un paramètre à un objet. En présence de telles instructions, nous devons prendre en compte, en plus de l'état de la base de données, l'assignation des paramètres du programme à des objets de la base. Jusqu'à présent, nous avons évité ce problème en considérant que cette assignation était fixe et quelconque : nous en faisions donc abstraction. Ceci n'est plus possible en présence d'assignations entre paramètres.

Alors qu'avant, nous confondions l'assignation d'un terme avec ce terme, nous devrons maintenant les distinguer. Ceci sera fait à l'aide d'un nouveau symbole de fonction : σ. Si p est un paramètre d'une méthode, alors $\sigma(p)$ dénotera l'assignation de ce paramètre à un objet de la base. La prise en compte des assignations, et de leur évolution, aura un impact non négligeable :

– Une méthode ne sera plus simplement une fonction qui prend en entrée une base de donnée et produit une autre base de données, mais une fonction qui prend en entrée un couple (base de données, assignation) et produit un autre couple du même type.

– Afin de permettre au transformateur de prédicats de capturer les changements d'assignation, nous devrons aussi considérer des formules du type :

$$\sigma(p) = x$$

où p dénote un paramètre, et x dénote un objet de la base. Notez qu'une formule de la forme $\sigma(p) = \sigma(q)$ devra être décomposée selon un mécanisme similaire à celui déjà exposé au chapitre 4 (en l'occurrence, cela donnera : $\exists x\,:\,(\sigma(p) = x) \wedge (\sigma(q) = x)$). Sur le plan logique, un paramètre sera à la fois une constante du langage et un élément du domaine, d'une façon similaire à ce qui se passe dans les interprétations de Herbrand.

Le changement d'assignation de p sera noté comme suit :

$$\sigma(p){:=}o$$

où o est un objet qui est la nouvelle assignation de p. Un changement d'assignation $\sigma(p)\texttt{:=}\sigma(q)$ devra d'abord être décomposé (encore une fois, ce mécanisme est tout à fait similaire à celui décrit au chapitre 4. Nous l'illustrons plus loin). Prenons une instruction simple :

$$p\texttt{:=}q$$

Cette instruction assigne au paramètre p le même objet qu'à q. Nous pouvons donc traduire cette instruction comme suit :

$$\sigma(p)\texttt{:=}\sigma(q)$$

Qui, après décomposition, donne :

$$\texttt{forone}\ x\ \texttt{where}\ \sigma(q) = x\ \texttt{do}\ \sigma(p)\texttt{:=}x$$

Prenons maintenant une instruction d'affectation qui modifie l'attribut a d'un objet :

$$p.a\texttt{:=}q$$

Cette instruction affecte à l'attribut a de l'objet $\sigma(p)$ (et non plus p), l'objet $\sigma(q)$. Cette instruction se traduira donc de la façon suivante :

$$\begin{array}{l}\texttt{forone}\ x\ \texttt{where}\ \sigma(p) = x \\ \texttt{do}\ \ \texttt{forone}\ y\ \texttt{where}\ \sigma(q) = y \\ \qquad \texttt{do}\ x.a\texttt{:=}y\end{array}$$

où $x.a{:=}y$ dénote l'affectation avec les “vrais” objets, et plus seulement les paramètres utilisés pour les représenter. Il faut maintenant étendre $\mathcal{POST}$ pour capturer les modifications d'assignations.

Définition 32 Soient A un atome et une i une instruction telle que $i \equiv \sigma(p){:=}q$. Nous étendons $\mathcal{POST}$ de la façon suivante :

- Si $A \equiv \sigma(p) = r$, alors $\mathcal{POST}(i, A) \equiv \sigma(p) = q$
- Sinon $\mathcal{POST}(i, A) \equiv A \wedge \sigma(p) = q$

Nous ne démontrerons pas la correction de cette définition. La preuve est une simplification de la preuve de $\mathcal{POST}$ pour les formules atomiques et les instructions d'affectation. Cette simplification provient du fait qu'en ce qui concerne les paramètres, nous n'avons pas de situation d'aliasing : une modification de l'assignation d'un paramètre ne modifie en rien l'assignation des autres paramètres.

Pour conclure, nous pouvons remarquer qu'en l'absence de boucles, les affectations entre paramètres sont totalement inutiles. Celles-ci ne servent en général qu'à modifier une variable d'itération dans une boucle. Sans boucles, les affections entre paramètres peuvent toujours être évitées, car elles ne servent qu'à la création d'alias inutiles. En outre, la boucle “base de données”, que nous étudions plus loin, permet de spécifier une variable d'itération dont la modification à chaque itération est automatique, sans avoir à faire appel à une affectation entre paramètres. L'utilisation d'une autre variable d'itération est dangereuse car elle remet en cause l'indépendance de chaque itération.

5.7.2 La création d'objets

Jusqu'à présent, nous avons considéré que les méthodes ne faisaient que modifier les attributs des objets. Nous nous placions dans l'hypothèse où les objets existent déjà et n'ont pas à être créés. Cette hypothèse est évidement irréaliste et notre langage devra aussi comporter une instruction de création d'objets :

$$\texttt{creer}\ p$$

où p est un paramètre du programme qui référencera le nouvel objet créé. Notre approche s'applique bien aux méthodes avec création d'objets, pour peu que le sens logique de cette nouvelle opération soit clairement élucidé. La création d'objets recouvre différentes opérations de nature hétérogène qu'il convient de bien séparer :

- L'allocation d'espace mémoire : cette opération alloue une zone mémoire à un objet. Cette zone mémoire définit déjà un objet, avec une identité et une valeur indéfinie. Cependant, cet objet ne fait pas partie de la base de données. En fait, l'allocation d'espace mémoire pour l'objet est une opération strictement physique qui doit être ignorée sur le plan logique. Elle n'a aucun effet sur la base de données et toute condition qui est satisfaite avant une allocation mémoire sera évidement satisfaite après.
- L'initialisation de l'objet : cette opération à pour but d'initialiser les attributs de l'objet précédemment créé. On peut considérer que cette opération ne fait pas à proprement parler partie de la création d'objets, puisqu'elle est équivalente à une série d'affectations d'attributs.
- La persistance : celle-ci peut être faite de différentes façons :
  - Explicitement, en utilisant des commandes spécifiques :

    • `persistant` p • `temporaire` p

    La première commande a pour effet de faire persister l'objet dénoté par le paramètre p, alors que la seconde le rend temporaire.
  - Par atteignabilité : dans ce contexte, il n'y a pas de commande explicite pour rendre un objet persistant. Comme nous l'avons déjà dit dans le chapitre 4, celui-ci est persistant si et seulement si il peut être atteint à partir des racines de persistance. La persistance se fait donc indirectement, par modification des attributs des objets et des racines de persistance.

Les aspects physiques de la création d'objets pouvant être ignorés, et l'initialisation des objets étant déjà capturée par le transformateur de prédicats actuel, il nous reste à prendre en compte la persistance.

L'expressions même de "création d'objet" est trompeuse, et risque de nous induire en erreur si nous essayons de définir un transformateur de prédicats pour cette opération. Elle tend en effet à nous faire croire que le domaine de la base de données n'est pas stable mais dépend des objets qui en font partie. Ceci est totalement faux : en effet, nous avons vu que, d'un point de vue logique, le domaine de la base de données est :

$$D = \mathcal{O} \cup \mathbb{N}$$

c'est-à-dire l'ensemble de *tous* les identifiants d'objets (et pas seulement ceux des objets de la base), et les entiers (nous pouvons ajouter d'autres types de valeurs, comme les chaînes de caractères, etc. cela ne change rien). En outre un objet a bien été défini comme un couple :

(identificateur d'objet,valeur)

Sur le plan logique, un objet n'est donc pas un élément du domaine (à l'opposé de son identité), mais une *relation* entre éléments du domaine. La création d'objets ne change donc pas le domaine de la base de données, mais seulement les liens qui relient les éléments du domaine. Le transformateur de prédicats $\mathcal{POST}$ peut donc être étendu naturellement pour capturer la création d'objets :

– En ce qui concerne la persistance explicite, il faut simplement ajouter au langage logique considéré le prédicat unaire :

$$\texttt{persistant}(x)$$

la définition de $\mathcal{POST}$ est donc naturellement :

Définition 33 Soit A un atome et i une instruction telle que $i \equiv \texttt{persistant}(p)$, alors on étend $\mathcal{POST}$ de la façon suivante :

  – Si $A \equiv \neg\texttt{persistant}(x)$, alors $\mathcal{POST}(i, A) \equiv (x = p \wedge \texttt{persistant}(x)) \vee (x \neq p \wedge \neg\texttt{persistant}(x) \wedge \texttt{persistant}(p))$
  – Sinon $\mathcal{POST}(i, A) \equiv A \wedge \texttt{persistant}(p)$

Une extension similaire peut être définie pour $i \equiv \texttt{temporaire}\ p$. La preuve de correction de ces extensions est similaire à celles déjà données.

– Le cas de la persistance par attachement est encore plus simple car il suffit alors de prendre en compte les modifications des attributs et des racines de persistance, ce que $\mathcal{POST}$ fait déjà (nous voyons comment traiter les ensembles plus loin).

5.7.3 Les valeurs structurées

Une caractéristique fondamentale des systèmes orientés objet est d'autoriser le stockage de données structurées d'une manière complexe. La prise en compte par le transformateur de prédicats de ces valeurs structurées se fait bien moyennant, encore une fois, une bonne interprétation logique des opérations réalisées. La littérature concernant ces valeurs structurées les décrit comme étant “non partageables” et “non modifiables”. Cette expression signifie en fait que chaque objet se voit attribuer une et une seule zone mémoire pour stocker sa valeur et qu'il ne peut pas en changer.

La façon d'étendre $\mathcal{POST}$ sera entièrement illustrée à l'aide d'un exemple. Considérons une classe donnée, dont les objets font référence à une valeur structurée :

```
classe c1 :  [ a:[b:integer,c:integer],
               d:[e:integer,f:integer]]
```

Les attributs a et d sont structurés comme des n-uplets. Le cadre défini au chapitre 4 ne permettant pas de construire des valeurs structurées, nous devons la décomposer en trois classes :

```
classe c2 :  [ a:c3,d:c4]
classe c3 :  [ b:integer,c:integer]
classe c4 :  [ e:integer,f:integer]
```

Le choix d'une structure plutôt qu'une autre n'est pas anodin et a de profondes conséquences sur le sens des affectations. En effet, l'instruction :

$$o_1.\mathsf{a}{:=}o_2.\mathsf{a}$$

aura différents effets :

– Si o_1 et o_2 sont deux objets de la classe c_1 alors cette instruction a pour effet de recopier la valeur o_2.a à l'emplacement mémoire réservé au stockage de o_1.a.

– Si o_1 et o_2 sont deux objets de la classe c_2, alors cette instruction fait pointer o_1.a et o_2.a vers le même objet.

Une conséquence importante de cette différence sera que, dans le premier cas, une modification de o_1.a.b sera sans effet sur o_2.a.b, alors que dans le second cas o_1.a.b et o_2.a.b seront modifiés de la même façon car ils représentent la même zone mémoire.

Notons au passage que l'expression o_1.a.b est un abus de notation et que les différentes occurrences du point "." n'ont pas le même sens. En O_2C, une distinction est faite en utilisant des opérateurs différents : "*", qui permet d'obtenir la valeur attribuée à un objet, et ".", qui permet d'accéder aux différents champs de cette valeur (il y a aussi l'opérateur "->", qui est une combinaison des deux opérateurs précédents).

La "bonne" manière de capturer les modifications de valeurs structurées avec le transformateur de prédicats passe par une décomposition des assignations. En fait, l'instruction :

$$o_1.\mathsf{a}{:=}o_2.\mathsf{a}$$

où o_1 et o_2 sont des objets de la classe c_1, est une abréviation de la séquence d'instructions suivante :

$$o_1.\mathsf{a.b}{:=}o_2.\mathsf{a.b};$$
$$o_1.\mathsf{a.c}{:=}o_2.\mathsf{a.c};$$

Ce type de décomposition peut être fait systématiquement. Au total, la prise en compte des valeurs structurées se fera très bien, mais au prix d'une décomposition approfondie des instructions, qui va plus loin que la décomposition décrite au chapitre 4. Ces valeurs structurées vont aussi changer notre façon de décomposer les formules. En effet, une formule :

$$o_1.\mathsf{a} = x$$

devra être développée en :

$$o_1.\mathsf{a.b} = x.\mathsf{b} \land o_1.\mathsf{a.c} = x.\mathsf{c}$$

Un point important est que les chemins dans les valeurs structurées, par exemple "o_1.a.b", ne doivent pas être décomposés, à l'opposé des chemins qui relient des objets. En effet, une telle décomposition aboutirait par exemple dans le cas précédent une autre formule du type o_1.a $= x$, et nous tournerions en rond.

En bref, l'adaptation de $\mathcal{POST}$ pour les valeurs structurées, si elle est possible, ne se fera pas sans mal. En effet, l'utilisation de valeurs structurées permet d'abréger des notations qui, si elles sont décomposées, mènent à des instructions et des formules dont la taille est beaucoup plus importante que celle des instructions et formules d'origine. Il est donc peu probable qu'une méthode faisant une utilisation intensive des valeurs structurées puisse être analysée efficacement. Pour terminer, voyons la décomposition qui serait obtenue à partir d'une instruction qui affecte à un objet o_1 de la classe $\mathtt{c}_1$ la valeur d'un autre objet o_2. Cette instruction pourrait s'écrire : $(*o_1)$:=$(*o_2)$. Elle se traduit par les instructions suivantes :

forone x_1 where o_2.a.b $= x_1$
 do o_1.a.b:=x_1;
forone x_2 where o_2.a.c $= x_2$
 do o_1.a.c:=x_2;
forone x_3 where o_2.a.e $= x_3$
 do o_1.a.e:=x_3;
forone x_4 where o_2.a.f $= x_4$
 do o_1.a.f:=x_4;

Partant d'une instruction de taille très réduite, nous aboutissons à tout un programme ! Nous pouvons en outre remarquer que rien de ce qui est fait dans ce programme n'est inutile. Une fois cette décomposition faite, la définition d'un transformateur de prédicats adéquat est identique aux transformateurs pour instructions d'affectations et formules atomiques (avec comme seule nuance que les chemins dans les valeurs structurées ne sont pas décomposés).

5.7.4 Ensembles, listes et multi-ensembles

Le schéma de bases de données décrit au chapitre 4 comporte une racine de persistance :

racine LesPersonnes : { Personne }

La base de données est définie comme l'ensemble des objets qui sont accessibles à partir des racines de persistance. Or, en l'absence d'instructions d'insertion et de suppression d'éléments dans des ensembles, nous n'avons aucun moyen de créer effectivement une base de données. Nous ajoutons donc deux instructions :

- insert u in v
- delete u from v

La première instruction a pour but d'insérer un élément dénoté par u dans un ensemble, dénoté par v. Nous pouvons bien sur remplacer u et v par des termes quelconques, mais cela se fera à nouveau au prix d'une décomposition, à l'aide de l'instruction `forone`. $\mathcal{POST}$ peut être naturellement étendu pour prendre en compte les ensembles :

Définition 34 Soit A un atome et i une instruction ensembliste. Nous étendons $\mathcal{POST}$ de la façon suivante :

– Si $A \equiv (x \notin y)$ et $i \equiv$ `insert` u `in` v, alors :

$$\begin{aligned}\mathcal{POST}(i, A) \quad \equiv \quad & ((u = x \wedge v = y \wedge x \in y) \\ & \vee (v \neq y \wedge u \in v \wedge x \notin y) \\ & \vee (x \neq u \wedge u \in v \wedge x \notin y))\end{aligned}$$

– Si $A \equiv (x \in y)$ et $i \equiv$ `delete` u `from` v, alors :

$$\begin{aligned}\mathcal{POST}(i, A) \quad \equiv \quad & ((u = x \wedge v = y \wedge x \notin y) \\ & \vee (v \neq y \wedge u \notin v \wedge x \in y) \\ & \vee (x \neq u \wedge u \notin v \wedge x \in y))\end{aligned}$$

– Si $A \not\equiv x \notin y$ et $i \equiv$ `insert` u `in` v, alors : $\mathcal{POST}(i, A) \equiv A \wedge u \in v$

– Si $A \not\equiv x \in y$ et $i \equiv$ `delete` u `from` v, alors : $\mathcal{POST}(i, A) \equiv A \wedge u \notin v$

Cette extension de $\mathcal{POST}$ est correcte et la preuve de correction peut être trivialement calquée sur celles déjà données. D'un point de vue pratique, la prise en compte des ensembles va faire surgir de nouveaux problèmes. Nous illustrons ces problèmes par un exemple. Considérons une nouvelle racine de persistance :

`racine LesEpoux :` $\{$ `Personne` $\}$

Cette nouvelle racine de persistance a pour but de regrouper les époux des objets appartenant à l'ensemble `LesPersonnes`. Or, la seule définition de `LesEpoux` ne garantit en rien que l'époux de chaque objet de `LesPersonnes` appartienne à `LesEpoux`. Il faut donc compléter cette définition par une contrainte d'intégrité qui dit : *l'attribut* `epoux` *de chaque objet de* `LesPersonnes` *doit appartenir à* `LesEpoux`. Ceci se traduit par :

`C10`:$\forall x \in$ `LesPersonnes` : x.`epoux` $\in$ `LesEpoux`

Nous définissons en outre une méthode `mariage3` comme suit :

```
method mariage3(q:Personne) in class Personne{
  self.epoux:=q;
  insert q in LesEpoux;}
```

Cette méthode ne viole pas la contrainte `C10`, car elle insère q dans `LesEpoux`, en même temps qu'elle affecte q à l'attribut `epoux` de `self`. Or, notre approche, sous sa forme actuelle, sera incapable de prouver que `mariage3` préserve `C10`. Voyons d'où vient le problème :

Imaginons, hypothèse absurde, que LesEpoux=LesPersonnes, c'est-à-dire que LesEpoux soit un alias de LesPersonnes (cela revient à dire qu'ils désignent la même zone mémoire, ce qui est faux). Dans ce cas la méthode mariage3 peut effectivement violer la contrainte C10. En effet, l'instruction "insert q in LesEpoux" modifie en fait LesPersonnes, et comme rien ne garantit que (q.epoux∈LesEpoux), la contrainte C10 risque d'être violée. Notez que tout ce raisonnement est parti d'une hypothèse fausse : LesEpoux=LesPersonnes. C'est pourtant cette hypothèse qui empêche notre technique d'aboutir à une preuve.

Le problème de fond est que notre approche ne considère comme hypothèse de départ que la contrainte d'intégrité. Or, de nombreuses autres propriétés, dépendantes entres autres du modèle de données, sont satisfaites, et doivent à ce titre être prises en compte. Une de ces propriétés est par exemple : *Les racines de persistance désignent des zones mémoire distinctes*. Cela aura comme conséquence qu'une modification d'une racine de persistance laissera inchangées les autres racines. Dans l'exemple précédent, nous n'arrivions pas à prouver qu'en général :

$$\mathcal{POST}(\texttt{mariage3}, \texttt{C10}) \Rightarrow \texttt{C10}$$

Par contre, nous pouvons aisément prouver que :

$$(\texttt{LesPersonnes} \neq \texttt{LesEpoux}) \Rightarrow (\mathcal{POST}(\texttt{mariage3}, \texttt{C10}) \Rightarrow \texttt{C10})$$

De façon générale, des propriétés supplémentaires, imposées par le modèle de données, devront être prises en compte. Nous insistons sur le fait qu'une description axiomatique de ces différentes propriétés ne pourra être traitée efficacement. Il faudra donc transformer le démonstrateur automatique de théorème pour qu'il incorpore ces propriétés, de la même façon que la règle de substitution permettait d'incorporer les axiomes de l'égalité à la méthode des tableaux.

Nous ne mentionnerons les listes et multi-ensembles que pour faire prendre la mesure des difficultés rencontrées. Nous pouvons à nouveau étendre le transformateur de prédicats pour qu'il capture les opérations sur les listes et les multi-ensembles. Il est cependant très peu probable que de quelconques propriétés non triviales de méthodes manipulant de telles structures de données puissent être prouvées. En effet, si on examine les théories sous jacentes à ces structures, on se rend compte que les listes, par exemple, sont des structures ordonnées. Il faudra donc prendre en compte les axiomes régissant les relations d'ordre. En général, les structures de données complexes satisferont des conditions qui seront très difficiles à prendre en compte (sur le plan pratique, s'entend. En théorie, cela marche très bien !).

5.7.5 Application au modèle relationnel

Notre approche n'est nullement liée au modèle de données présenté dans le chapitre 4. Par contre, elle dépend étroitement du langage utilisé pour exprimer les contraintes : la logique du premier ordre. Nous pouvons donc définir une approche tout à fait similaire pour les bases de données relationnelles. La seule différence est que, au lieu d'être définie comme un ensemble d'objets reliés entre eux par des attributs, une base de données relationnelle est un ensemble de relations, chaque relation étant un ensemble de n-uplets. Cette différence va se retrouver à deux niveaux :

– Au niveau des contraintes d'intégrité, les formules atomiques seront de la forme : $R(\theta)$, où R dénote un symbole de relation, et θ un n-uplet.

– Au niveau du langage de programmation, les instructions d'affectation seront remplacées par des instructions d'insertion et de suppression de n-uplets :

- $\texttt{insert}\ \theta\ \texttt{in}\ R$
- $\texttt{delete}\ \theta\ \texttt{from}\ R$

À part ces deux changements, les formules et méthodes seront construites de la même façon. Nous pouvons nous contenter de redéfinir $\mathcal{POST}$ pour les formules atomiques et les instructions d'insertion et de suppression :

Définition 35 Soit A un atome et i une instruction relationnelle. Nous redéfinissons $\mathcal{POST}$ de la façon suivante :

– Si $A \equiv \neg R(\theta_1)$ et $i \equiv \texttt{insert}\ \theta_2\ \texttt{in}\ R$,
alors : $\mathcal{POST}(i, A) \equiv ((\theta_1 = \theta_2 \wedge R(\theta_2)) \vee (\theta_1 \neq \theta_2 \wedge \neg R(\theta_1) \wedge R(\theta_2)))$

– Si $A \equiv R(\theta_1)$ et $i \equiv \texttt{delete}\ \theta_2\ \texttt{from}\ R$,
alors : $\mathcal{POST}(i, A) \equiv ((\theta_1 = \theta_2 \wedge \neg R(\theta_2)) \vee (\theta_1 \neq \theta_2 \wedge R(\theta_1) \wedge \neg R(\theta_2)))$

– Si $A \not\equiv \neg R(\theta_1)$ et $i \equiv \texttt{insert}\ \theta_2\ \texttt{in}\ R$, alors : $\mathcal{POST}(i, A) \equiv A \wedge R(\theta_2)$

– Si $A \not\equiv R(\theta_1)$ et $i \equiv \texttt{delete}\ \theta_2\ \texttt{from}\ R$, alors : $\mathcal{POST}(i, A) \equiv A \wedge \neg R(\theta_2)$

5.7.6 Application au modèle déductif : quelques pistes

Concernant les bases de données déductives, il y aura deux problèmes à résoudre :

– D'un point de vue théorique, le point délicat sera celui de la définition d'un transformateur de prédicats pour les formules atomiques faisant référence à une relation intentionnelle. En effet, une insertion d'un n-uplet dans une relation pourra remettre en cause la valeur de vérité de bien d'autres n-uplets dans des relations intentionnelles. Une solution a été suggérée par [Law95], mais celle-ci procède par recopie et duplication du programme logique qui décrit la base de données. Elle est donc très coûteuse et, en outre, ne s'applique qu'à une classe restreinte de programmes logiques.

– D'un point de vue pratique, les règles déductives seront vues comme des axiomes supplémentaires. Leur prise en compte par le démonstrateur automatique de théorèmes va donc induire un coût supplémentaire important.

L'extension effective au modèle déductif a donc peu de chances d'aboutir à un méthode opérationnelle. En outre, même si les problèmes mentionnés précédemment sont résolus, cette méthode risque d'être incapable de prouver des propriétés intéressantes de méthodes faisant un usage intensif des aspects déductifs. En effet, le démonstrateur de théorèmes est capable de démontrer des propriété vraies dans tous les modèles d'une théorie donnée. Or, lorsqu'on a à faire à une base de données déductive, on ne s'intéresse pas à tous les modèles du programme logique, mais seulement à son plus petit modèle. Ceci ne pourra être pris en compte dans un formalisme de premier ordre.

L'approche "transformateur de prédicats" peut donc être étendue et généralisée. Des extensions de $\mathcal{POST}$ ont été suggérées pour de nombreuses constructions comme les valeurs structurées, les ensembles, les boucles, etc. Sur le plan pratique, il est clair que la prise en compte de caractéristiques sophistiquées des modèles de données évolués sera très coûteuse. Un langage de programmation pour bases de données dont les méthodes seraient l'objet de preuves de programmes devra donc proposer avant tout un certain nombre de constructions fondamentales, dont la sémantique puisse être aisément capturée.

5.8 Définition et utilisation d'un transformateur de prédicats arrière

L'approche que nous avons présenté jusqu'à présent était fondée sur l'utilisation d'un transformateur de prédicats *avant*, nommé $\mathcal{POST}$. À partir d'une pré-condition et d'un programme, $\mathcal{POST}$ calcule une post-condition qui sera nécessairement satisfaite par les données en sortie du programme, pour peu que la pré-condition soit satisfaite par les données en entrée. Nous pouvons aussi définir un transformateur de prédicats *arrière*, que nous nommerons $\mathcal{PRE}$. Celui-ci, à partir d'une post-condition et d'un programme, calcule une pré-condition suffisante sur les données en entrée du programme pour que les données en sortie satisfassent la post-condition. $\mathcal{PRE}$ peut aussi être utilisé pour prouver qu'une méthode préserve une contrainte. Habituellement, en logique de Hoare, c'est ce type d'approche qui est utilisé, plutôt que les transformateurs de prédicats avant. La raison est que les transformateurs de prédicats arrière produisent habituellement des formules plus simples et donc plus faciles à gérer que les transformateurs de prédicats avant. Nous étudierons au prochain chapitre les performances respectives des transformateurs de prédicats en *avant* et en *arrière*.

Nous avons défini un transformateur de prédicats arrière, ainsi qu'un nouveau critère pour prouver qu'une méthode préserve un contrainte. Étant donné que la plupart des remarques qui ont été faites sur le transformateur de prédicats avant s'appliquent aussi au transformateur de prédicats arrière, nous présenterons ce dernier beaucoup plus brièvement. Ce transformateur, nommé $\mathcal{PRE}$, est défini ci-dessous.

Définition 36 Soient i, i_1 et i_2 des instructions totalement décomposées, et F, F_1, F_2 et G des formules totalement décomposées. Le transformateur de prédicats arrière $\mathcal{PRE}$ est défini inductivement comme suit :

– **Si :** $i \equiv u.a{:=}v$ et $F \equiv x.a = y$
  Alors : $\mathcal{PRE}(\ i\ ,\ F\ ) \equiv ((x = u \wedge y = v) \vee (x \neq u \wedge x.a = y))$

– **Si :** $i \equiv u.a{:=}v$ et $F \equiv x.a \neq y$
  Alors : $\mathcal{PRE}(\ i\ ,\ F\ ) \equiv ((x = u \wedge y \neq v) \vee (x \neq u \wedge x.a \neq y))$

– **Si :** $i \equiv u.a{:=}v$ et F est un littéral différent de $(x.a = y)$ et $(x.a \neq y)$
  Alors : $\mathcal{PRE}(\ i\ ,\ F\ ) \equiv F$

– $\mathcal{PRE}(i, F_1 \wedge F_2)\ \equiv\ \mathcal{PRE}(i, F_1) \wedge \mathcal{PRE}(i, F_2)$

– $\mathcal{PRE}(i, F_1 \vee F_2)\ \equiv\ \mathcal{PRE}(i, F_1) \vee \mathcal{PRE}(i, F_2)$

- $\mathcal{PRE}(i, \forall x : F) \equiv \forall x : \mathcal{PRE}(i, F)$
- $\mathcal{PRE}(i, \exists x : F) \equiv \exists x : \mathcal{PRE}(i, F)$
- $\mathcal{PRE}(\ i_1;i_2\ , F) \equiv \mathcal{PRE}(i_1, \mathcal{PRE}(i_2, F))$
- $\mathcal{PRE}(\ \texttt{if}\ G\ \texttt{then}\ i,\ F\ ) \equiv (G \wedge \mathcal{PRE}(i, F)) \vee (\neg G \wedge F)$
- $\mathcal{PRE}(\ \texttt{forone}\ x\ \texttt{where}\ G\ \texttt{do}\ i\ ,\ F\ ) \equiv \exists x : G \wedge \mathcal{PRE}(i, F)$
- $\mathcal{PRE}(\ \{i\}\ ,\ F\ ) \equiv \mathcal{PRE}(i, F)$

Comme d'habitude, nous devons prouver que ce transformateur de prédicats est correct. Il y aura une différence notable entre la propriété de correction de $\mathcal{PRE}$ et celle de $\mathcal{POST}$. En effet, au lieu de prouver que $\mathcal{POST}$ génère une post-condition nécessaire, il faut prouver que $\mathcal{PRE}$ génère une pré-condition suffisante.

Théorème 7 Soit i une instruction, F une formule, B une base de données, et B^+ la base de données résultante de l'exécution de i sur B.
Si pour une assignation α des variables libres de F, $B_\alpha \models \mathcal{PRE}(i, F)$, alors $B^+_\alpha \models F$.

Preuve : Nous nous limiterons au premier cas, les autres cas étant très similaires, soit à ce premier cas, soit aux cas qui ont déjà été traités dans le théorème 2, page 67.

Soient $i \equiv u.a{:=}v$ et $F \equiv x.a = y$.
Alors : $\mathcal{PRE}(\ i\ ,\ F\ ) \equiv ((x = u \wedge u = v) \vee (x \neq u \wedge x.a = y))$.
Par hypothèse : $B_\alpha \models \mathcal{PRE}(\ i\ ,\ F\ )$.
On a donc :

- Soit $B_\alpha \models (x = u \wedge y = v)$, alors l'instruction i peut être réécrite $i \equiv x.a{:=}y$, et mène nécessairement à $(x.a = y)$.
  Donc si $B_\alpha \models (x = u \wedge y = v)$, alors $B^+_\alpha \models x.a = y$.
- Soit $B_\alpha \models (x \neq u \wedge x.a = y)$, alors, comme $(u \neq x)$, l'instruction i est sans effet sur l'attribut a de x, et $(x.a = y)$ est toujours vrai après l'exécution de i. Donc, si $B_\alpha \models (x \neq u \wedge x.a = y)$ alors $B^+_\alpha \models x.a = y$.

Au total, si $B_\alpha \models \mathcal{PRE}(i, F)$, alors $B^+_\alpha \models F$

Le résultat suivant donne un critère suffisant pour affirmer qu'une méthode préserve une contrainte, mais qui, à l'opposé du théorème 3 (page 70), utilise le transformateur de prédicats arrière $\mathcal{PRE}$.

Théorème 8 Soient m une méthode et C une contrainte d'intégrité. m préserve C si

$$C \Rightarrow \mathcal{PRE}(m, C)$$

Preuve : m préserve C si, lorsqu'elle est exécutée sur une base de données initiale B telle que $B \models C$, elle produit une base de données B^+ telle que $B^+ \models C$. Par hypothèse, si $B \models C$ alors $B \models \mathcal{PRE}(m, C)$. Avec le théorème 7, on sait que, si $B \models \mathcal{PRE}(m, C)$, alors $B^+ \models C$. m préserve donc C.

Le transformateur de prédicats arrière peut en outre servir à faire de la *correction* de méthodes. En effet, lorsqu'une méthode viole potentiellement une contrainte, nous pouvons utiliser l'information générée par $\mathcal{PRE}$ afin de modifier le code de cette méthode de telle manière à ce que la méthode modifiée préserve la contrainte. Cette modification est simple et consiste à inclure un test, au début de la méthode, qui garantisse que celle-ci mène à un état cohérent. Ce *pré-requis* peut être calculé avec le transformateur de prédicats arrière $\mathcal{PRE}$. Nous définissons donc, pour une méthode donnée, comment modifier cette méthode pour qu'elle préserve à coup sûr une contrainte donnée.

Définition 37 Soient m une méthode et C une contrainte d'intégrité. La version corrigée de m par rapport à C, notée m_C est définie comme :

$$\texttt{method}\ m_C\ \texttt{\{ if}\ \mathcal{PRE}(m, C)\ \texttt{then self}\ < -m\ \texttt{\}}$$

(pour simplifier, m et m_C n'ont pas de paramètres. Elles sont définies dans les mêmes classes).

Il reste à prouver que la version corrigée d'une méthode par rapport à une contrainte préserve effectivement cette contrainte.

Théorème 9 Soient m une méthode et C une contrainte d'intégrité. m_C préserve C.
Preuve : m_C préserve C si, pour une base de données quelconque telle que $B \models C$, le résultat de l'exécution de m_C sur B, noté B^+, est tel que $B^+ \models C$. Nous avons :

– Soit $B \models \mathcal{PRE}(m, C)$, dans ce cas, m est exécuté, et par le théorème 7, $B^+ \models C$.

– Soit $B \not\models \mathcal{PRE}(m, C)$, alors rien n'est exécuté et $B^+ = B$. Donc $B^+ \models C$.

Toutefois, rien ne permet de dire qu'il vaut mieux exécuter la version corrigée d'une méthode que la méthode d'origine. En effet, le pré-requis calculé par $\mathcal{PRE}$ risque fort d'avoir une taille importante (comme toutes les formules générées par les transformateurs de prédicats). Cette situation empirera lorsque, au lieu d'une contrainte, nous devrons considérer un ensemble de contraintes. Ce pré-requis va donc être très lourd à vérifier à l'exécution, et il est donc préférable de conserver la méthode d'origine et de vérifier la contrainte a posteriori. Afin de réduire la taille du pré-requis, la possibilité de mettre en œuvre des techniques de simplification logiques a été évoquée, notamment par [Law95]. Ces techniques ne sont cependant pas décrites.

5.9 Conclusion

Nous avons défini dans ce chapitre un *calcul*, permettant d'obtenir la sémantique logique des méthodes. Ce calcul est fondé sur la notion de transformateur de prédicats. Le transformateur de prédicats que nous avons défini en premier lieu, $\mathcal{POST}$, est une fonction qui prend comme paramètre une formule, appelée la pré-condition, et une méthode, et produit une autre formule, appelée la post-condition. Nous avons défini $\mathcal{POST}$ de telle façon à ce qu'il soit *correct*, c'est-à-dire : si la pré-condition est satisfaite avant l'exécution de la méthode, alors la post-condition sera *nécessairement* satisfaite après.

Le transformateur de prédicats $\mathcal{POST}$ n'est pas conçu pour capturer entièrement les effets potentiels qu'une méthode peut avoir sur une contrainte. Par contre, il est défini sous une forme claire, concise et modulaire, qui facilite grandement une éventuelle implantation. $\mathcal{POST}$ fonctionne d'une manière purement syntaxique. Cela nous offre de nombreuses possibilités d'optimisations qui devront être faites dans sa mise en œuvre.

L'application de $\mathcal{POST}$, pour prouver qu'une méthode préserve une contrainte, est naturelle. Il suffit de démontrer que la post-condition générée à partir de la contrainte et de la méthode implique cette contrainte. Un des grands avantages de cette approche est de pouvoir bénéficier des résultats et des techniques de la logique du premier ordre. En particulier, les techniques de démonstration automatique, comme la méthode des tableaux (présentée au chapitre 2), nous seront d'une grande utilité.

Nous avons aussi montré que cette approche pouvait être étendue pour prendre en compte des constructions plus sophistiquées comme les boucles, les appels de méthodes, les valeurs structurées...

Enfin, nous avons construit un transformateur de prédicats *arrière*, nommé $\mathcal{PRE}$, qui, au lieu de générer une post-condition nécessaire à partir d'une pré-condition, génère une pré-condition suffisante à partir d'une post-condition. À l'instar de $\mathcal{POST}$, $\mathcal{PRE}$ peut être utilisé pour prouver qu'une méthode préserve une contrainte. En outre, il permet d'envisager la correction des méthodes qui violent potentiellement les contraintes.

Il s'agit maintenant de savoir si, en plus d'être théoriquement fondées, les techniques que nous avons exposées dans ce chapitre peuvent être *effectivement* mises en œuvre dans un système réel. Ce sera le but du prochain chapitre.

Nous montrerons tout d'abord comment mettre en œuvre efficacement les transformateurs de prédicats. Nous décrirons ensuite le prototype qui met en œuvre ces techniques. Enfin, nous montrerons les résultats obtenus par le prototype sur les exemples du chapitre 4.

Chapitre 6

Mise en œuvre et implantation

Nous avons présenté, dans le chapitre précédent, les aspects théoriques de notre approche. L'implantation des transformateurs de prédicats $\mathcal{POST}$ et $\mathcal{PRE}$ est aisée, et peut se faire très rapidement dans un langage de prototypage rapide comme Lisp ou Prolog. Le problème est qu'une implantation naïve de notre approche risque d'être très lente et coûteuse, et ne fournira donc que des résultats très pauvres. La raison de ce coût élevé est le fonctionnement *purement syntaxique* des transformateurs de prédicats. Nous avons déjà présenté certains des défauts de notre approche: les transformateurs de prédicats sont myopes, c'est-à-dire qu'ils ignorent le sens des formules qu'ils manipulent et évaluent des hypothèses qui sont en contradiction flagrante avec elles. En outre, ils bégaient, c'est-à-dire que la même information est dupliquée maintes et maintes fois, inutilement. Nous allons voir dans ce chapitre comment améliorer le fonctionnement des transformateurs de prédicats. Cette mise en œuvre sera avant tout décrite pour le transformateur de prédicats avant $\mathcal{POST}$. La mise en œuvre du transformateur de prédicats arrière $\mathcal{PRE}$ est tout à fait similaire.

- Dans un premier temps, en section 6.1, nous définissons un certain type de formules, pour lesquelles une sémantique peut aisément être définie. Ces formules sont en fait des expressions booléennes (c'est-à-dire sans quantificateurs) conjonctives.
- Ensuite, en section 6.2, nous définissons un nouveau transformateur de prédicats avant $\mathcal{POST}^*$ pour ces formules. L'intérêt de ce transformateur de prédicats est qu'il traite ces formules de façon globale, et non pas seulement au niveau des littéraux, comme cela était fait au chapitre précédent.
- Nous étudierons en section 6.3 les différents ordres d'évaluation de $\mathcal{POST}^*$. Ceci justifiera l'ordre particulier qui a été retenu dans l'implantation finale des prototypes.
- Enfin, nous décrirons succinctement ces prototypes et donnerons des mesures qui ont été faites sur les exemples du chapitre 4. Ceci nous permettra de faire une comparaison pratique des performances respectives des transformateurs de prédicats avant et arrière.

Insistons d'emblée sur le fait que ce travail d'implantation n'avait pas pour but d'aboutir à un gestionnaire de contraintes d'intégrité industriel. La seule finalité de ce prototype est d'établir *la faisabilité* de notre approche. En outre, les résultats que nous obtenons — et c'est souvent le cas en informatique — ne se prêtent pas à une interprétation binaire du type: “ça marche” ou “ça marche pas”. Ce chapitre a donc pour but de bien fixer les idée sur ce que l'on peut attendre d'une telle méthode (et de toute autre méthode du même acabit), sans donner de faux espoirs. Toutefois, en anticipant un peu, nous pouvons dire que cette méthode remplit assez bien son rôle.

6.1 Mieux capturer la sémantique des formules: les environnements

En dehors du fait qu'elles sont décomposées, la structure des formules est quelconque. Bien évidement, leur sémantique est aussi quelconque. Nous définissons dans cette section une catégorie spécifique de formules, pour lesquelles un transformateur de prédicats efficace peut être défini.

Définition 38 Un environnement est une formule totalement décomposée de la forme:

$$A_1 \wedge ... \wedge A_n$$

où A_1,...,A_n sont des atomes.

Un environnement est simplement une conjonction de faits atomiques. Voici un exemple d'environnement:

$$x.\texttt{epoux} = y \wedge y.\texttt{epoux} = x \wedge p \neq x$$

Étant donnée la commutativité de la conjonction, nous pouvons voir un environnement $A_1 \wedge ... \wedge A_n$ comme *un ensemble* de faits atomiques: $\{A_1, ..., A_n\}$.

Alors qu'en règle générale, il est difficile de dire si une formule est conséquence d'une autre (sauf à mettre en œuvre des techniques de démonstration automatique), il est aisé de dire si un formule atomique est conséquence d'un environnement. Prenons par exemple l'environnement:

$$\{x.\texttt{epoux} = y, p = x\}$$

Nous pouvons évidement déduire de cet environnement que $p.\texttt{epoux} = y$. L'ensemble des faits qui peuvent être déduits d'un environnement n'est pas toujours fini. Prenons par exemple l'environnement: $\{x.\texttt{epoux} = x\}$. De cet environnement nous pouvons déduire que $x.\texttt{epoux.epoux} = x$, $x.\texttt{epoux.epoux.epoux} = x$, etc. Nous donnons ci-après un moyen de déduire un nombre fini de faits (ou d'atomes) d'un environnement donné. Ceci nous permettra d'éclaircir partiellement la sémantique des formules que le transformateur de prédicats prend en paramètre, et de définir en conséquence un transformateur plus efficace.

Définition 39 Étant donné un environnement E, on définit un nouvel environnement, noté $\Delta(E)$, comme suit:

$$\begin{array}{rcl} \Delta(E) & = & \{A(t_2) | t_1 = t_2 \in E, A(t_1) \in E, A(t_2) \text{ est totalement décomposée}\} \\ & & \cup \{A(t_1) | t_1 = t_2 \in E, A(t_2) \in E, A(t_1) \text{ est totalement décomposée}\} \end{array}$$

Δ définit en fait un ensemble de faits que l'on peut déduire d'un environnement. Par exemple:

- Pour $E = \{x.\texttt{epoux} = y, p = x\}$, nous avons $\Delta(E) = \{p.\texttt{epoux} = y\}$.

– Pour $E = \{x.\texttt{epoux} = x,\}$, nous avons $\Delta(E) = \emptyset$, car aucun des atomes $x.\texttt{epoux.epoux} = x$, $x.\texttt{epoux.epoux.epoux} = x$, etc, n'est totalement décomposé.

Δ ne fait qu'appliquer la très classique *règle de substitution*, qui existe dans tous les systèmes de preuves en logique égalitaire. Celle-ci a déjà été décrite en page 14 du chapitre 2. Il est bien évident que $E \Rightarrow \Delta(E)$.

Définition 40 Soit E un environnement. Nous définissons une suite d'environnements E_i, pour $i \in \mathbb{N}$, de la façon suivante:

– $E_0 \equiv E$

– $E_{i+1} \equiv E_i \cup \Delta(E_i)$

Aussi, nous définissons l'environnement $\overline{E}$ comme suit:

$$\overline{E} \equiv \bigcup_{i \in \mathbb{N}} E_i$$

$\overline{E}$ décrit un ensemble de conséquences de E. Une caractéristique de cet environnement est qu'il est *fini*.

Propriété 1 Soit E un environnement. $\overline{E}$ est fini.
Preuve: tous les faits dans $\overline{E}$ sont entièrement décomposés, et construits à partir des constantes et variables de E, qui sont en nombre fini. Nous avons aussi un nombre fini de symboles de relations. Il n'y a donc qu'un nombre fini d'environnements possibles.

Voyons maintenant l'application de $\overline{E}$ sur un exemple plus complexe. Considérons l'environnement suivant:

$$E \equiv \{x_1 = x_2, x_1.\texttt{epoux} = y_1, x_2.\texttt{epoux} = y_2, y_1.\texttt{epoux} = z_1, y_2.\texttt{epoux} = z_2\}$$

Nous avons:

$$\Delta(E) \equiv \{x_1 = x_1, x_2 = x_2, x_2.\texttt{epoux} = y_1, x_1.\texttt{epoux} = y_2\}$$

Et

$$E_1 \equiv \left\{ \begin{array}{c} x_1 = x_2, x_1.\texttt{epoux} = y_1, x_2.\texttt{epoux} = y_2 \\ y_1.\texttt{epoux} = z_1, y_2.\texttt{epoux} = z_2, x_1 = x_1 \\ x_2 = x_2, x_2.\texttt{epoux} = y_1, x_1.\texttt{epoux} = y_2 \end{array} \right\}$$

Ensuite:

$$\Delta(E_1) \equiv \{y_1 = y_2, y_2 = y_1\}$$

En continuant le calcul, nous obtenons

$$\overline{E} \equiv \left\{ \begin{array}{c} x_1 = x_2, x_1.\texttt{epoux} = y_1, x_2.\texttt{epoux} = y_2, y_1.\texttt{epoux} = z_1, y_2.\texttt{epoux} = z_2, x_1 = x_1 \\ x_2 = x_2, x_2.\texttt{epoux} = y_1, x_1.\texttt{epoux} = y_2, y_1 = y_2, y_2 = y_1, y_1 = y_1 \\ y_2 = y_2, y_2.\texttt{epoux} = z_1, y_1.\texttt{epoux} = z_2, z_2 = z_1, z_1 = z_2, z_1 = z_1, z_2 = z_2 \end{array} \right\}$$

Le calcul des E_i aboutit en fait à un point fixe (le plus petit point fixe de la fonction Δ partant de E). Nous n'insisterons pas plus sur les aspects théoriques de Δ. Ce qui compte ici, c'est avant tout l'application que nous faisons de cette fonction. Cette fonction va nous servir à résorber partiellement la myopie du transformateur de prédicats $\mathcal{POST}$. Notons au passage une relation évidente, mais importante: $\overline{E} \Leftrightarrow E$.

6.2 Amélioration du transformateur de prédicats

Nous définissons un nouveau transformateur de prédicats, $\mathcal{POST}^*$, qui a pour fonction de mieux prendre en compte la sémantique des formules qu'il manipule. À l'opposé de $\mathcal{POST}$ qui ne "voit" qu'un seul atome à la fois, $\mathcal{POST}^*$ peut traiter globalement des environnements. Ceci est particulièrement important en ce qui concerne l'instruction d'affectation, ainsi que pour l'instruction "`forone`".

6.2.1 Définition de $\mathcal{POST}^*$ pour l'affectation

Définition 41 Soient i une instruction d'affectation telle que $i \equiv u.a{:=}v$, A un atome, et E un environnement tel que $A \in E$. La fonction **impact** est définie comme suit:

– Si $A \equiv x.a = y$, alors:

  – Si $(u = x) \in \overline{E}$, alors $\mathbf{impact}(i, A, E) \equiv \mathtt{vrai}$
  – Si $(u \neq x) \in \overline{E}$, alors $\mathbf{impact}(i, A, E) \equiv (x.a = y)$
  – Sinon $\mathbf{impact}(i, A, E) \equiv (u = x) \vee (u \neq x \wedge x.a = y)$

– Si $A \equiv x.a \neq y$, alors:

  – Si $(u = x) \in \overline{E}$, alors $\mathbf{impact}(i, A, E) \equiv \mathtt{vrai}$
  – Si $(u \neq x) \in \overline{E}$, alors $\mathbf{impact}(i, A, E) \equiv (x.a \neq y)$
  – Sinon $\mathbf{impact}(i, A, E) \equiv (u = x) \vee (u \neq x \wedge x.a \neq y)$

– Sinon $\mathbf{impact}(i, A, E) \equiv A$.

Intuitivement, la fonction **impact** a pour but de calculer l'impact d'une instruction i sur un atome A d'un environnement donné. Il y a seulement deux cas possibles: soit A est préservé, soit il est remis en cause. Outre l'impact de i sur A, il faut prendre en compte le fait nouveau apporté par i (c'est-à-dire : $u.a{:=}v$). C'est ce qui est fait dans la suite par le transformateur de prédicats $\mathcal{POST}^*$

Définition 42 Soit E un environnement, et i une instruction d'affectation $i \equiv u.a{:=}v$. Nous définissons $\mathcal{POST}^*$ comme suit:

$$\mathcal{POST}^*(i, E) \equiv \bigwedge_{A \in E} \mathbf{impact}(i, A, E) \wedge (u.a = v)$$

Nous devons maintenant prouver que la définition de $\mathcal{POST}^*$ est correcte. Commençons par un lemme:

Lemme 1 Soient i une instruction d'affectation telle que $i \equiv u.a{:=}v$, A un atome, et E un environnement tel que $A \in E$. Soient B une base de données, B^+ la base de données résultante de l'exécution de i sur B et α une assignation pour les variables libres de E. Alors:

$$B_\alpha \models E \Rightarrow B^+_\alpha \models \mathbf{impact}(i, A, E)$$

Preuve:

– Si $A \equiv x.a = y$, alors:

  – Si $(u = x) \in \overline{E}$, alors, trivialement, $B_\alpha^+ \models \texttt{vrai}$ (c'est toujours le cas).
  Donc $B_\alpha^+ \models \mathbf{impact}(i, A, E)$

  – Si $(u \neq x) \in \overline{E}$, alors $B_\alpha \models (u \neq x)$.
  Aussi: $B_\alpha^+ \models (x.a = y)$.
  Donc: $B_\alpha^+ \models \mathbf{impact}(i, A, E)$.

  – Sinon:

    – Soit $B_\alpha \models (u = x)$, et $B_\alpha^+ \models (u = x)$.
    – Soit $B_\alpha \models (u \neq x)$, alors $B_\alpha^+ \models (u \neq x \wedge x.a = y)$.

  Au total: $B_\alpha^+ \models ((u = x) \vee (u \neq x \wedge x.a = y))$.
  Donc: $B_\alpha^+ \models \mathbf{impact}(i, A, E)$.

– Pour $A \equiv x.a \neq y$, la preuve est tout à fait similaire.

– Sinon, l'attribut a ne paraît pas dans A et l'instruction i ne peut pas remettre en cause A. On a alors $B_\alpha^+ \models A$, et donc $B_\alpha^+ \models \mathbf{impact}(i, A, E)$.

Théorème 10 Soient i une instruction d'assignation telle que $i \equiv u.a{:=}v$, et E un environnement. Soient B une base de données, B^+ la base de données résultante de l'exécution de i sur B et α une assignation pour les variables libres de E. Alors:

$$(B_\alpha \models E) \Rightarrow (B_\alpha^+ \models \mathcal{POST}^*(i, E))$$

Preuve: Par hypothèse, $B_\alpha \models E$.
D'après le lemme 1, Pour tout atome $A \in E$, on a: $B_\alpha^+ \models \mathbf{impact}(i, A, E)$
Donc: $B_\alpha^+ \models \bigwedge_{A \in E} \mathbf{impact}(i, A, E)$
En outre: $B_\alpha^+ \models (u.a = v)$.
Alors: $B_\alpha^+ \models \bigwedge_{A \in E} \mathbf{impact}(i, A, E) \wedge (u.a = v)$.
Donc: $B_\alpha^+ \models \mathcal{POST}^*(i, E)$.

Nous avons donc un nouveau moyen fiable de capturer la sémantique de l'affectation. Nous allons maintenant montrer, à l'aide d'un exemple, que le transformateur de prédicats est moins myope qu'avant, et qu'il introduit, au cours de son calcul, beaucoup moins de redondances. Insistons sur la difficulté d'évaluer précisément le gain obtenu. En effet, la sémantique d'une formule est sans rapport avec sa taille. Or, les optimisations mises en œuvre ici partent de considérations sémantiques.

Soient l'instruction $i \equiv u.a{:=}v$, et l'environnement E suivant:

$$E \equiv (x \neq u \wedge x.a = y \wedge y \in z)$$

Comparons les résultats obtenus avec $\mathcal{POST}$ et $\mathcal{POST}^*$. Avec la version donnée dans le chapitre précédent nous obtenons:

$$\begin{aligned}
\mathcal{POST}(i, E) &= \mathcal{POST}(i, (x \neq u \wedge x.a = y \wedge y \in z)) \\
&= \mathcal{POST}(i, x \neq u) \wedge \mathcal{POST}(i, x.a = y) \wedge \mathcal{POST}(i, y \in z) \\
&= (x \neq u \wedge u.a = v) \\
&\quad \wedge ((x = u \wedge u.a = v) \vee (x \neq u \wedge u.a = v \wedge x.a = y)) \\
&\quad \wedge (y \in z \wedge u.a = v)
\end{aligned}$$

Notons que l'atome $u.a = v$ apparaît en quatre exemplaires, alors qu'un seul aurait suffit, et que le transformateur de prédicats a considéré l'hypothèse $(x = u)$, qui est contradictoire avec l'environnement de départ. Voyons ce qui se passe avec l'autre version de $\mathcal{POST}$:

$$\begin{aligned}
\mathcal{POST}^*(i, E) &= \mathcal{POST}^*(i, (x \neq u \wedge x.a = y \wedge y \in z)) \\
&= \mathbf{impact}(i, x \neq u, E) \wedge \mathbf{impact}(i, x.a = y, E) \\
&\quad \wedge \mathbf{impact}(i, y \in z, E) \wedge (u.a = v) \\
&= (x \neq u) \wedge (x.a = y) \wedge (y \in z) \wedge (u.a = v)
\end{aligned}$$

Cette version du calcul est clairement plus courte, tout en étant équivalente. Pourquoi mettre en œuvre un transformateur de prédicats différent, puisque, moyennant des simplifications logiques, le résultat fourni par la version première de $\mathcal{POST}$ pouvait être amélioré? Le problème de ces simplifications logiques est qu'elles sont extrêmement coûteuses, et qu'en outre, elles sont effectuées sur la post-condition, qui est en général assez lourde. En particulier, la mise sous forme normale disjonctive, qui est indispensable pour simplifier une formule, a un coût prohibitif. Elle ne peut donc être appliquée qu'à des formules simples ou de taille réduite. Les éventuelles optimisations qui sont mises en œuvre ne doivent pas être faites a posteriori (voire "post-mortem"), mais au vol, c'est-à-dire au cours même du calcul.

Prenons un exemple un peu plus abstrait, mais tout aussi convaincant. Considérons l'environnement suivant:

$$E \equiv \bigwedge_{i \in [1..10]} (u = x_i \wedge x_i.a = y_i)$$

Avec l'instruction $i \equiv u.a{:=}v$, nous obtenons très facilement avec $\mathcal{POST}^*$:

$$\mathcal{POST}^*(i, E) \equiv \bigwedge_{i \in [1..10]} (u = x_i) \wedge u.a = v$$

C'est-à-dire que d'une formule avec 20 atomes, nous passons à une formule avec 11 atomes. L'ancienne version de $\mathcal{POST}$ aurait fourni:

$$\mathcal{POST}(i, E) \equiv \bigwedge_{i \in [1..10]} ((u = x_i \wedge u.a = v) \wedge ((u = x_i \wedge u.a = v) \vee (u \neq x_i \wedge u.a = v \wedge x_i.a = y_i)))$$

Cette formule comprend 70 atomes, mais surtout, sa simplification, qui passe par une mise sous forme normale disjonctive, produit 2^{10} formules, ayant chacune entre 40 et 50 atomes ! L'importance d'une mise en œuvre efficace est donc clair.

6.2.2 Définition de $\mathcal{POST}^*$ pour l'instruction `forone`

Nous rappelons que l'instruction `forone` permet d'isoler et de capturer les aspects navigationnels des instructions. Considérons une instruction:

$$p\texttt{.epoux.epoux:=}q$$

Nous réécrivons cette instruction:

$$\texttt{forone}\ x\ \texttt{where}\ p\texttt{.epoux}=x\ \texttt{do}\ x\texttt{.epoux:=}q$$

Nous donnons ici une version améliorée de $\mathcal{POST}$ pour ce type d'instruction, en considérant que la formule en entrée est un environnement.

Définition 43 Soit E un environnement, et i une instruction telle que:

$$i \equiv \texttt{forone}\ x\ \texttt{where}\ y.a = x\ \texttt{do}\ j(x)$$

Cette instruction exécute j, où x dénote le seul objet tel que $y.a = x$. Nous définissons $\mathcal{POST}^*(i, E)$ comme suit:

- Si il existe un atome A dans E tel que $A \equiv y.a = u$, alors $\mathcal{POST}^*(i, E) \equiv \mathcal{POST}(j(u), E)$.
- Sinon $\mathcal{POST}^*(i, E) \equiv \exists x : \mathcal{POST}^*(i, (y.a = x) \wedge E)$ (c'est-à-dire, la définition de $\mathcal{POST}^*$ reste inchangée).

Cette définition de $\mathcal{POST}^*$ a pour but d'éviter d'introduire inutilement une variable représentant un objet particulier, lorsque cet objet peut directement être trouvé et référencé dans l'environnement initial. Comme toujours, nous prouvons que cette nouvelle version est correcte.

Théorème 11 Soit E un environnement, et i une instruction telle que:

$$i \equiv \texttt{forone}\ x\ \texttt{where}\ y.a = x\ \texttt{do}\ j(x)$$

Soient B une base de données, B^+ la base de données résultante de l'exécution de i sur B et α une assignation pour les variables libres de E. Alors:

$$(B_\alpha \models E) \Rightarrow (B^+_\alpha \models \mathcal{POST}^*(i, E))$$

Preuve: Le second cas a déjà été prouvé dans le théorème 2, page 67. Pour l'autre cas, supposons qu'un atome $A \equiv y.a = u$ figure dans E. Dans B_α, le seul objet qui satisfasse $y.a = u$ est u. Pour tout autre objet v qui satisfait $y.a = v$, on a $v = u$, car a est une fonction. L'instruction j est donc exécutée sur l'objet u. Par hypothèse d'induction, on a donc bien: $B^+_\alpha \models \mathcal{POST}^*(j(u), E)$.

Nous illustrons maintenant à l'aide d'un exemple le gain qui est obtenu par l'application de ce nouveau transformateur de prédicats. Soit l'instruction:

$$i \equiv \texttt{forone}\ x\ \texttt{where}\ p.\texttt{epoux} = x\ \texttt{do}\ x.\texttt{patron} := \texttt{nil}$$

et l'environnement suivant:

$$E \equiv (p.\texttt{epoux} = q)$$

L'application de $\mathcal{POST}$ à i et E donne:

$$\begin{aligned}
\mathcal{POST}(i, E) &= \mathcal{POST}(i, (p.\texttt{epoux} = q)) \\
&= \exists x : \mathcal{POST}(\ x.\texttt{patron} := \texttt{nil}\ ,\ p.\texttt{epoux} = x \wedge p.\texttt{epoux} = q\ ) \\
&= \exists x : \mathcal{POST}(\ x.\texttt{patron} := \texttt{nil}\ ,\ p.\texttt{epoux} = x \wedge p.\texttt{epoux} = q\ ) \\
&= \exists x : p.\texttt{epoux} = x \wedge ((p = x \wedge x.\texttt{patron} = \texttt{nil}) \\
&\quad \vee (p \neq x \wedge x.\texttt{patron} = \texttt{nil} \wedge p.\texttt{patron} = q))
\end{aligned}$$

Le résultat fourni par la nouvelle version est plus simple:

$$\begin{aligned}
\mathcal{POST}^*(i, E) &= \mathcal{POST}^*(i, (p.\texttt{epoux} = q)) \\
&= \mathcal{POST}^*(\ q.\texttt{patron} := \texttt{nil}\ ,\ p.\texttt{epoux} = q\ ) \\
&= p.\texttt{epoux} = q \wedge q.\texttt{patron} = \texttt{nil}
\end{aligned}$$

Afin de compléter la définition de $\mathcal{POST}^*$, nous devons considérer aussi les cas où, soit l'instruction n'est ni une affectation, ni un `forone`, soit la formule n'est pas un environnement.

Définition 44 Soit i une instruction et F une formule. Si soit:

– i n'est ni une affectation. ni un `forone`,

– F n'est pas un environnement,

alors $\mathcal{POST}^*$ est défini comme suit:

$$\mathcal{POST}^*(i, F) \equiv \mathcal{POST}(i, F)$$

6.3 Ordre d'évaluation du transformateur de prédicats $\mathcal{POST}^*$

La définition de $\mathcal{POST}$ donnée dans le chapitre précédent ne suggère pas un algorithme déterministe pour capturer la sémantique d'une méthode. Différents ordres d'évaluation sont possibles, et le théorème 2 page 67 nous dit qu'ils sont tous corrects. Toujours dans la volonté d'aboutir à une implantation efficace, nous allons indiquer dans cette section un ordre d'évaluation "satisfaisant", dans le sens où il produit des formules plus courtes et avec moins de quantificateurs. Cet ordre d'évaluation a bien sûr été déterminé par l'expérience, en faisant fonctionner le prototype, plutôt qu'avec des considérations théoriques.

Le principe des optimisations est simple: elles consistent toujours à éviter au maximum les redondances, en factorisant au maximum ce qui peut l'être. En outre, nous allons commencer à prendre en compte la phase ultérieure du traitement, à savoir, la démonstration automatique de théorèmes. Au total, il nous faut:

– réduire la taille des formules manipulées et générées,

– limiter le nombre de quantificateurs des formules.

La réduction du nombre de quantificateurs concerne plus particulièrement la phase de démonstration automatique de théorèmes. L'évaluation du transformateur de prédicats passe par différentes phases:

1. Mise sous forme prénexe disjonctive de la pré-condition.
2. Simplification de la pré-condition par élimination des contradictions et des redondances.
3. Application de $\mathcal{POST}^*$ pour les quantificateurs.
4. Application, le cas échéant, de $\mathcal{POST}^*$ pour l'instruction "`if`" et l'instruction de séquencement.
5. Application, le cas échéant, de $\mathcal{POST}^*$ pour la disjonction.
6. Application de $\mathcal{POST}^*$ pour l'instruction "`forone`", ainsi que pour l'instruction d'assignation. La pré-condition est forcément un environnement.

Chaque phase est justifiée ci-après, ainsi que l'ordre des phases.

6.3.1 Mise sous forme prénexe disjonctive

Nous rappelons qu'une formule F est sous forme prénexe disjonctive si et seulement si elle peut s'écrire:

$$F \equiv Q_1 x_1 : ...Q_n x_n : E_1 \vee ... \vee E_m$$

où Q_1,...,Q_n sont des quantificateurs ($\forall$ ou $\exists$), x_1,...,x_n sont des variables, et E_1,...,E_m sont des environnement (c'est-à-dire des conjonctions d'atomes). En outre, toute formule est équivalente à une formule sous forme prénexe disjonctive (pour plus de précisions, consulter par exemple [CL93]), d'où l'expression: "mettre sous forme prénexe disjonctive". Nous pouvons donc supposer sans rien perdre en généralité que la pré-condition est sous cette forme. Remarquons au passage que la première pré-condition est de taille réduite: c'est la contrainte d'intégrité. Elle est donc aisée à mettre sous forme prénexe disjonctive. Ensuite, partant d'une formule qui est sous cette forme, $\mathcal{POST}^*$ produit une autre formule qu'il est aisé de maintenir sous cette forme. Le coût de cette transformation n'est donc pas prohibitif. L'intérêt de cette phase est double:

– Elle isole les quantificateurs.

– Elle fait bien ressortir chaque environnement dont est constitué une formule.

6.3.2 Simplification de la pré-condition

Pour chaque environnement E_i de F, nous calculons $\overline{E_i}$, pour détecter un éventuelle contradiction. En effet, si il existe une atome A tel que $A \in \overline{E_i}$ et $\neg A \in \overline{E_i}$, alors $E_i \Leftrightarrow \texttt{faux}$. Nous pouvons donc réécrire F de la façon suivante:

$$F \equiv Q_1 x_1 : ...Q_n x_n : E_1 \vee ... \vee E_{i-1} \vee E_{i+1} \vee E_m$$

6.3.3 Extraction des quantificateurs

Ensuite, il est important d'extraire les quantificateurs à l'aide de la définition appropriée de $\mathcal{POST}^*$, dont nous rappelons la définition:

$$\mathcal{POST}^*(i, Qx : F) \ \equiv \ Qx : \mathcal{POST}^*(i, F)$$

où Q dénote un quantificateur. Cette extraction doit se faire *avant* une éventuelle application de $\mathcal{POST}^*$ pour l'instruction conditionnelle. En effet, considérons une instruction conditionnelle quelconque:

$$i \equiv \texttt{if}\ G\ \texttt{then}\ j$$

ainsi qu'une formule F comportant n quantificateurs. L'application de $\mathcal{POST}^*$ à i et F est définie par:

$$\mathcal{POST}^*(i, F) \equiv \mathcal{POST}^*(j, G \wedge F) \vee (\neg G \wedge F)$$

Cette formule comportera au minimum $2 \times n$ quantificateurs (au moins n dans le premier membre, ainsi que dans le second). Par contre si $F \ \equiv \ Q_1 x_1 \ : \ ..., Q_n x_n \ : \ H$, alors, en commençant par une extraction des quantificateurs, nous obtenons:

$$\mathcal{POST}^*(i, F) \equiv Q_1 x_1 : ...Q_n x_n : \mathcal{POST}(j, G \wedge H) \vee (\neg G \wedge H)$$

Les quantificateurs sont naturellement factorisés par l'ordre dévaluation. Cette factorisation peut bien sûr se faire a posteriori, mais elle est nettement plus coûteuse.

6.3.4 Application de $\mathcal{POST}^*$ aux instructions conditionnelles

Ensuite, si l'instruction en entrée de $\mathcal{POST}$ est une instruction conditionnelle, il faut impérativement appliquer la définition de $\mathcal{POST}^*$ correspondante, avant de décomposer éventuellement la formule. En effet, considérons une instruction conditionnelle quelconque:

$$i \equiv \texttt{if}\ G\ \texttt{then}\ j$$

Ayant déjà appliqué l'extraction des quantificateurs lors de la phase précédente, nous pouvons supposer que la pré-condition F s'écrit:

$$F \equiv E_1 \vee ... \vee E_n$$

En appliquant $\mathcal{POST}^*$ d'abord à la disjonction, puis au "`if`", nous obtenons:

$$\begin{array}{rcl} \mathcal{POST}(i,F) & = & \mathcal{POST}(i,E_1) \vee ... \vee \mathcal{POST}(i,E_n) \\ & = & (\mathcal{POST}(j,G \wedge E_1) \vee (\neg G \wedge E_1)) \\ & & \wedge(\mathcal{POST}(j,G \wedge E_2) \vee (\neg G \wedge E_2)) \\ & & ... \\ & & \wedge(\mathcal{POST}(j,G \wedge E_n) \vee (\neg G \wedge E_n)) \end{array}$$

La formule G est répliquée à maintes et maintes reprises ($2 \times n$ fois). Cette évaluation sera évidement catastrophique et beaucoup moins efficace qu'une l'évaluation qui commence par traiter l'instruction conditionnelle:

$$\mathcal{POST}(i,F) \quad = \quad \mathcal{POST}(j,G \wedge F) \vee (\neg G \wedge F)$$

Les séquences d'instruction sont traitées au même niveau que les instructions conditionnelles. Ce choix est sans effets sur le résultat fourni par $\mathcal{POST}^*$.

6.3.5 Conjonctions, affectations, `forone` et séquencement

Arrivés à ce point, nous pouvons supposer que la pré-condition est une formule F:

$$F \equiv E_1 \vee ... \vee E_n$$

Et que l'instruction i est soit une instruction d'affectation, soit une instruction "`forone`". Nous commençons donc par appliquer $\mathcal{POST}^*$ à la disjonction. Ensuite, il ne reste plus qu'à appliquer $\mathcal{POST}^*$ à l'instruction en cours. Ceci est possible grâce à la définition de $\mathcal{POST}^*$ donnée précédemment. Elle nous permet traiter les environnements de manière globale, sans avoir à appliquer $\mathcal{POST}$ pour la conjonction.

La mise en œuvre décrite en détail pour le transformateur de prédicats $\mathcal{POST}$ peut être naturellement appliquée pour le transformateur de prédicats arrière $\mathcal{PRE}$. L'idée qui guide l'ensemble de ces optimisations est prendre en compte, d'une façon simplifiée, la sémantique des formules en entrée. Ceci passe toujours par la mise en forme prénexe disjonctive, afin de faire ressortir des environnements, qui peuvent être traités de façon globale.

6.4 Description des prototypes

Les deux prototypes que nous décrivons ici appliquent les techniques de preuve de programme que nous avons exposé au chapitre 5, ainsi que les transformateurs de prédicats optimisés $\mathcal{POST}^*$ et $\mathcal{PRE}^*$. Le premier prototype utilise le transformateur de prédicats *avant*, alors que le second utilise le transformateur de prédicats *arrière*. Il est clair qu'un gestionnaire de contraintes d'intégrité complet doit aussi intégrer les techniques décrites dans [Nic79, LT85, HI85, BM86, KSS87, HCN84, WSK83, BDM88, BD95], qui permettent d'optimiser la vérification des contraintes à l'exécution.

À l'opposé d'un produit fini industriel, ces prototypes ont été conçus principalement pour expérimenter facilement différentes stratégies et heuristiques pour la preuve de programme. Ils offrent donc avant toutes choses des fonctions de base qui permettent de faire de la vérification de programme. Le fonctionnement de ces prototypes est très simple: ils prennent en entrée un fichier qui comporte une contrainte d'intégrité et une méthode et indiquent si la méthode est susceptible de violer cette contrainte d'intégrité. Partant de cette fonctionnalité de base, des preuves de programme plus complexes peuvent être réalisées, avec notamment plusieurs contraintes et plusieurs méthodes (ou mieux: un schéma entier). Les différents modules qui composent les deux prototypes sont les suivants:

- Un gestionnaire de formules de logique. Il permet la construction et la manipulation des formules de logique. Les fonctions de ce module sont toutes très classiques (mise sous forme prénexe disjonctive, substitutions, etc.). Ce module comporte en outre les fonctions qui permettent de décomposer formules et instructions, comme cela est fait au chapitre 4.
- Un gestionnaire d'environnements. Ce module permet de capturer la sémantique des environnements (en calculant $\overline{E}$ à partir d'un environnement E). Ceci est fait afin de mettre en œuvre l'ensemble des optimisations décrites plus tôt dans ce chapitre.
- Les transformateurs de prédicats (un par prototype).
- Le démonstrateur automatique de théorèmes. En fait, nous utilisons deux démonstrateurs de théorèmes:
  - Pour l'analyse en *avant*, nous utilisons un démonstrateur très simple fondé sur la méthode des tableaux. Ce démonstrateur est particulièrement adapté car les formules produites par cette analyse sont en forme prénexe disjonctive et la méthode des tableaux procède en fait en mettant la formule à réfuter en forme normale disjonctive.
  - Pour l'analyse en *arrière*, nous utilisons un démonstrateur très sophistiqué nommé "OTTER" [McC], fondé sur la résolution. La pré-condition obtenue via le transformateur de prédicats arrière est, pour des raisons d'efficacité, en forme normale disjonctive. La formule à réfuter (qui comporte la négation de la pré-condition) est donc en forme normale *conjonctive* (ou forme clausale). L'application de la méthode des tableaux est donc prohibitive, alors que la forme clausale est particulièrement adaptée pour la résolution.
- Le module d'analyse lexicale et syntaxique, écrit en Lex et Yacc. C'est aussi ce module qui met en œuvre la vérification automatique des méthodes, en faisant appel aux différents autres modules.

L'avantage de ces prototypes est qu'ils offrent les fonctionnalités de base d'un système de preuve de programme d'une manière très modulaire. Ainsi, ils nous ont permis de tester rapidement différentes stratégies de preuve sans avoir à les coder de façon définitive, et sans remettre en cause l'ensemble des prototypes. La structure globale des prototypes est décrite en figure 6.1. Ces prototypes ont été écrits en langage C et comportent au total environs 6000 lignes de code.

Le déroulement d'une preuve de méthode se déroule de la façon suivante:

- La contrainte d'intégrité et les instructions sont décomposées.

– Le transformateur de prédicats (avant ou arrière) est appliqué à la méthode, avec en entrée la contrainte d'intégrité.

– Le résultat fourni par le transformateur de prédicats est récupéré par le démonstrateur automatique de théorèmes qui tente de prouver que la post-condition implique bien la contrainte (en avant), ou que la contrainte implique bien la pré-condition (en arrière).

En outre une analyse préliminaire fondée sur l'information de typage peut être appliquée pour détecter les cas ou la contrainte sera trivialement préservée. Cette analyse est détaillée dans [BD95]. Le déroulement d'une preuve de programme est détaillé en figure 6.2.

$$\begin{array}{ccc} & \boxed{\text{Analyseur syntaxique}} & \\ \swarrow & & \searrow \\ \boxed{\text{Transformateur de prédicats}} & & \boxed{\text{Démonstrateur automatique de théorèmes}} \\ \downarrow & & \\ \boxed{\text{Gestionnaires de formules et d'environnements}} & & \end{array}$$

FIG. 6.1 - *Structure des prototypes.*

$$\begin{array}{c} \boxed{\text{Analyse préliminaire}} \\ \downarrow \\ \textit{indique si une analyse approfondie est nécessaire} \\ \downarrow \\ \boxed{\text{Décomposition de la contrainte et de la méthode}} \\ \downarrow \\ \textit{produit une nouvelle contrainte et une nouvelle méthode.} \\ \downarrow \\ \boxed{\text{Application du transformateur de prédicats}} \\ \downarrow \\ \textit{produit une post ou pré-condition} \\ \downarrow \\ \boxed{\text{Phase de démonstration automatique}} \\ \downarrow \\ \textit{indique si la contrainte est préservée par la méthode} \end{array}$$

FIG. 6.2 - *Déroulement d'une preuve de méthode.*

6.5 Résultats de l'expérimentation

Il est important de pouvoir évaluer d'un point de vue pragmatique la faisabilité de notre approche. En effet, il est aisé de voir qu'en théorie, notre approche peut être très coûteuse. Il y a deux raisons à cela:

– Pour commencer, les transformateurs de prédicats produisent des formules dont la taille est assez grande. Ceci n'est pas propre à notre approche mais est liée au langage de programmation. Celui-ci autorise notamment l'aliasing, ce qui induit une certaine explosion combinatoire. Or, il est impossible de concevoir un langage de programmation pour bases de données sans aliasing.

– Ensuite, la démonstration automatique de théorèmes en logique du premier ordre (avec égalité) est extrêmement coûteuse. On peut à ce titre s'étonner de ce que nous arrivions à prouver quoi que ce soit. Selon Fitting [Fit90], "Avec les règles d'égalité, les démonstrateurs automatiques de théorèmes ont tellement de chemins à explorer pour essayer de produire des preuves qu'ils ne réussissent que pour des théorèmes simples et pas très intéressants. [ces systèmes] peuvent mettre des années pour prouver quoi que ce soit d'intéressant."

Le coût d'une preuve de méthode n'est donc pas propre à notre approche, mais est lié au langage de programmation et à la théorie sous-jacente au système de preuve. Il n'y a donc peu d'espoir qu'une autre méthode fournisse de bien meilleurs résultats.

Nous donnons ci-dessous différents résultats. Dans un premier temps, nous nous concentrerons sur le premier prototype, qui utilise le transformateur de prédicats avant.

– Le premier prototype utilisant un démonstrateur automatique de théorèmes, nous avons, dans un premier temps, fait abstraction des performances propres de ce démonstrateur. Nous nous sommes concentrés sur le temps mis pour générer la post-condition. Ces temps sont indiqués en figure 6.3.

– Ensuite, cette génération n'étant que la première étape d'une preuve de programme, nous avons étudié le temps mis par le premier prototype pour obtenir une preuve complète, avec une profondeur de recherche, pour la phase de démonstration, fixée a priori. Ces temps sont indiqués en figure 6.4. Nous rappelons que la profondeur de recherche de la méthode des tableaux correspond au nombre de fois que la règle d'expansion de branches γ est utilisée.

– Enfin, nous avons indiqué, en figure 6.5, le temps mis par le prototype pour trouver une preuve, sans fixer a priori une profondeur de recherche. Dans ce cas, le démonstrateur fait plusieurs tentatives de preuves, avec différentes profondeurs de recherche.

Cette évaluation en trois parties a pour intérêt de bien mettre en valeur le fait que la phase la plus coûteuse de notre approche est la démonstration automatique de théorèmes. Cette conclusion s'applique aussi bien à l'analyse en avant qu'en arrière.

Nous soulignons en outre que, en ce qui concerne le premier prototype, le démonstrateur que nous avons choisi est vraiment très rudimentaire et qu'il ne comporte quasiment pas d'optimisations. Un produit industriel devra évidement utiliser un démonstrateur efficace. En particulier, lors d'une recherche sans profondeur de recherche fixée, le démonstrateur doit refaire maintes et maintes fois le même traitement. Ces redondances doivent à tout prix être évitées.

À titre de comparaison, nous avons aussi mis en figure 6.5 les temps obtenus pour l'analyse en arrière. Nous ne tenterons pas de justifier en théorie les performances obtenues. D'une façon très pragmatique, nous pouvons observer que le premier prototype met plus de temps pour les exemples "simples", mais beaucoup moins de temps pour les exemples complexes. L'exemple de la méthode `mariage2` avec la contrainte `C3` est particulièrement marquant. D'autant plus que le premier prototype est en fait le couplage d'un transformateur de prédicats avant et de la méthode des tableaux, qui ont tous les deux la réputation d'être respectivement moins efficaces que le transformateur de prédicats arrière et la résolution.

Ces exemples permettent de bien établir que notre méthode peut effectivement être mise en œuvre. Nous n'avons pas de règle générale pour déterminer à l'avance le temps pour obtenir une preuve. Toutefois, nous pouvons bien voir que les couples (méthode,contrainte) pour lesquelles la preuve de programme fonctionne bien sont relativement courts. Les contraintes ont un ou deux quantificateurs (nous soulignons au passage que certains quantificateurs sont produits par la décomposition décrite au chapitre 4). Les méthodes sont elles aussi de taille réduite. L'utilisation des termes complexes, que ce soit dans les méthodes ou les contraintes, est à éviter.

6.5.1 Une famille d'exemples : les contraintes d'implication

Les temps obtenus précédemment étaient difficiles à comparer car les méthodes et les contraintes ont des structures assez distinctes, et il est difficile de repérer une quelconque régularité dans la preuve. Nous allons nous intéresser à une famille d'exemples, dont la structure est régulière. Commençons par décrire les contraintes. Celles-ci sont de la forme:

$$\boxed{CI_n \equiv \forall x \in S : (x.a_1 = \texttt{nil} \wedge ... \wedge x.a_n = \texttt{nil}) \Rightarrow x.b = \texttt{nil}}$$

Nous appellerons ce type de contraintes des *contraintes d'implication*. S dénote un ensemble d'objets d'une classe κ. Ensuite, nous définissons deux types de méthodes:

```
method t_n^1 (q) in class κ {
  if (self.a_1 ≠ nil ∨ ... ∨ self.a_n ≠ nil)
    self.b:=q;}

method t_n^2 () in class κ {
  self.a_1:=nil;
  ...
  self.a_n:=nil;
  self.b:=nil;}
```

méthode	classe	contrainte	temps
`separe`	`Personne`	`C1`	0.02s.
`separe`	`Parent`	`C1`	0.02s.
`separe`	`Employe`	`C1`	0.04s.
`separe`	`Employe`	`C4`	0.04s.
`separe`	`Parent`	`C6`	0.07s.
`mariage`	`Personne`	`C2`	0.05s.
`mariage`	`Parent`	`C2`	0.11s.
`mariage`	`Parent`	`C3`	0.13s.
`debit`	`Personne`	`C5`	0.10s.
`debit`	`Parent`	`C5`	1.61s.
`meilleur-ami`	`Personne`	`C7`	0.05s.

FIG. 6.3 - *Temps de calcul du transformateur avant.*

méthode	classe	contrainte	profondeur	temps
`separe`	`Personne`	`C1`	2	0.18s.
`separe`	`Parent`	`C1`	5	0.31s.
`separe`	`Employe`	`C1`	3	0.22s.
`separe`	`Employe`	`C4`	1	1.16s.
`separe`	`Parent`	`C6`	4	1.38s.
`mariage`	`Personne`	`C2`	1	0.46s.
`mariage`	`Parent`	`C2`	3	1.30s.
`mariage`	`Parent`	`C3`	3	3.37s.
`debit`	`Personne`	`C5`	3	1.17s.
`debit`	`Parent`	`C5`	3	1.47s.
`meilleur-ami`	`Personne`	`C7`	3	0.24s.

FIG. 6.4 - *Temps de preuve avec profondeur fixée.*

Une propriété remarquable des deux méthodes t_n^1 et t_n^2 est qu'elles préservent toutes les deux la contrainte CI_n. Remarquons en outre que nous n'avons pas affaire à une contrainte et à une méthode particulières, mais à toute une famille de contraintes et de méthodes: il suffit de faire varier le paramètre n.

Nous avons étudié le temps mis par les deux prototypes pour détecter que t_n^1 et t_n^2 préservent CI_n, en faisant varier n. Remarquons tout d'abord que la profondeur de recherche (pour le premier prototype) semble être indépendante de la valeur de n (ce n'est pas un résultat théorique, mais une constatation expérimentale). Cette profondeur de recherche, dans tous les cas, est de 1. Nous donnons en figure 6.6 et en figure 6.7 les temps respectifs obtenus.

Les performances de l'analyse en arrière semblent être ici supérieures à celles de l'analyse en avant. Cependant, pour les méthodes du type t_n^1, à partir de $n > 4$, l'analyse en arrière s'enlise et ne fournit plus de résultat. Pour les méthodes du type t_n^2, la supériorité de l'analyse en arrière est flagrante. Remarquez que pour ces méthodes, nous pourrions aussi appliquer la méthode ad-hoc définie en section 5.1.2 au chapitre 5 (page 60). Les temps de calcul seraient alors quasiment nuls.

6.6 Conclusion

Nous avons décrit dans ce chapitre la mise en œuvre concrète des techniques exposées au chapitre précédent.

En premier lieu, nous avons défini un type de formules, les *environnements*, qui permettent de percevoir très partiellement la sémantique des formules manipulées par les transformateurs de prédicats.

Nous avons redéfini un transformateur de prédicats avant pour ce type de formules qui, à l'opposé des transformateurs présentés dans le chapitre précédent, prend en compte la sémantique des formules traitées. Nous avons aussi défini un ordre d'application du transformateur de prédicats avant qui limite la taille des formules générées. Ces optimisations s'appliquent aussi naturellement au transformateur de prédicats arrière. Soulignons que ces résultats n'ont pas été obtenus a priori, mais sont le fruit d'une expérimentation. Le premier prototype implanté reprenait textuellement la définition originelle du transformateur de prédicats et, pour des raisons d'efficacité, n'a pu être exploité.

Nous avons ensuite présenté les deux prototypes qui appliquent les techniques de preuve de programmes. Le premier prototype est fondé sur l'utilisation d'un transformateur de prédicats avant, alors que le second utilise le transformateur de prédicats arrière. Les premiers résultats obtenus ne permettent pas de dire qu'une approche est meilleure que l'autre. L'analyse en avant semble fournir de meilleurs résultats sur des exemples complexes, alors que l'analyse en arrière est plus rapide sur des exemple simples.

La conclusion essentielle de cette expérimentation est la *faisabilité* de la vérification automatique que les méthodes préservent les contraintes. Les méthodes vues ici sont certes courtes mais sont assez représentatives des méthodes écrites dans les systèmes réels. Le but que nous nous étions fixé n'était pas de prouver qu'une méthode quelconque préserve une contrainte quelconque, mais de détecter les méthodes qui "naturellement", préservent certaines contraintes. Or, en général, ces méthodes sont courtes et font des modifications simples. La détection de telles méthodes se fera donc aisément. À ce titre, notre approche remplit assez bien son rôle.

Les problèmes rencontrés, et notamment le temps d'obtention d'une preuve, ont principalement pour origine l'expressivité du langage utilisé pour capturer la sémantique des méthodes (et son indécidabilité).

méthode	classe	contrainte	temps en avant	temps en arrière
`separe`	`Personne`	`C1`	0.21s.	0.12s.
`separe`	`Parent`	`C1`	0.56s.	0.37s.
`separe`	`Employe`	`C1`	0.53s.	0.49s.
`separe`	`Employe`	`C4`	1.06s.	0.22s.
`separe`	`Parent`	`C6`	1.21s.	2.48s.
`mariage`	`Personne`	`C2`	0.41s.	0.33s.
`mariage`	`Parent`	`C2`	1.30s.	9.04s.
`mariage`	`Parent`	`C3`	3.60s.	1h 06min. 35s.
`debit`	`Personne`	`C5`	1.30s.	76.42s.
`debit`	`Parent`	`C5`	1.65s.	154.23s.
`meilleur-ami`	`Personne`	`C7`	3.26s.	0.31s

FIG. 6.5 - *Temps de preuve global en avant et en arrière.*

valeur de n	temps en avant	temps en arrière
1	1.16s.	0.23s.
2	1.57s.	0.61s.
3	6.86s.	2.14s.
4	20.87s.	17.32s.
5	53.40s.	échec au bout de 6h.2min.43s.
6	116.86s.	-
7	237.69s.	-

FIG. 6.6 - *Temps de preuve pour t_n^1 et CI_n.*

valeur de n	temps en avant	temps en arrière
1	0.90s.	0.24s.
2	5.70s.	0.21s.
3	20.81s.	0.25s.
4	70.75s.	0.30s.
5	182.94s.	0.33s.

FIG. 6.7 - *Temps de preuve pour t_n^2 et CI_n.*

Dans la mesure où la résolution de notre problème est intimement liée à l'utilisation de ce langage, il est peu probable qu'une méthode équivalente à la notre obtienne de bien meilleurs résultats.

Chapitre 7

Conclusions et perspectives

7.1 Conclusions

Le but de cette thèse était de proposer une approche afin de déterminer à la compilation si un programme qui met à jour une base de donnée est susceptible de violer un ensemble de contraintes d'intégrité. Une telle approche permet de réduire l'ensemble des contraintes à vérifier après l'exécution d'un programme. Ainsi, les contraintes qui ne sont pas susceptibles d'être violées par ce programme n'ont pas à être vérifiées, car elles sont par nature *préservées*. Sur le plan pratique, ceci permet de réduire le coût en temps de la gestion de l'intégrité.

Ce travail est avant tout un travail *d'application* et *d'adaptation* des techniques d'analyse statique développées depuis trente ans au cadre des bases de données. Nous avons présenté un certain nombre de ces techniques dans le chapitre 3 : Logique de Hoare, Calcul de Dijkstra et Interprétation Abstraite. Nous avons aussi évoqué les solutions existantes à notre problème. Ces solutions ont pour principal défaut d'être difficiles à mettre en œuvre, et se concentrent avant tout sur les aspects théoriques du problème.

Nous avons ensuite présenté dans le chapitre 4 le cadre d'application de notre approche : il s'agit d'un système de gestion de bases de données orienté objet. Dans ce contexte, un schéma décrit trois aspects d'une base de données : les structures de données (les types), les traitements (les méthodes), et les contraintes d'intégrité. Le modèle de données intègre la plupart des concepts des modèles orienté objet : types, classes, héritage, méthodes, surcharge, etc. Nous avons en outre présenté une façon de simplifier la structure des contraintes et des méthodes afin de faciliter l'analyse statique. Cette simplification a pour but d'isoler les aspects navigationnels des langages de programmation orientés objet, des mises à jour proprement dites.

Dans le chapitre 5, nous avons présenté le concept central sur lequel est fondé notre approche : celui de *transformateur de prédicats*. Nous définissons deux transformateurs de prédicats, dont le but est de capturer la sémantique logique des méthodes. Nous présentons en premier lieu un transformateur de prédicats *avant*, qui produit une post-condition qui est nécessairement satisfaite après l'exécution d'une méthode, pour peu qu'une pré-condition soit satisfaite avant. Nous présentons aussi un transformateur de prédicats *arrière*, qui génère une pré-condition suffisante pour qu'une post-condition soit satisfaite après l'exécution d'une méthode. Ces deux transformateurs de prédicats permettent de déterminer si une méthode est susceptible de violer une contrainte d'intégrité. Nous ramenons en fait notre problème à celui de la preuve d'un théorème en logique du premier ordre. Ce théorème peut être prouvé à l'aide de techniques de démonstration automatique de théorèmes comme la méthode des tableaux, ou la résolution. En outre, le transformateur de prédicats arrière peut en théorie être utilisé pour corriger les méthodes.

Enfin, dans le chapitre 6, nous montrons comment les transformateurs de prédicats sont mis en œuvre dans deux prototypes différents. Cette mise en œuvre consiste grossièrement à prendre en compte la sémantique des formules manipulées, afin de réduire la taille des résultats générés. Les résultats des différentes expérimentations effectuées sont fournis. Ces mesures indiquent clairement la faisabilité de notre approche, en particulier dans le cadre des bases de données orientées objet, où les traitements opérés (les méthodes) sont de petite taille et très modulaires.

7.2 Perspectives

Nous avons vu que les problèmes rencontrés par notre approche sont de deux ordres :

– Premièrement, l'essentiel du coût d'une analyse statique vient en (très) grande partie de la phase de démonstration automatique de théorèmes. Ceci peut dans bien des cas empêcher d'obtenir un résultat dans un temps raisonnable.

– Ensuite, nous avons vu que les transformateurs de prédicats sont "abstraits". Ils ne capturent pas toute la sémantique des méthodes (notre approche est correcte mais pas complète). L'information collectée par les transformateurs peut donc être insuffisante et notre approche peut conclure à tort qu'une méthode viole potentiellement une contrainte.

En ce qui concerne le coût de la phase de démonstration automatique de théorèmes, il y a peu d'espoir de voir une amélioration significative se produire. La logique du premier ordre, que nous utilisons à outrance, est indécidable. Ceci limite considérablement les possibilités d'optimisations de cette phase. Toutefois, il n'est pas interdit d'espérer et notre travail bénéficiera immédiatement de tous les progrès faits dans le domaine de la démonstration automatique de théorèmes.

Ensuite, il est possible d'"affiner" les résultats fournis par les transformateurs de prédicats. Nous avons vu en particulier que le fait d'émettre des hypothèses au vol rendait le transformateur de prédicats oublieux (ces hypothèses pouvaient être contradictoires avec l'hypothèse de cohérence initiale). Il faudrait donc émettre l'ensemble des hypothèses "a priori", les confronter à la contrainte d'intégrité, puis faire l'analyse statique partant des hypothèses qui ne sont pas en contradiction avec la contrainte d'intégrité. Cette approche alternative pourrait fournir des résultats positifs sur des exemples que nous n'avons pas pu traiter avec notre technique. Nous espérons aussi pouvoir utiliser avec profit les techniques de test de logiciels.

Ce travail peut être étendu pour englober les contraintes d'intégrité temporelles. Les contraintes que nous avons considérées dans ce travail ne limitent l'état des objets de la base de données qu'à un instant donné. Les contraintes temporelles ont pour but de limiter aussi l'évolution dans le temps des objets.

Une autre voie pour l'extension de notre approche serait le domaine des bases de données déductives. Il serait très intéressant de pouvoir exprimer des contraintes d'intégrité portant sur des prédicats intentionnels. La difficulté ici serait de définir un transformateur de prédicats fiable pour ces prédicats intentionnels.

Sur un plan pratique, l'outil que nous avons présenté dans ce travail semble fournir des résultats suffisamment positifs pour être utilisé dans un véritable système de gestion de bases de données. Il serait en particulier très intéressant de construire un nouveau système de gestion de bases de données bénéficiant à la fois des dernières avancées dans le domaine des langages de programmation orientés objet (en particulier le typage), ainsi que des techniques exposées dans cette thèse. Ces techniques doivent aussi naturellement s'appliquer au cadre de Eiffel, qui est un langage de programmation orienté objet qui permet de poser des conditions sur les données en entrée et en sortie des méthodes par le biais de pré et de post-conditions. Nos analyses devraient permettre de déterminer si il est nécessaire de tester à l'exécution la post-condition d'une méthode.

Reprenant le constat fait par [AHV95], nous pouvons dire que l'étude de la sémantique des mises à jour dans le contexte des bases de données a été à ce jour relativement négligée. Pourtant, de nombreux outils comme l'interprétation abstraite, l'évaluation partielle et le test de logiciels doivent trouver leur application dans le domaine des bases de données. Au delà de l'intégrité, il existe de nombreux problèmes dont la résolution nécessite avant tout une bonne compréhension de la sémantique des mises à jour. Ainsi, le contrôle de concurrence sémantique et le problème de la mise à jour de vues devraient pouvoir être traités à l'aide de ces outils.

Bibliographie

[AB87] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *ACM International Conference on Management of Data (SIGMOD)*, pages 238–248, Denver, Colorado, USA, May 1991.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *International Conference on Very Large Databases*, pages 39–52, 1993.

[ABGO95] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. Fibonacci: a programming language for object databases. *VLDB Journal*, 4(3):403–444, July 95.

[AH87] S. Abramsky and C. Hankin. *Abstract interpretation of declarative languages*. Ellis Horwood, 1987.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Database Systems*. Adison Wesley, 1995.

[Apt81] K. R. Apt. Ten years of hoare's logic. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[ASU86] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Adison-Wesley, 1986.

[BD95] V. Benzaken and A. Doucet. Thémis: A Database Programming Language Handling Integrity Constraints. *VLDB Journal*, 4(3):493–518, 1995.

[BDM88] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In Missikoff Schmidt, Ceri, editor, *Proceedings of the EDBT International Conference*, LNCS 303, pages 488–505, 1988.

[BDS94] V. Benzaken, A. Doucet, and X. Schaefer. Integrity constraint checking optimization based on abstract interpretation. In *Proceedings of the 10th PRC BD3* , Clermont-Ferrand, France, August 1994.

[BDS95] V. Benzaken, A. Doucet, and X. Schaefer. Integrity Constraint Checking Optimization based on Abstract Databases Generation and Program Analysis. *Journal de l'Ingénierie des Systèmes d'Information*, 1(3):9–29, 1995.

[BM79] R. S. Boyer and J. S. Moore. *A computational logic*. Academic Press, 1979.

[BM86] F. Bry and R. Manthey. Checking Consistency of Database Constraints: A Logical Basis. In *Proceedings of the VLDB International Conference*, pages 13–20, August 1986.

[Bou93] F. Bourdoncle. *Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite*. PhD thesis, École Polytechnique, 1993.

[BS96] V. Benzaken and X. Schaefer. Ensuring efficiently the integrity of persistent object systems via abstract interpretation. In *Actes de la 12ème conférence sur les bases de données avancées*, Cassis, Août 1996. Morgan-Kaufmann. La version étendue se trouve dans [**?**].

[BS97] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In M. Aksit and S. Matsuoka, editors, *European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 60–85. Springer-Verlag, 1997.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types*, LNCS 173, pages 51–67. Springer-Verlag, 1984.

[CB80] M. A. Casanova and P. A. Bernstein. A formal system for reasonning about programs accessing a relational database. *ACM Transactions on Database Systems*, 2(3):386–414, July 80.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2^{nd} International Symposium on Programming*, 1976.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th popl, San Antonio, Texas*, pages 269–282, January 1979.

[CL93] R. Cori and D. Lascar. *Logique mathématique*, volume 1 of *Collection Axiomes*. Masson, 1993.

[Cla79] E. M. Clarke. Programming languages constructs for which it is impossible to obtain good hoare axiom systems. *Journal of the ACM*, 26(1):129–147, January 79.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[Del86] J. P. Delahaye. *Outils logiques pour l'intelligence artificielle*. Eyrolles, 1986.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Fit90] Melvin Fitting. *First-order logic and automated theorem proving*. Texts and monographs in computer science. Springer-Verlag, 1990.

[Flo67] Floyd. Assigning meanings to programs. In *Proc. Mathematical Aspects of Computer Science*, 1967.

[GG94] P. Gochet and P. Gribomont. *Logique, méthodes formelles pour l'étude des programmes.*, volume 2. Hermès, 1994.

[GM79] G. Gardarin and M. Melkanoff. Proving the Consistency of Database Transactions. In *VLDB International Conference*, pages 291–298, Rio, Brasil, October 1979.

[HCN84] L. Henschen, W. Mc Cune, and S. Naqvi. Compiling constraint checking programs from first order formulas. In H. Gallaire, J. Minker, and J.M. Nicolas, editors, *Advances in Database Theory*, volume 2. Plenum, 1984.

[Hec77] M. S. Hecht. *Flow analysis of computer programs*. North-Holland, 1977.

[Hen90] M. Hennessy. *The semantics of programming languages*. Wiley, 1990.

[HI85] A. Hsu and T. Imielinski. Integrity Checking for Multiple Updates. In *Proceedings of the ACM SIGMOD International Conference*, pages 152–168, 1985.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12, 1969.

[JN95] N. D. Jones and F. Nielson. Abstract interpretation. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, chapter 5, pages 527–636. Oxford Science Publication, 1995.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *Proceedings of the VLDB International Conference*, pages 61–70, 1987.

[Lal90] R. Lalement. *Logique, réduction, résolution*. Masson, 1990.

[Law95] Michael Lawley. Transaction safety in deductive object-oriented databases. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases, Singapore*, number 1013 in LNCS, pages 395–410. Springer-Verlag, Dec 1995.

[Llo87] J. W. Lloyd. *Foundations of logic programming*. Springer Verlag, 1987.

[LMB94] J. R Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly, 1994.

[LT85] J. W. Lloyd and R. W. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2(2), 1985.

[McC] William W. McCune. *OTTER 3.0 Reference Manual aud Guide*. Argonne National Laboratory. Peut être obtenu à l'adresse internet suivante: http://www.mcs.anl.gov/home/mccune/ar/otter.

[MH89] W. W. McCune and L. J. Henschen. Maintaining state constraints in relational databases: A proof theoretic basis. *Journal of the ACM*, 36(1):46–68, January 1989.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edimburgh, December 1981.

[Nic79] J.M. Nicolas. Logic for Improving Integrity Checking in Relational Databases. Technical report, ONERA-CERT, 1979.

[Pol96] Pierre-Yves Policella. *Cohérence dans les bases de données orientées objet*. PhD thesis, L.R.I., Université de Paris XI - Orsay, 96.

[Qia90] Xiaolei Qian. An axiom system for database transactions. *Information Processing letters*, 36:183–189, November 1990.

[Qia93] Xiaolei Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.

[Rif93] J. M. Rifflet. *La programmation sous Unix*. Ediscience, 1993.

[Sch86] D. A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Inc., 1986.

[SDA94] C. Sousa, C. Delobel, and S. Abiteboul. Virtual schemas and bases. In M. Jarke, J. Bubenko, and K Jeffery, editors, *International Conference on Extending Database Technology*, number 779 in LNCS, pages 81–95. Springer-Verlag, 1994.

[SS89] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transaction on Database Systems*, 14(3):322–368, September 1989.

[vD83] D. van Dalen. *Logic and structure*. Springer-verlag, 1983.

[WSK83] W. Weber, W. Stugky, and J. Karzt. Integrity Checking in database systems. *Information Systems*, 8(2):125–136, 1983.